The AORTA Architecture: Integrating Organizational Reasoning in Jason

Jensen, Andreas Schmidt; Dignum, Virginia; Villadsen, Jørgen

Published in:
Pre-proceedings of EMAS 2014: 2nd Workshop on Engineering Multi-Agent Systems

Publication date:
2014

Citation (APA):
Preface

This volume contains the papers presented at EMAS 2014: 2nd Workshop on Engineering Multiagent Systems held on May 5-6, 2014 in Paris, in conjunction with the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014).

We received 41 submissions. Each submission was reviewed by at least three program committee members. One submission was desk rejected because clearly out of scope. The committee decided to accept 22 papers. In this edition of EMAS, we introduced four paper categories as a way to foster the submission of papers covering different aspects of multi-agent systems engineering. The accepted papers are classified as follows: technological (7), methodological (6), analytical (5), and empirical (4). We have then grouped the papers as follows: Abstractions and primitives, platforms and architectures, methodologies, patterns, verification and testing, agent reasoning, interaction and collaboration.

The two-day program also features two invited talks by Koen Hindriks (Delft University of Technology, the Netherlands) and Maarten Sierhuis (Nissan Research Center Silicon Valley, USA).

Due to timing constraints, the papers in these informal proceedings were not significantly revised from the submitted versions. These papers will be subsequently revised (and re-reviewed) for the Springer LNAI post-proceedings.

The EMAS 2014 chairs would like to acknowledge the excellent review work done by the members of the Program Committee and their sub-reviewers. Reviews were in general detailed (and, we hope, useful to the authors), and followed by an intensive and sometimes controversial discussion among the members of the Program Committee and the Chairs.

EMAS 2014 is the second edition of the workshop and follows the successful first edition that was held in 2013 in St. Paul, Minnesota.

EMAS was formed in 2013 as a merger of three existing workshops with a long-standing tradition in the agent community: Agent-Oriented Software Engineering (AOSE), Declarative Agent Languages and Technologies (DALT), and Programming Multiagent Systems (ProMAS). EMAS is overseen by a steering committee that is responsible for the continuity of the workshop and for ensuring its quality.

We are looking forward to a lively workshop that will serve as a means to discuss ideas, exchange opinions, and initiate new collaborations.

March 18, 2014
Utrecht/Clausthal/Delft

Fabiano Dalpiaz
Jürgen Dix
M. Birna van Riemsdijk
# Table of Contents

Monday, May 5, 2014

## Session 1: Invited Talk 1

A brief history of Engineering MAS: from Mission Control to Healthcare and Autonomous Vehicles ................................. 1  

*Maarten Sierhuis*

## Session 2: Abstractions and primitives

Keeping a clear separation between goals and plans ..................... 2  

*Costin Caval, Amal El Fallah Seghrouchni and Patrick Taillibert*

Proposal for a Notion of Modularity in Multiagent Systems .......... 21  

*António Carlos Rocha Costa*

A Stepwise Renement based Development of Self-Organizing Multi-Agent Systems: Application on the Foraging Ants .......... 41  

*Zeineb Graja, Frederic Migeon, Christine Maurel, Marie-Pierre Gleizes, Amira Regayeg and Ahmed Hadj Kacem*

Capability Relationships in BDI Agents ................................. 56  

*Ingrid Nunes*

## Session 3: Platforms and Architectures

A scalable runtime platform for multiagent-based simulation .......... 73  

*Tobias Ahlbrecht, Jürgen Dix, Michael Köster, Philipp Kraus and Jörg P. Müller*

Security Games in the Field: Deployments on a Transit System ....... 92  

*Francesco Delle Fave, Matthew Brown, Chao Zhang, Eric Shieh, Heather Rosoff, Albert Jiang, Milind Tambe and J.P. Sullivan*

The AORTA Architecture: Integrating Organizational Reasoning in Jason 112  

*Andreas Schmidt Jensen, Virginia Dignum and Jørgen Villadsen*

## Session 4: Methodologies

Keep improving MAS method fragments: a Medee-based case study for MOISE+ ................................................................. 129  

*Sara Casare, Anarosa A. F. Brandão and Jaime S. Stichman*

Towards Process-Oriented Modelling and Creation of Multi-Agent Systems 146  

*Tobias Küster, Axel Hessler and Sakin Albayrak*
Environments and Organizations in Multi-Agent Systems: From Modelling to Code .................................................. 162
  Daniela Maria Uez and Jomi Fred Hubner

Session 5: Patterns

From Multi-Agent Programming to Object Oriented Design Patterns .... 179
  Mehdi Dastani and Bas Testerink

CaFé: A Group Process to Rationalize Technologies in Hybrid AAMAS Systems .................................................. 197
  H. Van Dyke Parunak, Marc Huber, Randolph Jones, Michael Quist and Jack Zaientz
Tuesday, May 6, 2014

Session 6: Invited Talk 2

Twenty Years of Engineering Multiagent Systems .......................... 213
Koen Hindriks

Session 7: Verification and testing

Efficient Verification of MASs with Projections .............................. 215
Davide Ancona, Daniela Briola, Amal El Fallah Seghrouchni, Viviana Mascardi and Patrick Taillibert

Infinite States Verification in Game-Theoretic Logics: Case Studies and Implementation ................................. 235
Slawomir Kmiec and Yves Lespérance

Side effects of agent are not just random ........................................ 253
Bruno Mermet and Gaëlle Simon

Mutation Testing for Jason Agents .............................................. 272
Zhan Huang, Rob Alexander and John Clark

Session 8: Agent reasoning

Tractable Reasoning about Group Beliefs ...................................... 288
Barbara Dunin-Keplicz, Andrzej Szalas and Rineke Verbrugge

Semantic Representations of Agent Plans and Planning Problem Domains 307
Artur Freitas, Daniela Schmidt, Alison Panisson, Felipe Meneguzzi, Renata Vieira and Rafael Bordini

N-Jason: Run-Time Norm Compliance in AgentSpeak(L) .................. 323
Jeehang Lee, Julian Padget, Brian Logan, Daniela Dybalova and Natasha Alechina

Session 9: Interaction and collaboration

Typing Multi-Agent Systems via Commitments .............................. 341
Matteo Baldoni, Cristina Baroglio and Federico Capuzzimati

Robust Collaboration: Enriching Decisions with Abstract Preferences ....... 360
Loïs Vanhée, Frank Dignum and Jacques Ferber

The Interaction as an Integration Component for the JaCaMo Platform . 378
Maicon Rafael Zatelli and Jomi Fred Hubner
Program Committee

Natasha Alechina University of Nottingham
Matteo Baldoni Università di Torino
Cristina Baroglio Università di Torino
Jeremy Baxter QinetiQ
Olivier Boissier ENS Mines Saint-Étienne
Rafael H. Bordini Pontificia Universidade Católica do Rio Grande do Sul
Lars Braubach University of Hamburg
Rem Collier University College Dublin
Massimo Cosentino National Research Council of Italy
Fabiano Dalpiaz Utrecht University
Mehdi Dastani Utrecht University
Scott A. Deloach Kansas State University
Louise Dennis University of Liverpool
Virginia Dignum Delft University of Technology
Juergen Dix Clausthal University of Technology
Amal El Fallah Seghrouchni University of Pierre and Marie Curie
Aditya Ghose University of Wollongong
Paolo Giorgini University of Trento
Adriana Giret Technical University of Valencia
Jorge Gomez-Sanz Universidad Complutense de Madrid
Christian Guttmann IBM
James Harland RMIT University
Vincent Hilaire Université de Technologie de Belfort-Montbéliard
Koen Hindriks Delft University of Technology
Benjamin Hirsch Khalifa University
Tom Holvoet Katholieke Universiteit Leuven
Jomi Fred Hubner Federal University of Santa Catarina
Joao Leite Universidade Nova de Lisboa
Yves Lespérance York University
Brian Logan University of Nottingham
Viviana Mascardi University of Genova
Philippe Mathieu University of Lille 1
John-Jules Meyer Utrecht University
Frederic Migeon IRIT
Ambra Molesini University of Bologna
Pavlos Moraitis Paris Descartes University
Haralambos Mouratidis University of East London
Jörg P. Müller Clausthal University of Technology
Peter Novák Delft University of Technology
Andrea Omicini University of Bologna
Juán Pavón Universidad Complutense de Madrid
Alexander Pokahr University of Hamburg
Additional Reviewers

G
Gou, Yingzhi

K
King, Thomas Christopher
Konnerth, Thomas

N
Noroozian, Arman

R
Ribino, Patrizia

S
Secq, Yann

T
Taillibert, Patrick
Testerink, Bas

Z
Zateli, Maicon
A brief history of Engineering MAS
From Mission Control to Healthcare and Autonomous Vehicles

Maarten Sierhuis
Nissan Research Center Silicon Valley, United States

In this talk I will give a brief history of my experience and learning to engineer Multi-Agent Systems for NASA, for healthcare and for developing autonomous and connected vehicles. For over 12 years, Maarten was a senior researcher in human behavior modeling and agent systems at NASA Ames Research Center. He also headed the Knowledge, Language & Interaction group at (Xerox) PARC. Maarten is a co-founder of Ejenta, a San Francisco startup, developing a cloud-based sensor and agent platform. Currently, Maarten is leading the new Nissan Research Center Silicon Valley in developing autonomous and connected vehicles.
Keeping a clear separation between goals and plans

Costin Caval\textsuperscript{1,2}, Amal El Fallah Seghrouchni\textsuperscript{1}, and Patrick Taillibert\textsuperscript{1}

\textsuperscript{1} LIP6, Paris, France, \{costin.caval, amal.elfallah, patrick.taillibert\}@lip6.fr,
\textsuperscript{2} Thales Airborne Systems, Elancourt, France

Abstract. Many approaches on BDI agent modelling permit the agent developers to interweave the levels of plans and goals. This is possible through the adoption of new goals inside plans. These goals will have plans of their own, and the definition can extend on many levels. From a software development point of view, the resulting complexity can render the agents’ behaviour difficult to trace, due to the combination of elements from different abstraction levels, i.e. actions and goal adoptions. This has a negative effect on the development process when designing and debugging agents. In this paper we propose a different approach that aims to provide a more apprehensible agent model with benefits for the ease of engineering and the fault tolerance of agent systems. This is achieved by imposing a clear separation between the high level reasoning that handles goals and the actual plans that contain the actions of the agent. The adoption of sub-goals inside classic plans is therefore forbidden. The approach is illustrated using two theoretical scenarios as well as an agent-based maritime patrol application. We argue that by constraining the agent model we gain in clarity and traceability therefore benefiting the development process and encouraging the adoption of agent-based techniques in industrial contexts.

Keywords: goal directed agents, goal reasoning, goal-plan tree

1 Introduction

In the field of intelligent agents, BDI agents are used extensively due to their pro-activeness, adaptability and similarity between the abstract representation and human reasoning. These agents are enticed with beliefs to cover their view of the world, a reason for their behaviours in the form of desires or goals, and a description of the means to act, in the form of plans or intentions.

The classic BDI model proposed by Rao and Georgeff [1] specifies a cycle that considers the options for desires, deliberates on them to update the existing intentions and then executes the actual actions. In more practical approaches, automata are used to handle the life-cycle of goals from their adoption to the appropriate plan selection and execution [2, 3].

In practice, various works [3, 4] and programming frameworks (Jason [5], Jadex [6] etc.) employ a model where plans can adopt new goals, often termed
sub-goals. The semantics of this structure is that for a goal or sub-goal to be satisfied, only one of the plans needs to be completed successfully\(^3\), while for a plan to be successful, all of its sub-goals have to be achieved. This recursive construction has the advantage of using already existing BDI building blocks and can help abstract certain aspects of an agent’s behaviour offering the possibility to define the agent in a top-down approach. However, it also creates a structure which is difficult to trace and whose depth may be unpredictable. Important aspects in the behaviour of an agent might be hidden from the eyes of a developer or reviewer due to this intricate design. One might always wonder whether the current plan is a terminal one or whether the model continues with further sub-goals. Given that the adoption of a goal usually implies a new reasoning process with an automaton and further plans, it shouldn’t be treated the same as an atomic action.

Winikoff et al. [7] and Dastani et al. [8] highlight the difference between the declarative and procedural aspects of the model, i.e. between goals and plans. However, this delimitation is diminished if the goal and plan levels are interwoven.

While at runtime it is inevitable to alternate between the reasoning level and the plan level, it is much less natural to design a BDI agent using a similar approach where these levels of abstraction follow in turn. Furthermore, agent oriented development methodologies such as Tropos [9] and Prometheus [10] have a top-down approach where they start with main goals and construct a hierarchy of sub-goals before defining plans and other details. Implementing agent systems modelled using such methodologies would also be more natural if goal-plan separation is applied.

To address these issues we propose a model for goal reasoning that simplifies the agent representation by requiring a clear delimitation of the levels of abstraction in an agent’s definition. We call the approach Goal-Plan Separation, or GPS.

This paper is organised as follows. Section 2 presents the original approach of the paper which is illustrated on a generic example. In section 3 a Mars Rover scenario is studied with respect to the GPS. Section 4 discusses implementation issues while Section 5 some aspects of the goal execution. Section 6 presents an experimentation in the domain of maritime patrol. In Section 7 we discuss some fault tolerance issues with respect to the experimentation. Section 8 addresses the related work and Section 9 concludes the paper.

## 2 Goal-Plan Trees vs. Goal-Plan Separation

Thangarajah [4, 11] formalises the representation of the agent model introduced above in the form of an AND-OR tree, the goal-plan tree, or GPT. Goals are OR nodes since only one plan suffices for the achievement of a goal, and plans are AND nodes in order to denote the obligation to achieve all goals for a

---

\(^3\) There is often an achievement condition that also has to be validated
Fig. 1. a) Left - An example of Goal-Plan Tree, b) Right - A goal-plan separation of the same example.

Successful plan execution. Furthermore, two operators are added to the plan node, to indicate either that the goals have to be achieved in sequence (:) or in parallel (||). An example which illustrates all these is given in Figure 1 a). Here, the GPT using the two operators spreads in depth across several levels. Note that there can be more than one tree for a given agent, in other words more than one root goal.

To illustrate the Goal-Plan Separation approach, the generic example we used above was modified to obtain a possible goal-plan separation, as seen in Figure 1 b). The new representation separates the two abstraction levels. To save space, we consider that that the default operator for the AND nodes is the sequence operator, unless stated otherwise, e.g. in the case of SG23. To preserve the original structure, goals are also allowed to be OR nodes, in order to depict cases where a goal or sub-goal can be achieved in more than one way. Similarly, goals that have more than one plan are OR nodes. While the original goals were preserved, the plans that were not leaves were replaced by sub-goals, e.g. SG11.

To compensate, plan names of the form P’ were used to indicate a variation of an original P plan which at least removes the goal adoptions. Note, however, that this exact transformation is not unique for the given example as it depends on the plan’s specific features. SG12 was introduced to avoid the existence of siblings of different types. This example shows that transforming an existing agent is possible. Nevertheless, as is the case with many such translations, it is likely that a complete redesign of the agent would produce a more appropriate result.

---

4 E.g. a plan that turns on a sensor, adopts a goal to retrieve data and then saves that data. Such a plan would rather transform into a main goal with three sequential sub-goals, the first corresponding to the beginning of the original plan, and the last corresponding to its final part.
It is important that sub-goaling can be retained in the model, albeit in a different form (i.e. in a goal hierarchy rather than inside plans), and therefore the advantages of hierarchisation for top down design be preserved.

In this paper, we will call goal reasoning the part of an agent’s specification that contains no plans or actions at all, as can be seen in Figure 1 b) where it is delimited by a rectangle with rounded corners. However, as will be discussed further on, other mechanisms can appear at this level, e.g. for handling events or various types of goal dependencies.

In [12], GPTs are used as support for a study on plan coverage and overlap, with the hypothesis that the plan libraries discussed have no cycles. This is important to note as in the general case adopting goals inside plans may produce cycles, sometimes even with unwanted consequences similar to the infinite loops in classic programming. We, on the other hand, do not restrict cycles, as will be seen in the scenario in Section 6. The goal-plan separation does however present the advantage of not allowing cycles created through plans.

3 Mars Rover scenario

To illustrate the GPS, let us consider a Mars rover example from [11]. Figure 2 represents a goal-plan tree for a Mars rover’s goal to analyse soil samples. The tree’s depth varies between $P7$: \textit{ExpSoilByDelegationPlan} that is at a depth of one and $P6$: \textit{TransmitTo(Lander)Plan}, at a depth of 5. While all leaf nodes are plans, there are also intermediary plans which adopt goals. It is the case of $P1$: \textit{ExpSoilBySelfPlan} and $P4$: \textit{RecordResultsPlan}, both of which having terminal
Fig. 3. A possible translation of the Mars Rover scenario

plan nodes at the same high: P7: ExpSoilByDelegationPlan for P1 and P2: Collect(Soil)Plan as well as P3: Analyse(Soil)Plan for P4. The tree is therefore highly imbalanced and mixes plans and goals between its various levels.

One way to apply our approach to this scenario is to translate it into a form where goals appear only as leaves of the tree, the same as we did with the example in section 2. In this case, as depicted in Figure 3, as the plans are more specific, we will assume that neither P1 nor P4 contain any actions and so a simple transformation would have P1 transformed into a sub-goal and P4 disappear completely as there is already SG3 to regroup the corresponding sub-tree. For P7, a parent sub-goal SG12 is created to avoid having two siblings of the G1 node of different types (i.e. a goal and a plan). SG12 also carries the precondition originally contained by P7. Another approach would be to rewrite the Mars Rover’s behaviour in a format similar to the goal diagram from Tropos [9], as in Figure 4. The representation can also be seen as a type of plan. It starts with a decision node that corresponds to P7’s precondition from the original scenario. The sequence operator is represented through the arrows that depict the dependencies between goals, while the parallelism is implied through the fact that two arrows start from the same entity, i.e. SG2. Both versions of the scenario obey the GPS principle as in both the two levels, the goal reasoning level and the plan level, can be clearly distinguished. This shows the applicability of the goal-plan separation is not restricted to a specific goal reasoning formalism.

4 GPS method implementation

Throughout the evolution of programming, languages and development tools often advanced by limiting the programmer’s freedom to access lower level el-
Fig. 4. A modified representation of the Mars Rover scenario with a clear goal-plan separation.

ments such as registers and pointers to data, and offering in exchange higher level tools and constructs such as variables and dynamically created references to data. These evolutions allowed for the creation of increasingly complex systems while decreasing the possibilities for coding errors. Similarly, we do not refrain from restraining the freedoms of the programmers and designers in the interest of clarity and reliability.

To achieve the goal-plan separation, rather than adopting sub-goals, an agent’s executing plans would accomplish the necessary tasks and then relinquish control to the higher level where the following necessary goal is adopted. This creates, as illustrated in the examples above, a distinct goal reasoning level where an agent’s goals are chosen.

As the Goal-Plan Separation approach in its simplest form is the requirement to keep a clear distinction between the two abstraction levels, it is general enough so that it can be applied using any of the BDI frameworks that allow goal adoptions in plans, such as Jason and Jadex. The important condition, however, is to restrict goal adoptions inside plans that are themselves being executed for the achievement of a goal.

4.1 Reasoning through a goal plan

As required by the GPS method, the goal reasoning level should be kept separate from the plans that handle action composition. Considering that relations between goals can be similar to those between actions, we can envisage to use a modified plan language to represent the relations between goal adoptions. We call these plans that handle goal composition goal plans and we oppose them to classic plans.

A goal plan is an oriented graph defined as follows:
Fig. 5. A goal plan for the Mars Rover example

\[ GP = \langle N, E \rangle \quad // \text{goal plan} \]
\[ N = A \cup O \cup T \quad // \text{nodes} \]
\[ A = \{ \text{adopt}(G) \mid G \in \text{Goals} \} \]
\[ O = \{ \text{op} \mid \text{op} \in \{ \text{startNode}, \text{finishNode}, \text{AND}, \parallel \} \} \]
\[ T = \{ \text{test}(\text{stateCondition}) \mid \text{stateCondition} \in \{ \text{Beliefs}, \text{Events} \} \} \]
\[ E = \{ n_1 \to n_2 \mid n_1, n_2 \in N \} \quad // \text{edges} \]

The three node types allowed in the graph are:

- \( A \), the goal adoption nodes, as the unique action allowed in the goal plan. Each node represents the invocation of an automaton associated with the goal.

- \( O \), the operator nodes, with operations including a unique start node and at least one finish node. There is also an operator for branching parallel threads and one for the logical condition \( \text{AND} \) that can be used to synchronise threads or to indicate the obligation of two or more conditions to be all true, for example to require several goals to be achieved in order for the execution to continue.

- \( T \), the condition test nodes that can handle state conditions for belief values and events such belief change and message arrival.

Edges indicate the succession of nodes in the goal plan and cycles are possible, for example to indicate a recurrent goal adoption.

The Mars Rover scenario in Figure 4 with its inline goal dependencies can easily be transformed into a goal plan, as seen in Figure 5. There are two possible finish nodes, with one for a successful mission where either \( G7 \) or both \( SG4 \) and \( SG5 \) were achieved, and one to indicate all other cases as failures.

While implicit relations between entities may be enticing due to their ease of definition and generality, they are also difficult to follow and may hide unwanted
interactions. Hence, this reasoning model favours the use of explicit specifications of dependencies between goals. If for example a Mars rover needs to perform an experiment at a location $X$ and it has two goals for achieving this, one being $G_1 = \text{“move to } X\text{”}$ and the other $G_2 = \text{“drill”}$, then it is clearer to link the adoption of $G_2$ to the successful achievement of $G_1$ rather than for example the belief that the rover is at location $X$.

In a framework like Jadex, this model can be implemented using a plan that is triggered at agent’s birth. The plan would specify the dependencies between sub-goals and adopt them without any other actions.

This kind of reasoning is suitable for agent systems where the behaviour can be thoroughly specified at design time so that all dependencies can be accurately included. Adding new goals and other modifications, however, are difficult to apply.

### 4.2 Alternatives for representing goal reasoning

**Reasoning through rules** This approach is the opposite of the goal plan, in that it uses implicit relations between goals. Using the goal trigger rules, a dependency tree similar to the reasoning above can be constructed at runtime.

This reasoning model can be implemented in Jadex by simply specifying trigger conditions for each goal but without creating explicit connections between them.

The advantage of this approach is that the representation can handle more complex systems that act in highly dynamic environments, with new goals added effortlessly.

**Reasoning using a planner** Rather than having goals simply triggered by rules, a planner can be used to select among goals and create a sort of goal plan. The difference from first reasoning type described above is that this time the goal plan would be dynamically created and would easily adapt to various changes and include new goals. Another difference is that a planner would render the agent proactive as it would not have to wait for events in order to act. The job of the planner would be to select, order and parallelise goals and for this it would use certain operators [13]. The example in Section 6 does not correspond to this method as no planner is used and its goal plan is defined at design time.

### 5 Execution

While not explicitly presented in the GPT, as seen in Figure 6, between the goal and plan levels there is usually a goal automaton [2, 3] which handles the goal life-cycle. This life-cycle usually starts with the adoption of the goal and includes the choice and execution of plans.

An example of goal lifecycle for which an automaton is used is depicted in Figure 7. It uses a series of beliefs for state changes, such as desirable to indicate the presence in the automaton, selected to indicate the passage in an active state
and satisfaction that indicates if the goal was achieved. We use these beliefs to control the execution of goals by linking them to other beliefs that justify them, for example the goal adoption conditions for desirable. In case any of these conditions is no longer valid, the belief is no longer justified so the automaton changes its state automatically, which in the case of the desirable belief means that the goal is aborted. If we take the example in Figure 5, supposing that during the execution of G7 the condition FreeRover(X) is contradicted by an observation, the adoption of the goal will no longer be justified and the goal will fail automatically.

The beliefs can also be used to control the goal automaton from the higher level in a more straightforward manner, if for example we added another operator that causes a goal to abort its execution.

For the GPS approach the automaton is a black box that is given a goal to adopt and possible plans to execute and this is why we represent only goals and plans in our modelling examples. The execution can cause side effects such as belief changes that can lead the reasoning level to take actions with respect to current goal or even the adoption or execution of other goals. For example, this can cause the goal to be aborted in case it is estimated to take the agent in an unsafe state, or it can cause the adoption of a reparation or compensation goal to counter certain unwanted effects. Note that several automatons can function at a given moment as parallelism is allowed in our method. Conflict management is however not in the scope of this paper.

6 Experimenting with GPS

The GPS approach has been experimented in an industrial context at Thales Airborne Systems on an application designed for experimenting on AI in general and more precisely on Interval Constraints propagation and on multi-agent systems (MAS). The purpose of this application (Interloc) is the localization of boats from a maritime patrol aircraft. It is implemented as a MAS and can contain dozens of agents implemented as Prolog processes.
Interloc was initially designed as a set of non goal-directed autonomous agents. This means that the agents had only one goal that was achieved through a set of associated plans. Subsequently, it was redesigned in order to improve the autonomy level of the agents using a goal directed approach. A clear separation between the goal and plan abstract levels was sought as a means to achieve a better understanding of the behaviour of the agents (intelligibility).

The implementation technique used was the one identified as goal plan in Section 4. This means that a plan was designed where the only possible action was goal adoption (and also sub-plan activation whose only possible action was also goal adoption). The intention of the designer (prior to the GPS methodology presented in the present paper) was to exhibit an abstract (goal) level describing the main features of the behaviour of agents so that the only reading of the goal level description was sufficient to understand the salient behaviour of the agents.

We first present the application itself, then the goal plan of one of the main agents, the aircraft, abstracted as a Petri net [14] and finally a discussion about the advantages of the GPS approach in the specific case of the Interloc application.

6.1 Interloc

The main goal of the application is the localization of boats using a goniometer\(^5\) on-board a maritime patrol aircraft. The sole use of a goniometer allows for a stealth detection (i.e. detect without being detected) of boats which is important for some missions such as gas-freeing prevention\(^6\). If the boats were steady, the problem would be simple. The fact that they move obliges to rely on non-linear regression methods (as is the case of existing commissioned implementations).

\(^5\) Tool which measures the angle between the boat and the north pole

\(^6\) Deterring tankers from polluting the environment by cleaning their fuel tanks at sea
or interval constraint propagation (Interloc). Most of the agents, i.e. boats, the goniometer and data visualization agent, have been designed for the purpose of simulation. The main agent (aircraft) must follow all the boats visible from its location, compute in real-time their position by accumulating bearings and interacting with computation agents (more precisely artifacts [15]) operating interval propagation, adapt its trajectory to observations and contingencies and transmit results to the visualization agent. For the patrol aircraft, boats may appear or vanish at any time. Several aircraft might be present at the same time, but so far they do not communicate with each other. Typically 20 to 30 agents or artifacts are active in the system at a given time.

6.2 The aircraft agent

Boats and aircraft have been designed following the GPS method. We present here the aircraft which is the most complex and hence the most interesting for illustrating the methodology.

Five goals were identified:

– **Initialisation** of the system. The goal is to get data related to the aircraft trajectory (pre-defined, planned or human-guided) and various parameters characterising the simulation.

– **Move**: Execute one step forward

– **Measure**: Initiate measurement of the bearing of all the visible boats

– **Treat**: Processing of a received measurement

– **Visualization**: Processing of a single request from the visualization agent

The sole knowledge of the various goals present in the system is not sufficient to understand its behaviour. One must also describe the way these goals are adopted and what happens when they are achieved, for example by specifying their chronology, conditions for becoming a desire, conditions for becoming an intention. This knowledge may be provided in different forms, corresponding to the different ways of applying the GPS approach. In Interloc we designed a plan, *i.e.* a goal plan, to specify this behaviour.

Informally, the goal plan is the following (A more formal description of this plan is given in Figure 8 as a Petri net): the achievement goal *initialization* is adopted. If the goal is not achieved, the system is halted. Else, the four sub-goals main_move, main_measure, main_visualization and main_analyse are activated in parallel thanks to the corresponding goal plan.

The main_move sub-goal:

– Wait for a move_time_step delay

– Adopt the move goal (which associated plans will compute and execute the next time step)

– Wait for the move goal achievement

– Loop.

The main_measure sub-goal:
Adopt the \textit{measure} goal (the associated plans will measure the bearings of all the visible boats which implies interactions with the measurement artifact and the (simulated) boat agents).

Once achieved, the goal will be re-adopted after a given time delay.

The \textit{main_analyse} sub-goal:

- Wait for a measurement. They arrive randomly after a request measurement is issued.
- Record the newly present boats
- Adopt the goal \textit{treat} (the associated plan will generate a constraint to be added to the previously received measurements and send it to an interval constraint propagation artifact which will compute a more and more precise boat location)
- Loop, in order to process waiting measurements.

The \textit{main_visualization} sub-goal:
- Wait for a request from the visualization agent
- Adopt the visualization goal in order to process the request
- Wait for the achievement
- Loop to process pending requests

6.3 Discussion

With GPS, iterative and timed behaviours appear at goal level: in the pre-GPS version of the application, the natural tendency was to incorporate dynamic aspects in goals, making them fairly complex. For instance, the move goal was not conceived as a single step as presented above, but as the complete management of the aircraft’s trajectory, including the loop sequencing individual steps. The move time-step, which is important in the global understanding of the behaviour of the aircraft, was buried in the plan achieving the goal. In the GPS version, deciding to rewrite it as a simpler goal - i.e. the achievement of a single movement step - created the need for the definition of the time-step and the iterative behaviour at the goal plan level. The fact that such details are at an upper level of abstraction emphasises their importance and improves the understanding of the agent behaviour.

With GPS, relevant perceptions of the environment are required at the goal reasoning level: it is the case of messages coming from the visualization or the measurement agents. Here again, it emanates from the fact that some perceptions are essential for the global understanding of the agent behaviour. In Interloc, measurements trigger the adoption of a goal whose achievement is more or less secondary since other measurements can occur rapidly. That is the reason why it seems to be a good approach to handle these measurements at the upper level of abstraction. A filtering strategy can also appear in this goal plan, possibly by the adoption of a specific goal prior to the adoption of the measure goal itself.

With GPS, handling errors is easier to take into account: this is because errors, whatever their cause, often manifest through the failure of goals. This provides an adequate range of exception mechanisms in the language in which plans are written. Hence, the programmer’s effort with regard to fault tolerance is mainly to take into account the processing of non-achieved goals. Of course, this does not concern the goal plan itself, which has to be designed traditionally by explicitly introducing fault tolerance actions. However the amount of code regarding the classic plans is far greater than the amount of the goal plan code. In the Interloc application, no specific fault tolerance effort has been carried out but a clean processing of non-achieved goals in order to stop the system rather than have it crash. As a consequence, application debugging was greatly facilitated. For the same reasons, the GPS approach proved to facilitate the evolution of the multi-agent system. Thus, the aircraft agent was easily changed into a UAV (Unmanned Autonomous Vehicle), with a larger autonomy in the trajectory choice. Here again, the abstraction obtained by separating goals and plans seems to be the reason.

In Interloc, we used an in-house agent plan language (Alma) to implement goal plans. All the required primitives were available, since a goal plan is a type
of plan. Nonetheless, it appears that specific primitives could be introduced to facilitate the programming of the goal level. These concern mainly iterative and time-controlled behaviours.

7 Discussion on the fault tolerance with goal reasoning

In real life applications agents tend to have more refined representations than the ones discussed in Sections 2 and 3. In particular when it comes to handling errors, the specification easily grows in complexity as specific cases have to be taken into consideration [16]. Goals give agents a level of abstraction that is beneficial for a system’s robustness as errors, exceptions, anomalies etc. usually occur during plan execution which, in a robust system, only cause the plan to fail and the goal automaton to react normally and reattempt to achieve the goal. While there are studies that treat the more general case of partial goal satisfaction [17], if we only consider a binary goal definition, a goal’s adoption has only two possible outcomes at reasoning level: the goal is either achieved or not. Requiring the programmer to specify not only the actions to take after the achievement of a goal, but also the actions to take in case the goal fails enhances the reliability of the agent without dramatically increasing its complexity.

In the Mars Rover scenario represented in Figure 5, the failure to delegate the task to another agent (i.e. the failure of G7) causes the Rover to attempt to accomplish the mission by itself through the adoption of SG1. Similarly, in the aircraft specification of the Interloc application, both the successful achievement and the failure of goals are represented in the Petri net and also in the implementation. However, for simplicity reasons, no special actions are taken and the only result of a goal failure is to ensure the agent does not reach unforeseen states. Also, the current format implies an infinite life for the agent, which is not necessarily desirable in a real application.

In the paper cited above [17], goal satisfaction is evaluated using a progress metric. Partial goal satisfaction could be integrated with our model by enforcing the coverage of the whole range of possible values for the progress metric used. For example for a surveillance goal, instead of specifying success and fail behaviours, it could be interesting to estimate the percentage of the assigned area that was covered and to use thresholds for the desired behaviours: less than 30% would be considered a mission failure with the area announced as unsafe, a coverage between 30 and 80% would require a call for backup to finish the job, while a coverage of more than 80% would be considered a success. Note that this does not concern the intermediary stages such as those that are handled by the goal automata, but final goal failures, i.e. when all alternatives have been tried and no positive outcome resulted.

7 In this case, we understand by robust an agent system in which an error or exception in a plan is caught and only causes that plan to fail, while the rest of the agent continues to function normally, i.e. does not cause the whole agent to fail.
8 Related work

While we discuss the goal reasoning level in the need to better organise the levels below, Morandini et al. [18] approach the same level from a different perspective: the need to fill in the gap between goal based engineering and goal implementations. They propose a tool for transforming a Tropos representation into Jadex code, for which they introduce a formalism based on rules for the life-cycle of non-leaf goals in a goal hierarchy. This segregation between leaf and non-leaf goals creates a goal level that corresponds to our goal reasoning level and their work is consistent with the GPS approach. This further confirms our earlier statement with respect to the utility of a goal-plan separation for the implementation of goal based methodologies. One of the interesting aspects is that Morandini et al. take into account the fact that even if the sub-goals are achieved, the parent goal may still fail due to its own achievement condition, which is often not taken into consideration when discussing the Goal-Plan Trees. While this formalism is rich and GPS-compliant, as our application example shows, our approach aims to provide a model that allows for a more complex representation, with more diverse goal relations, event-based goal reasoning and time constraints.

The aspect of the goal-plan separation that handles goal reasoning is situated at what Harland et al. [3] and Thangarajah et al. in earlier works [4, 11] call agent deliberation level. It is where agent goals are considered, which constitutes the point where goals start their life-cycle. It is the same level where top level commands are issued to interfere with the goal life-cycle, for example when deciding to drop or suspend the goal. As they point out, goal deliberation can deal with issues like goal prioritisation, resource management and even user intervention. These aspects are beyond the scope of this paper but can be considered for future developments of our approach.

In [10] the authors praise the GPT formalism for its capacity to accommodate a high number of achievement alternatives for a root goal. However, their Prometheus methodology better suits the goal-plan separation model because their approach progressively refines the agent specification from higher abstraction levels downwards to plans and data structures and therefore these levels are easily separable. It is interesting to note that in [3] changes in goal state have preference over any executing plans. Similarly, the agent’s deliberation should take precedence over the other lower levels which it controls, namely the goal life-cycle automata and plan execution.

The goal-plan trees have been used in various works for representing agent specifications and as a basis for further treatments. For example [4] use the GPTs to gather resource requirements called summary information and identify possible goal interactions. This is due to the hierarchical structure of the tree where summary information can be propagated upwards towards the root of the tree. Further works on the subject [3] reuse the model to illustrate their operational semantics for the goal life-cycle. Furthermore, Shaw et al. propose different approach for handling goal interactions using Petri Nets [19] and constraints [20] instead of GPTs. These, as well as other works that use GPTs, such as [21] on
intention conflicts, can be used with GPS, but the actual extent of the required adaptations would have to be studied for each case.

Another representation used for resource handling is the task expansion tree described in [22]. This tree represents the decomposition of a task (a concept similar to goals in our work) into subtasks. The particularity is the introduction of special composite tasks that are used to compose other tasks in a functional manner. These include, besides the sequence and parallel operators present in the GPT model described in this paper, other tasks that allow other types of branching and tests. The use of these operators in a tree structure situates their model between classic goal hierarchies and our goal plan.

Singh et al. [23] use learning for plan selection in BDI agents. They also use GPTs to describe the agents and even note briefly that leaf plans interact directly with the environment, which is consistent with the GPS approach. This allows for a representation where, given the results - i.e. success or fail - of the executions of all leaf nodes, the success or failure of the root node is decided by simply propagating these logic values in the AND-OR tree. This is a confirmation of the benefits of the GPS approach, since in a more general case, including actions in intermediary plans can mean that even if all sub-goals of a plan are achieved, the plan does not necessarily cause the achievement of its parent goal. The GPT is therefore already a simplification of the system, as it uses the rather strong hypothesis that there are no perturbations, such as the one in the aforementioned case, in the AND-OR tree.

In [24] Pokahr et al. address the issue of goal deliberation but do not address the level of definition discussed here as they consider only goals that have already been adopted. While preceding the research cited above, their work focuses on the similar issue of goal interactions (i.e. when goals interfere positively or negatively with each other) and they base their proposed strategy on the extension of the definition of goals. They include for example inhibition arcs that block the adoption of a certain goal or type of goal when another goal is adopted. Such mechanisms can be integrated when specifying the goal reasoning level discussed in our approach.

The goal automaton proposed by Braubach et al. [2] presents a goal state labeled New with a Creation condition acting as a triggering condition for the goal before the adoption and the actual goal life-cycle. This state, together with the condition are at the level of our goal reasoning level. A goal that was defined for the agent is considered to be in the New state, as opposed to a goal that can for example be received from the exterior or generated through the agent reasoning. Only when such a goal is received does it pass into the New state. All the goals discussed in the examples in this paper are already in this state.

Note that, while we use the GPT representation to justify our approach, the GPS is concerned with more general agent models. Also, this paper does not argue against the GPT formalism, neither does it dispute the plethora of works that use it as a model, but rather discusses the more general issue of specifying agents with interwoven goal and plans levels. The current paper complements the cited works on goal interactions as it concerns the agent specification rather
than the runtime mechanisms that aim to improve the efficiency, proactivity, reactivity etc. of the agents.

9 Conclusion and future work

In this paper, we argued that the separation between goals and plans is important for the specification and construction of BDI agents. It was shown that the possibility to mix atomic actions with goal adoptions in various agent models and languages can have negative effects on the resulting representation and can hinder the development process. A series of examples illustrated what an agent would look like when complying with the Goal-Plan Separation approach. The importance of tidy agent representation lies with the ease of development, which can, in turn, facilitate the wide-scale adoption of the development model.

As discussed in the paper, on the side of BDI agent modelling there are many studies on goal representations and goal life-cycles. However, the higher level that is placed above these automata is less examined in the literature and constitutes a point of this paper that we plan further study. For this, a more in-depth research on specifying the agent’s goal reasoning will have to be undertaken. Among other primitives, the handling of temporal constraints is important for agent systems and should be taken into consideration. Furthermore, as stated above, there are fault tolerance aspects related to this direction in agent development that can be exploited. An empirical evaluation of the approach and its advantages on agent design will have to be undertaken in order to provide further show of the interest of GPS. In the long run, the goal is to integrate this approach in an agent development methodology.

References


Proposal for a Notion of Modularity in Multiagent Systems

António Carlos da Rocha Costa

Programa de Pós-Graduação em Computação
Centro de Ciências Computacionais
Universidade Federal do Rio Grande – FURG
96.203-900 – Rio Grande, RS, Brasil
ac.rocha.costa@gmail.com

Abstract. This paper builds on the idea that lack of an appropriate notion of modularity is the main limitation preventing the intensive adoption of MAS technology by Software Engineering methods and tools in the development of conventional software, and preventing a wider and fuller exploitation of such methods and tools in the development of MAS themselves. The paper distinguishes between agent organizations and agent societies, and proposes agent organizations as the proper foundation for the notion of MAS module, for both agent societies and conventional software systems. Additionally, a notion of functionalist specification of agent organizations is introduced, such that for any given functionalist specification, a corresponding conventional functional specification can (possibly) be found, so that in certain cases, a suitably specified and implemented agent organization can be seamlessly integrated to (and correctly operate as an encapsulated module in) either an agent society or a conventional software system. Given that the proposed functionalist specifications are to be based on notions of functional rights and functional duties of agent organizations, the need for normative environments in agent societies modularized in terms of agent organizations is, next, made clear. Finally, a few complementary principles, necessary for the full exploitation of the notions introduced in the paper, are also indicated.

1 Introduction

This paper issues from the idea that the main difficulty for the intensive adoption of MAS technology by usual Software Engineering methods and tools, in the development of conventional software systems (and also for a wider and fuller adoption of those methods and tools in the development of MAS themselves), lies in the lack of an appropriate notion of modularity for agent systems.

The paper proposes, then, a notion of modularity for multiagent systems that seems able to both (i) leverage the systematic development of multiagent systems, and (ii) ease the incorporation of MAS modules into conventional software systems.

* Work partially supported by CNPq and FAPERGS.
The paper starts, on the basis of previous work, by presenting a conceptualization that makes a difference between the notions of agent organization and agent society. Then, it introduces the notions of functional rights and functional duties for agent organizations, and explains how agent organizations may be specified in a functionalistic way, in terms of functional rights and duties.

Next, the paper shows how agent organizations can be taken as a proper foundation for a notion of MAS module, capable of supporting both (i) the structuring of agent societies in modular ways, and (ii) the seamless integration of MAS modules into conventional software systems.

Following that, a few complementary principles for the application of agent organizations to the modular structuring of MAS and for the incorporation of MAS modules into conventional software systems are presented.

Finally, after discussing related work, the paper’s conclusion briefly considers some prospects both for the future of agent societies modularized in the vein suggested in the paper, and for the adoption of MAS modules by usual Software Engineering methods and tools, as a means for the incorporation of MAS technology in the development of conventional software systems.

2 Agent Societies and Agent Organizations

We start to deal with the problem of modularity in multiagent systems by adopting definite notions of agent society and agent organization, notions that will be central to our proposal.

For the meaning of the term agent society we take the meaning introduced in previous works (cf., e.g., [1]):

An agent society is a multiagent system that is open, organized, persistent, and situated, where:

- openness means that agents can enter and leave the society freely;
- organized means that sub-sets of agents joint together in “sub-systems” that interrelate in a systematic way;
- persistence means that the organization of the society persists in time, independently of the entering or leaving of the agents, or of the agents changing their behaviors or interactions, due to learning or other motive;
- situatedness means that the society operates in a definite physical environment.

We note that, for a preliminary treatment of the issue of modularity, we will need only to deal with the properties of organization, openness, and persistence.

For the concept of an organization, we adopt B. Malinowski’s notion of institution [2] (which we equate with that of organization, in this paper):

An agent organization is a sub-system of an agent society, characterized by the following features:
– a charter, that is, a specification of the organization’s goals and structural composition;
– a personnel, that is, a set of agents that act and interact within the organization, structurally related to each other according to its charter;
– an activity, that is, an internal functioning, resulting from the combined action and interaction of the organization’s personnel;
– an apparatus, that is, a set of means with which the organization interacts with the environment in which it can be placed;
– a set of norms, that is, a set of rules specifying the range of acceptable variations both in the behaviors of (and the interactions among) the personnel of the organization and in the behaviors of the organization seen as a whole (and in the corresponding interactions of the organization with its environment);
– a set of functions, that is, a set of ways in which the functioning of the organization contributes to satisfy the operational needs of other organizations in the society, and of the society as a whole.

We remark that the term organization is ambiguous and may be used in two main ways, in connection to multiagent systems: it may be used either to denote a property of a multiagent system (the way the multiagent system is organized in terms of “groups” or “sub-systems” of agents – cf. the first definition above) or to denote such groups or sub-systems of agents, when the multiagent system is an agent society (in this case, being equivalent to “entity”, “corporation”, or “institution”, in human societies, e.g., an industrial company, a social club, or a university – cf. the second definition above).

Thus, to avoid this ambiguity, in the following we will always refer to organizations as agent organizations, as we did in the second definition above, when we want to refer to the second meaning of the term.

3 Modularizing Agent Societies

3.1 The Locus of Modularization in Agent Societies

In this section, we first summarize the main features of the PopOrg organizational model of agent societies that we have been developing. Then, we make use of that model to indicate the structural level within the organization of agent societies where the proposed notion of MAS module is to be introduced.

The PopOrg model of agent societies was first introduced in [3], and explored and elaborated in several directions further works (e.g., [4–6, 1, 7]).

For the purpose of indicating the appropriate structural level for the notion of MAS module, it is enough to consider the structural dimension of the PopOrg model, which is pictured in Fig. 1.

Essentially, we have that an agent society is structurally organized, according to the model, as a two-fold structure, encompassing:
– a populational level, comprising the set of agents that inhabit the society;
Fig. 1. The PopOrg picture of the structural dimension of agent societies.

- an organizational level, comprising all social structures that help to organize the behavior and interactions of the society’s population.

The organizational level, on the other hand, is itself structured as a three level structure, with:

- a micro-organizational level, comprising the set of social roles that the agents of the society may perform, together with the set of social interactions among those roles;
- a meso-organizational level, comprising the set of agent organizations that arise in the society (as a result of organizational connections among social roles), together with the set of interactions that those organizations perform between each other, for the purpose of realizing their own goals, or for performing functions for other organizations (or, for the society as a whole);
- a macro-organizational level, comprising the set of social systems through which the society as whole performs the main functions necessary for its maintenance and evolution (production and distribution of goods, socialization of new members, improvement of its accumulated knowledge base, etc.).

We say that the types of interactions mentioned above are organizational interactions, and that organizational interactions between the mentioned above organizational components (social roles, agent organizations, social systems) create organizational links between the components that perform them.
3.2 Agent Organizations as MAS Modules

The concept of MAS modules that has been proposed in the classical literature of MAS is the concept of agent itself, and this since at least the concept of agentification, introduced in [8] (cf., also, [9] – but see Sect. 7 for more recent proposals).

Taking into account the above three-level organizational structure of an agent society it should be clear that an appropriate notion of MAS module should reside in either the micro or the meso level, for any concept of basic system module should preferably reside at the lowest possible structural level in the system.

This leaves either the micro or the meso-level as the appropriate level for structurally locating the concept of MAS module.

To motivate a choice between such alternatives, we note that, in general, in a human society, one would expect that any long term social action, minimally capable of effectively influencing the structure and operation of the society in which it is realized, or any of its social systems (economic, educational, political, etc.) of that society, would be carried out with most probability of success through an organization (company, school, party, etc.) than by an isolated individual realizing any of the social roles involved in those social systems.

That is, organizations (corporations, associations, institutions, etc.) are the main means through which effective, large scale, and long term, social actions are realized in human societies (in the complex ones, at least), so that such societies effectively operate (in most of its essential processes) as systems structurally composed of “modules” (the organizations) that exist at the meso-organizational level, not of “modules” that could possibly exist either at the population level (the individuals) or at the micro-organizational level (the social roles).

In other terms, from this point of view, an agent society is a system of agent organizations, at least as it is a system of individual agents.

In consequence, we propose that MAS modules should reside at the meso-level of agent societies, and should thus be cast as agent organizations.

4 A Functionalist Approach to Agent Organizations

The type of modularization that we have proposed above has as its base a functionalist conception of agent organizations (about Functionalism in Social Sciences, see, e.g., [10]), which means that agent organizations should be specified in terms of the social functions that they may perform in agent societies.

A clear and operational notion of social functions should then be given. Conceptual and methodological means for such functionalist specifications of agent organizations, should also be provided.

We summarize below, in Sects. 4.1 and 4.2, a sample conceptualization of social functions, which we have elaborated in previous works [11–13, 1].

We let to Sect. 5 the exposition of the conception of functionalist specifications of agent organizations that we are introducing in the present paper.
4.1 The Operational Scheme Underlying Usual Situations of Organizational Interaction

Figure 2 shows an operational scheme underlying usual situations of organizational interaction, namely, organizational interactions that can be construed in terms of the well-known Producer-Consumer scheme\(^1\).

![Diagram of Producer-Consumer Scheme](image)

**Fig. 2.** The Producer–Consumer Scheme.

The scheme characterizes the interaction between two organizational components, one acting as a **Producer** \((P)\), the other acting as a **Consumer** \((C)\). The **Producer** periodically produces some **product** (object or service) and the **Consumer** periodically consumes that **product**, at its convenience and as such **product** is available.

Without loss of generality, we may assume, for the purpose of this paper, that the operation of the Producer-Consumer scheme follows this simple, cyclical interaction protocol:

1. the **Producer** delivers a **product** (operation DeliverProd) to the **Consumer** by storing the **product** in a **storage** (which not explicitly represented in the scheme), after producing it;
2. the **Consumer** consumes the **product** after taking it from the **storage** (operation ReceiveProd);

\(^1\) Another common operational view of organizational interactions is, of course, the Client-Server scheme. We prefer, however, to take the Producer-Consumer scheme as the reference operational scheme for this paper because the Producer-Consumer is more general than the Client-Server scheme: the latter can be construed as a particular case of the former, with the Client acting as the Consumer and the Server acting as the Producer. And, most importantly, the operational processes implied by the Producer-Consumer are bidirectional (in the sense that, in general, either the Producer or the Consumer can independently take the initiative of the interaction), while the operational processes implied by the Client-Server are strictly unidirectional (in the sense that, in general, only the Client can take the initiative of the interaction).
3. after consuming the product, the Consumer frees the storage to the Producer (operation FreeSto), so the latter can store in it the next product;
4. the Producer produces the next product after receiving the storage (operation ReceiveSto);
5. the cycle restarts.

Of course, in an interaction between two organizational components, it is possible that the two modules act, for each other, as both Producer and Consumer, in a two way interaction involving two symmetrically linked Producer-Consumer schemes.

4.2 Organizational Interactions and the Performance of Social Functions

The notion of social function has a long history in Social Sciences, specially Sociology and Anthropology, always being disputed in the operationality of the various definitions that it has received (cf., e.g., [10]).

For our present purposes, we state here the following definition:

A social function performed by an organizational component (a social role, an agent organization, a social system) of an agent society is an activity performed by that organizational component, which satisfies an operational need of another organizational component of the society (or, of the society as a whole).

We note that, implicit in this informal notion of social function, is the idea that a social function is performed in the context of a organizational interaction, that is, an interaction [14]:

- relationally characterized by the existence of a social need, that is, an operational need of a component of the agent society that can only be satisfied through the actions of the other component (consequently, by the reduction of the corresponding dependence relation [15] existent between the two components involved in the interaction);
- and operationally characterized by the performance of persistent, periodic exchanges between those elements (given the usual assumption, that should be taken in general, that social needs have a persistent, periodic character).

From the fact that social functions are driven by social needs (and, so, by social dependence relations [13]) we derive the existence of some operational requirements that are imposed on the behaviors of the involved organizational modules, and on their interaction process:

- an operational requirement on the behavior of the beneficiary of the realization of the social function, characterizing the way its need has to be satisfied;
- and an operational requirement on the interaction process itself, characterizing how the performer of the function should interact with the beneficiary in order for the function to occur (thus, in a way that is independent of any other behavior that the performer may have).
Symbolically, we express the combined operational requirements that specify the performance of a social function $F$ by a performer organizational module $i$ on behalf of a beneficiary organizational module $j$ as $F = (i : OR_{i,j} : j) (j : OR_j)$, where [16]:

- $OR_{i,j}$ is the operational requirement imposed on the interaction process between $i$ and $j$;
- $OR_j$ is the operational requirement imposed on the behavior of the beneficiary of the function $F$.

We note, then, the following:

- only the behavior of the beneficiary that corresponds to the satisfaction of its need is explicitly subject to the operational requirement, for the operational aspects of such need, which drives the performance of the social function, should be precisely expressed by that operational requirement;
- the behavior of the performer of the function is only indirectly subject to the operational requirement, because the details of the behavior proper of the performer are irrelevant, as long as such behavior satisfies the operational requirement imposed on the interaction between the two elements.

![Diagram](image)

**Fig. 3.** The social functions performed under the Producer-Consumer Scheme.

Figure 3 makes explicit the two social functions that are performed in a situation of organizational interaction that can be viewed according to the Producer-Consumer scheme, namely:
\[ PC = (P : \text{DeliverProd}; \text{ReceiveProd} : C) \triangleright (C : \text{Consume}) \]
\[ CP = (C : \text{FreeSto}; \text{ReceiveSto} : P) \triangleright (P : \text{Produce}) \]

That is:

- a function \( PC \) such that the Producer delivers products to the Consumer, so that the Consumer can receive the products and perform its Consume behavior;
- and a function \( CP \) such that the Consumer frees storages to the Producer, so that the Producer can receive the storages and perform its Produce behavior.

The fact that social functions go in pairs, so that the performance of a single organizational interaction, running under a single interaction scheme, simultaneously satisfies two needs (the need of production of the Producer, and the need of consumption of the Consumer, in the case of the Producer-Consumer scheme), and that there is no organizational interaction that can be persistently sustained in a society if it does not satisfy a pair of needs like these, is one of the main conceptual complicators for any attempt to reduce the notion of social function to a unidirectional relation or scheme of interaction.

Additionally, that is the reason why we refrain from the reduction of the notion of social function to the usual notion of service, widely used in Web contexts, and often in MAS contexts (cf., e.g. [17]).

We acknowledge, of course, that in many cases the performance of social functions by organizations can effectively be seen as the performance of services by those organizations (acting as servers), but it should be clear that, in general, organizations are more than servers: organizations are active entities, procuring their interests in the societies where they operate.

### 4.3 Rights and Duties vs Permissions and Obligations

In this section, we remark the difference between the general notions of permissions and obligations, as they are usually taken in the logical-deontic view of agent conducts (which are usually considered to be inter-definable, that is, reducible to one another [18]), and the general notions of rights and duties (which, as remarked by Hofeld [19], are correlative to each other, and so are irreducible to each other, and come in pairs). The notions of functional rights and functional duties, which we adopt in the work, are analyzed in Sect. 4.4.

We start by distinguishing between a behavioral and an interactional view of agent conducts. By behavior we mean a view of an agent conduct where one considers the organized set of the agent’s actions in themselves, irrespectively of their being performed in an interactional context involving other agents. By interaction, we mean a view of an agent conduct where one considers the organized set of the agent’s actions in connection to the organized set of actions of another agent, with which the former agent is in social relationship.

Thus, a typical deontic expression is \( \text{Obl}(i)[\alpha] \), where one sees that only agent \( i \) is being mentioned, together with the action \( \alpha \) that it is assumingly obligated to perform. Clearly, no other agent is mentioned in that deontic expression, besides
agent \(i\), meaning that no interactional context is relevant both for the understanding of the expression, and for the verification of the agent’s compliance with the obligation.

In the conception of rights and duties, on the other hand, one considers agents in interaction, possibly performing social functions for each other, as construed e.g. by the Producer-Consumer scheme that we analysed in Sect. 4.4.

Thus, an expression for the right and the duty attached to the performance of an organizational interaction by organizational components \(i\) and \(j\) should involve both components and, in the simplest case, an action to which the right and duty refers. A possible expression for the simplest case would be, e.g., \(RD(i, j)[\alpha]\), meaning that the performance of the action \(\alpha\) is simultaneously a right of agent \(i\) and a duty of agent \(j\).

However, as we will remark presently, an organizational interaction involves not only one action (or, object), but two actions/objects to be exchanged in the interaction.

### 4.4 The Functional Rights and Duties Implied in a Situation of Organizational Interaction

In fact, as seen in Sect. 4.2, in a general situation of organizational interaction, the action/object of the functional duty is not necessarily the action/object of the functional right: in general, they are two different actions/objects, but such that a relation of enablement holds between the former and the latter.

That is, if \(\alpha\) is the action/object of the functional duty and \(\beta\) is the action/object of the functional right, the general functional situation is such that the production of \(\alpha\) (or, the liberation of access to it) leads to the enabling of the consumption (or, access) to \(\beta\).

We formally express the enablement relation by \(\alpha \Rightarrow \beta\), so that the formal expression of the general functional situation in terms of functional rights and duties is given by \(RD(i, j)[\alpha \Rightarrow \beta]\).

In such general case, we say that the enablement of \(\beta\) through the production of \(\alpha\) is a functional right of the organizational component \(i\), while the production of \(\alpha\) so that \(\beta\) gets enabled is a functional duty of \(j\).

Also, we often express the general functional situation of rights and duties by \(RD(i, j)[\alpha; \beta]\)

The way functional rights and functional duties are present in the general Producer-Consumer scheme is better seen by analysing the scheme into two separate sub-schemes, each showing the performance of a social function (cf. the analysis of Fig. 3, above).

The functional rights and functional duties implied in each of the two social functions, \(PC\) and \(CP\), are respectively the following:

(a) \(RD(Consumer, Producer)[DeliverProd; ReceiveProd]\)
(b) \(RD(Producer, Consumer)[FreeSto; ReceiveSto]\)

meaning that:
(a) with respect to the social function \( PC \), it is a functional duty of the \textit{Producer} to \textit{DeliverProd} to the \textit{Consumer}, in order to enable \textit{ReceiveProd}, and it is a functional right of the \textit{Consumer} to \textit{ReceiveProd}, as it is enabled by \textit{DeliverProd};

(b) with respect to the social function \( CP \), it is a functional duty of the \textit{Consumer} to \textit{FreeSto} to the \textit{Producer}, in order to enable \textit{ReceiveSto}, and it is a functional right of the \textit{Producer} to \textit{ReceiveSto}, as it is enabled by \textit{FreeSto}.

It is clear from the set of functional rights and duties implied by the \textit{Producer-Consumer} scheme that:

– the compliance with those functional duties and the assurance of those functional rights are necessary for the appropriate operation of the scheme;

– the functional rights implied in the interaction scheme concern the actions performed in the interaction between the organizational components involved in the scheme (e.g., the \textit{Deliver} \ldots and the \textit{Receive} \ldots operations), not the operations that are private to the organizational components (e.g., \textit{Produce} for the \textit{Producer}, \textit{Consume} for the \textit{Consumer});

– and that any deviation from the operational scheme of the interaction, by any of the organizational components involved in it, imply a deviation from a functional duty and the consequent break of a functional right.

4.5 Functional Commitments in Organizational Interactions

In the same way that there are certain normative issues (rights and duties) implied in any proper performance of the \textit{Producer-Consumer} scheme, there are certain functional commitments implied in it. In particular, there are the commitments to the abiding to the normative issues (i.e., to the accomplishments of the duties and to the respect for the rights).

There is a long tradition concerning the use of the notion of commitment in the multiagent systems area, which we can not summarize here (cf. [20] and [21]. We, thus, base the following in Castelfranchi’s summary of the main issues concerning commitments, as presented in [20], specially in the notions of internal commitment, social commitment and collective commitment.

Thus, we say that for the proper operation of the \textit{Producer-Consumer} scheme, the elements implementing the scheme should be internally committed to the performance of the actions involved in the scheme, because that is the way they both socially and collectively commit to the social functions and to the functional rights and duties implied by the scheme.

Accordingly, the proper performance of the \textit{Producer-Consumer} scheme requires the following internal commitments:

– from the \textit{Producer}, the commitments to perform the \textit{DeliverProd} and the \textit{ReceiveSto} operations;

– from the \textit{Consumer}, the commitments to perform the \textit{ReceiveProd} and the \textit{FreeSto} operations,
which together imply that:

– the **Producer** socially commits to the **Consumer** both to comply with the **Producer**’s duty to *DeliverProd* and to respect the **Consumer**’s right to *ReceiveProd*;

– the **Consumer** socially commits to the **Producer** both to comply with the **Consumer**’s duty to *FreeSto* and to respect the **Producer**’s right to *ReceiveSto*.

Using a notation based on that in [20], where:

– icomm(*i*, *α*) means that *i* is *internally committed* to perform action (or, goal) *α*

– scomm(*i*, *j*, *α*) means that *i* is *socially committed* to *j* too perform action (or, goal) *α*

– ccomm(*I*, *α*) means that the set of agents *I* is *collectively committed* to perform the action (or, goal) *α*

we may say that the proper operation of the **Producer-Consumer** scheme is guaranteed only if:

\[
icomm(Producer, DeliverProd) \land \icomm(Producer, ReceiveSto) \land 
\icomm(Consumer, ReceiveProd) \land \icomm(Consumer, FreeSto)
\]

for only then the following are guaranteed:

\[
scomm(Producer, Consumer, PC)
scomm(Consumer, Producer, CP)
scomm(Producer, Consumer
\[
\text{RD}(Consumer, Producer)[DeliverProd; ReceiveProd])
scomm(Producer, Consumer
\[
\text{RD}(Producer, Consumer)[FreeSto; ReceiveSto])
\]
\[
ccomm\{Producer, Consumer\}, PC
\]
\[
ccomm\{Producer, Consumer\}, CP
\]
\[
ccomm\{Producer, Consumer\}
\[
\text{RD}(Consumer, Producer)[DeliverProd; ReceiveProd])
ccomm\{Producer, Consumer\}
\[
\text{RD}(Producer, Consumer)[FreeSto; ReceiveSto])
\]

where PC and CP are the social functions realized by the operational structure of the **Producer-Consumer** scheme.

### 5 Functionalist Specifications of Agent Organizations

We define a *functionalist specification* of an agent organization *Org* as a structure \(FS_{org'} = (RD, F)\) where:

– RD is a set of *functional rights and duties*, each given as
\[ RD(Org, Org')[\alpha; \beta] \]

or as
\[ RD(Org', Org)[\alpha; \beta] \]

– F is a set of social functions, each given as
\[ (Org : OR_{Org,Org'} : Org') \triangleright (Org' : OR_{Org}) \]

or as
\[ (Org' : OR_{Org',Org} : Org) \triangleright (Org : OR_{Org}) \]

where \( Org' \) stands for any other agent organization that can organizationally link to \( Org \) in a proper way (that is, in a way that satisfies the operational requirements in the specification).

A functionalist specification is such that whenever an organization \( i \) (either \( Org \) or \( Org' \)) is organizationally linked in a proper way to another agent organization \( j \) (resp., \( Org' \) or \( Org \)):

– the interaction between the agent organization \( i \) and the agent organization \( j \) is such that the persistent performance of the sequence of actions \( \alpha; \beta \) specified in one of the expressions of functional rights and duties \( RD(i, j)[\alpha; \beta] \) satisfies the operational requirement \( OR_{i,j} \) of one of the social functions \( F \) of the set of social functions \( F \);

– the agent organization that takes the place of \( j \) in an expression of functional rights and duties \( RD(i, j)[\alpha; \beta] \) behaves in such way that it persistently performs a behavior \( b \) that satisfies the operational requirement \( OR_{j} \), with the repeated performance of \( b \) occurring as a consequence of the persistent performance of the sequence of actions \( \alpha; \beta \) in the interaction of \( i \) with \( j \).

5.1 Template Functionalist Specification Based on the Producer-Consumer Scheme

Considering organizational links operating according to the Producer-Consumer interaction scheme, one can determine a (semi-formal) template functionalist specification, serving the set of functional rights and duties and the social functions pertaining to the scheme, showing how to specify the way an agent organization acting as a Producer \( (P) \) links to an agent organization acting as a Consumer \( (C) \) (or vice-versa, cf. Fig. 3).

Such template functionalist specification is as follows:

\[ FS_{PC} = (RD, F) \]

where:

- **RD** = \{ \( RD_1, RD_2 \) \}
  - is a set of functional rights and duties, with:
  \[ RD_1 = RD(P,C)[\alpha; \beta] \text{ and } RD_2 = RD(C,P)[\alpha'; \beta'] \]
  for some actions \( \alpha, \beta, \alpha', \beta' \);
- **F** = \{ \( F_1, F_2 \) \}
is a set of \textit{social functions}, with:

\[ F_1 = (P : OR_{P,C} : C) \triangleright (C : OR_C) \text{ and} \]
\[ F_2 = (C : OR_{C,P} : P) \triangleright (P : OR_P) \]

for some operational requirements \( OR_{P,C}, OR_C, OR_{C,P}, OR_P \) appropriately involving the actions \( \alpha, \beta, \alpha', \beta' \).

5.2 The Normative Environment Presupposed by a Functionalist Specification

For any functionalist specification of an agent organization, there is a set of \textit{operational schemes} that underlie the set of organizational interactions functionally specified in that specification (e.g., the \textit{Producer-Consumer} scheme, in the above analyses).

Such operational schemes, however, are not part of the functionalist specification, and possibly are not uniquely determined.

This means that, as long as the interactions between the specified agent organization and the other organizations to which it is linked are performed in ways that respect the specified set of functional rights and duties, and that fulfill the specified social functions, it is essentially irrelevant that the interactions be performed according to any precisely defined operational scheme.

However, in view of the openness of the agent society (and in consequence, of the possible openness of the agent organizations that comprise it) some \textit{regulatory means} should be available, to guarantee that the functionalist specification will be respected during the operation of the agent organization, irrespectively of which agents enter and leave the specified agent organization or its partner organizations (and of which organizational links are created or removed between that organization and its partners), and as long as a new functionalist specification is not imposed on that organization.

That regulatory means is needed because the establishment of an organizational link between an agent organization \( \text{Org} \) and another organization \( \text{Org}' \), according to a\textit{functionalist specification} \( FS \), is meant to \textit{bind} both \( \text{Org} \) and \( \text{Org}' \) to the set of \textit{functional rights} and \textit{functional duties} (and commit them to the performance of the social functions) specified by \( FS \), and that binding should be enforced, in view of those possible situations.

We express the need for such \textit{regulatory means} as a requirement on the structure of the agent society, namely, as a requirement for a \textit{normative environment} able to provide the required regulation, where by a \textit{normative environment} we understand a set of normative systems operating in (and on) the society.

In other words, the problem of integrating agent organizations into agent societies is, from the functionalist point of view, a problem of \textit{normativity}.

In the case of conventional software systems, on the other hand, given the relatively fixed nature of those systems, it seems more appropriate to \textit{fix} the functioning of the integrated agent organizations, that is, to restrict their functioning so that the set of agents participating in the agent organization do not change (or else, change but on the condition that the interactions of those agent
organizations with the rest of the system do not change), and so that the creation and deletion of organizational links occur in very well-known forms.

This way, the problem of integrating agent organizations to conventional software systems can be reduced to a problem of a different nature, namely, to the conventional problem of the verification of software modules.

6 Integrating MAS Modules into Agent Societies and Conventional Software Systems

In this section we seek to explore a few complementary principles, necessary for the integration of modular agent organizations into agent societies and conventional software systems.

In Software Engineering for conventional software systems, the term *functional requirements* usually denotes “a description of a behavior that a system will exhibit under specific conditions”, while the term “non-functional requirement” is usually used to denote “a description of a property or characteristic that a system must exhibit or a constraint that it must respect” (cf. [22]).

That is, the term *function* is used to refer essentially to a behavioral, observational description of the system being specified, which abstracts away the internal functioning of that system.

This way, then, the term *function*, in that context, concerns mainly the behaviors of the system being specified, not the *interaction processes* as such, which that system performs together with other systems.

On the other hand, as explained in Sect. 3.2, the term *function*, in the functionalist sense that we are using it here, implies exactly the opposite: it is relational, meaning that it concerns an interaction between organizational components, and it is directly concerned with the *internal operational requirement* of the organizational component whose operation constitutes the need that the performance of the function will satisfy.

This difference between those two uses of the term *function* has as a consequence the difference between the senses of the terms *functional specification*, as it is used in conventional Software Engineering, and *functionalist specification*, as it is used here.

The possibility of the seamless integration of a given agent organization into either an agent society or a conventional software system depends, then, on the availability of *functionalist specifications* from which one could derive verifiable *functional specifications*.

6.1 MAS Modules in Agent Societies

We call *MAS module* any structure $MASMod = (Org, FS_{PC})$, where:

- $Org$ is an agent organization;
- $FS_{PC} = (RD, F)$ is a functionalist specification of $Org$ such that whenever $Org$ links to another organization $Org'$, either $Org$ or $Org'$ may take the
places of $P$ of $C$ and, by doing so, become bound to the functional rights and duties specified in $RD$, and committed to the performance of the social functions specified in $F$.

As indicated in Sect. 5.2, the integration of a MAS module into an agent society supposes the existence in that society of a normative environment capable of dynamically regulating such integration, with the aim of preserving the organizational links established by that MAS module (and the functional rights and duties that they imply), given the possibility of the agents freely entering and leaving any modular organization, and the society as a whole.

6.2 Modular Agent Organizations in Conventional Software Systems

As discussed in Sect. 5.2, since there is no normative environment in conventional software systems, MAS modules should be restricted in the flexibility with which they are integrated to such systems.

That is, the general problem of safely integrating a variable modular agent organization to a conventional software system should become the problem of obtaining a functional specification from the functionalist specification of the agent organization, so that the software developer could verify that the agent organization operates in accordance the functional specification, even if it occurs both of the entering and leaving of the agents in the agent organization, and the changing of agent behaviors and interactions due to learning or other motive.

Clearly, the operational schemes that underlie the functionalist specifications are the simplest candidates for founding such functional specifications, the problem remaining open as to the means to obtain those operational schemes from the functionalist specifications (in systematic way, if possible).

However, more favorably for the system developer would be, of course, situations where s/he can be told and guaranteed that the agents of the MAS module don’t go through learning processes, and do not enter or leave the MAS module.

7 Related Works

In the literature concerned with the issue of modularity and multiagent systems, we could find the following types of related works, besides the works already mentioned above.

First, works regarding modularity in general. We take Parnas’ classical paper [23] as a reference. The main idea in the paper is that each module should hide some “design decision” from the rest of the system, so that future modifications in that decision will not have impacts outside the module. The two main examples of design decision given in that paper are: (i) decisions about the performance of the major steps in the system’s functioning; and (ii) decisions about the way information will be structured and stored in the system.

In the case of agent societies and the notion of MAS module proposed here, the most immediate interpretation of that idea is that MAS modules should
hide decisions about the performance of social functions. That is, in the design of agent societies, a first step should be the listing of the social functions that the society will perform, then assigning those functions to MAS modules (organizations) and systems of MAS modules (social systems), and finally designing the required MAS modules and the way they should be organizationally linked.

The second type of works we were able to find in the literature are those concerned with the introduction of the notion of modularity in agent programming languages. In this regard, we could find two types of works: (i) works concerned with the internal modularization of agents, e.g., [24–26]; works concerned with the introduction of modularization of organizations, e.g., [27].

In the first case, modularity is introduced at the intra-agent level, in the second case, at the intra-organization level. The paper [28], on the other hand, which is related to the modular formal modeling of multiagent systems, introduces modularity at an inter-agent level, but thus not refer to the notion of organization, thus keeping itself at the intra-organizational level.

The third kind of works are concerned with general inter-organizational issues. One work we could find is [29]. However, that paper is not concerned with modularity issues.

The fourth kind of works deal with the notion of contract in MAS, as a means of specification of interaction mechanisms, like [30] and even more specifically [31]. Although such works are concerned with the intra-organizational level, they are directly relevant to what we proposed here, presenting mechanisms that apparently can readily be lifted to the inter-organizational level, and that can help to support the flexibility of the inter-organizational interaction structure of agent societies.

Finally, works concerned with methodological issues. For instance, [32–34] that, again, focus on the modularization of the intra-organizational level.

However, the ideas in [35], concerning reusable organizations, are really close to the ideas introduced in the present paper. Organizations are treated as components for multiagent systems, based on the notion that organizations perform services, and services can be composed.

The main differences to what we introduced here seem to be: (i) that the overall structure generated by the composition of organization components are agent organizations, not agent societies; and (ii) that those organizations perform services, not social functions. As a consequence, the concern with the functional and normative issues accompanying the organizational linking of organizations is, in principle at least, absent from that work.

8 Conclusion: The Way Ahead

This paper results from a reflection about the organizational modularity of human societies, and the idea that human societies are systems of organizations at least as they are systems of individuals, differently from what is often held by the methodological individualism (and its accompanying emphasis on action theory
and rational choice theory), which seem to be largely endorsed in the MAS area, at present.

Also, the work presented here may be seen as a proposal for going one structural level up in the research on multiagent systems organization, from the intra-organizational concerns that have captured the attention of the researchers in the area since the beginning of its concern with organizations, to inter-organizational concerns, thus allowing for both the differentiation between agent organizations and agent societies, and for the possible establishment of a notion of MAS module, like the one that we have proposed.

Several issues remain to be worked out, besides the need for a formalized presentation capable of fine tuning the concepts that we have introduced. In particular: (i) the issue of situatedness of the agent societies; (ii) the characterization of the normative environment required by the organizational modularization of agent societies; and (iii) a (hopefully systematic) way of deriving functional specifications from functionalist specifications, to support the modular integration of MAS modules into conventional software systems.

Finally, one should mention some methodological issues. It seems important to try to take component-based organization-oriented software engineering approaches that are focused on the intra-organizational level (like the one introduced in [33]) to the inter-organizational level (as proposed in the present paper).

That will surely require a specific meta-methodological research effort, and perhaps a Method Engineering approach (in the vein of that described in [34]) can be useful in this respect. The ideas introduced here aim to indicate one possible direction for that effort.

Acknowledgments

The author is very grateful to the paper referees for their valuable comments, and specially for the references on related work that they suggested. This work was partially supported by FAPERGS and CNPq.

References


38


A Stepwise Refinement based Development of Self-Organizing Multi-Agent Systems: Application to the Foraging Ants

Zeineb Graja\textsuperscript{1,2}, Frédéric Migeon\textsuperscript{2}, Christine Maurel\textsuperscript{2}, Marie-Pierre Gleizes\textsuperscript{2}, Amira Regayeg\textsuperscript{1}, and Ahmed Hadj Kacem\textsuperscript{1}

\textsuperscript{1} Research on Development and Control of Distributed Applications laboratory (ReDCAD)  
Faculty of Economics and Management  
University of Sfax, Tunisia  
zeineb.graja@redcad.org, \{amira.regayeg, ahmed.hadjkacem\}@fsegs.rnu.tn

\textsuperscript{2} Institute for Research in Computer Science in Toulouse (IRIT)  
Paul Sabatier University  
Toulouse, France  
zeineb.graja, frederic.migeon, christine.maurel, marie-pierre.gleizes\}@irit.fr

Abstract. In this paper we propose a formal modeling for self-organizing Multi-Agent Systems (MAS) based on stepwise refinements, with the Event-B language. This modeling allows to develop this kind of systems in a more structured manner. In addition, it enables to reason, in a rigorous way, about the correctness of the derived models both at the individual level and the global level. Our work is illustrated by the foraging ants case study.

Keywords: Self-organizing MAS, foraging ants, formal verification, refinement, Event-B

1 Introduction

Self-Organizing Multi-Agent Systems (SO-MAS) are made of multiple autonomous entities (called agents) interacting together and situated in an environment. Each agent has a limited knowledge about the environment and possesses its own goals. The global function of the overall system emerges from the interactions between the individual entities composing the system as well as interactions between the entities and the environment. Thanks to their self-organizing mechanisms, SO-MAS are able to adjust their behavior and cope with the environment changes [1]. When designing this kind of systems, two levels of observation are generally distinguished: the micro-level which corresponds to the agents local behavior and the macro-level which describes the emergent global behavior.

One of the main challenges when engineering a SO-MAS is about giving assurances and guarantees related to its correctness, robustness and resilience. Correctness refers to fulfillment of the different constraints related to the agents
activities. Robustness ensures that the system is able to cope with changes and perturbations [2]. Whereas resilience informs about the capability of the system to adapt when robustness fails or a better performance is possible [3].

In order to promote the acceptance of SO-MAS, it is essential to have effective tools and methods to give such assurances. Some works propose using test and simulation techniques [4], others define metrics for evaluating the resulting behavior of the system [5]. Our proposal to deal with SO-MAS verification is to take advantage of formal methods. We propose a formal modeling for the local behavior of the agents based on stepwise refinement steps and the Event-B formalism [6]. Our refinement strategy guarantees the correctness of the system. In order to prove the desired global properties related to robustness and resilience, we make use of Lamport’s Temporal Logic of Actions (TLA) and its fairness-based proof rules. The use of TLA was recently proposed in [7] in the context of population protocols to prove liveness and convergence properties and fits well with SO-MAS. Our work is illustrated with the foraging ants case study.

This paper is organized as follows. Section 2 presents a background related to the Event-B language, the main principles on which it is based and TLA. In section 3, our refinement strategy of SO-MAS is presented. An illustration of this strategy on the foraging ants is given in section 4. Section 5 presents a summary of related works dealing with verification of SO-MAS. Section 6 concludes the paper and draws future perspectives.

2 Background

2.1 Event-B

The Event-B formalism was proposed by J.R. Abrial [6] as an evolution of the B language. It allows a correct-by-construction development for distributed and reactive systems. Event-B uses set theory as a modeling notation which enables, contrary to process algebra approaches, to support scalable solutions for system modeling. In order to make formal verification, Event-B is based on theorem proving. This technique avoids the problem of explosion in the number of the system states encountered with the model checkers.

The concept used to make a formal development is that of a model. A model is formed of components which can be of two types: machine and context. A context is the static part of the model and may include sets and constants defined by the user with their corresponding axioms. A machine is the dynamic part of the model and allows to describe the behavior of the designed system. It is composed by a collection of variables \( v \) and a set of events \( ev_i \). The variables are constrained by conditions called invariants. The execution of the events must preserve these invariants. A machine may see one or more contexts, this will allow it to use all the elements defined in the seen context(s). The structures of a machine and an event in Event-B are described as follows.
An event is defined by a set of parameters \( p \), the guard which gives the necessary conditions for the activation of the event \( G_{evi}(p, v) \) and the action \( A_{evi}(p, v, v') \) which describes how variables \( v \) are substituted in terms of their old values and the parameters values. The action may consist in several assignments which can be either deterministic or non-deterministic. A deterministic assignment, having the form \( x := E(p, v) \), replaces values of variables \( x \) with the result obtained from the expression \( E(p, v) \). A non-deterministic assignment can be of two forms: 1) \( x \in E(p, v) \) which arbitrarily chooses a value from the set \( E(p, v) \) to assign to \( x \) and 2) \( x : | Q(p, v, x') \) which arbitrarily chooses to assign to \( x \) a value that satisfies the predicate \( Q \). \( Q \) is called a before-after predicate and expresses a relation between the previous values \( v \) (before the event execution) and the new ones \( v' \) (after the event execution).

**Proof obligations.** Proof Obligations (POs) are associated with Event-B machines in order to prove that they satisfy certain properties. As an example, we mention the Preservation Invariant \( INV \) which is necessary to prove that invariants hold after the execution of each event.

**Refinement.** This technique, allowing a correct by construction design, consists in adding details gradually while preserving the original properties of the system.

The refinement relates two machines, an abstract machine and a concrete one.

The refinement of an abstract event is performed by strengthening its guard and reducing non-determinism in its action. The abstract parameters can also be refined. In this case, we need to use witnesses describing the relation between the abstract and the concrete parameters. The correctness of the refinement is guaranteed essentially by discharging POs \( GRD \) and \( SIM \). \( GRD \) states that the concrete guard is stronger than the abstract one. \( SIM \) states that the abstract event can simulate the concrete one and preserves the gluing invariant. An abstract event can be refined by more than one event. In this case, we say that the concrete event is split. In the refinement process, new events can be introduced. In order to preserve the correctness of the model, we must prove that these new introduced events do not take the control for ever; i.e. they will terminate at a certain point or are convergent. This is ensured by the means of a variant—a numerical expression or a finite set—that should be decreased by each execution of the convergent events.

**B-event** is supported by the Rodin platform\(^3\) which provides considerable assis-

---

\(^3\) http://www.event-b.org/
tance to developers by automating the generation and verification of all necessary POs.

2.2 Temporal Logic of Actions (TLA)

TLA combines temporal logic and logic of actions for specifying and reasoning about concurrent and reactive discrete systems [8]. Its syntax is based on four elements: 1) constants, and constant formulas - functions and predicates - over these, 2) state formulas for reasoning about states, expressed over variables as well as constants, 3) transition or action formulas for reasoning about (before-after) pairs of states, and 4) temporal predicates for reasoning about traces of states; these are constructed from the other elements and certain temporal operators [7]. In the remainder of this section, we give some concepts that will be used further in section 4.

**Stuttering step.** A stuttering step on an action $A$ under the vector variables $f$ occurs when either the action $A$ occurs or the variables in $f$ are unchanged. We define the stuttering operator $\langle A \rangle_f$ as: $\langle A \rangle_f \equiv A \lor (f' = f)$. Dually, $\langle A \rangle_f$ asserts that $A$ occurs and at least one variable in $f$ changes.

$\langle A \rangle_f \equiv A \land (f' \neq f)$.

**Fairness.** Fairness asserts that if a certain action is enabled, then it will eventually be executed. Two types of fairness can be distinguished: 1) Weak Fairness for action $A$ denoted $WF_f(A)$; which asserts that an operation must be executed if it remains possible to do so for a long enough time and 2) Strong Fairness for action $A$ denoted $SF_f(A)$; asserts that an operation must be executed if it is often enough possible to do so [8]. Formally $WF_f(A)$ and $SF_f(A)$ are defined as follows.

$WF_f(A) \equiv \Box \Diamond Enabled(A)_f \Rightarrow \Box \Diamond \langle A \rangle_f$

$SF_f(A) \equiv \Box \Diamond Enabled(A)_f \Rightarrow \Box \Diamond \langle A \rangle_f$

$\Box$ and $\Diamond$ are temporal operators. $\Box P$ called always $P$ means that $P$ is always true in a given sequence of states. $\Diamond P$ called eventually $P$ means that $P$ will hold in some state in the future.

$Enabled(A)_f$ asserts that it is possible to execute the action $\langle A \rangle_f$. In addition, we define the leads to operator: $P \leadsto Q \equiv \Box(P \Rightarrow \Diamond Q)$, meaning that whenever $P$ is true, $Q$ will eventually become true.

**Proof rules for simple TLA.** We consider the two proof rules $WF1$ and $SF2$ given below. $WF1$ gives the conditions under which weak fairness assumption of action $A$ is sufficient to prove $P \leadsto Q$. Condition $WF1.1$ describes a progress step where either state $P$ or $Q$ can be produced. Condition $WF1.2$ describes the inductive step where $\langle A \rangle_f$ produces state $Q$. Condition $WF1.3$ ensures that $\langle A \rangle_f$ is always enabled. $SF1$ gives the necessary conditions to prove $P \leadsto Q$ under strong fairness assumption. The two first conditions are similar to $WF1$. The third condition ensures that $\langle A \rangle_f$ is eventually, rather than always, enabled.
Formal modeling of the agents local behavior. The main concern at this level is the design of the behavior of the agents and their interactions. In a very abstract way, the behavior of each agent is composed by three steps: the agent senses information from the environment (perception step), makes a decision according to these perceptions (decision step) and finally performs the chosen action (action step). We refer to these steps as the perceive – decide – act cycle. Thus, an agent is characterised by the representations of the environment that it possesses (\(\text{rep}\)), a set of decision rules telling it which decisions to make (\(\text{decisions}\)), the set of actions it can perform (\(\text{actions}\)) and the set of operations (\(\text{perceptions}\)) allowing it to update its representations of the environment. Moreover, an agent is identified by its intrinsic characteristics such as the representations it has on itself (\(\text{prop}\)), sensors (\(\text{sensors}\)) and actuators (\(\text{actuators}\)). More formally, an agent is described by the tuple:

\[
\text{agent} \equiv < \text{prop}, \text{rep}, \text{sensors}, \text{actuators}, \text{decisions}, \text{actions}, \text{perceptions} >
\]

In Event-B, the characteristics of agents, their representations of the environment, sensors and actuators are modelled by means of variables. Whereas their decisions, actions and update operations are formalised by events. Hence, a before-after predicate can be associated with each one of them. As a consequence, the decisions of each agent \(ag\), belonging to the set of agents noted \(\text{Agents}\), can be considered as a set of before-after-predicates denoted \(\text{Decide}_{i}(ag, d, d')\), where \(d\) is the set of variables corresponding to the properties and actuators of \(ag\). Moreover, the actions of each agent \(ag\) can be considered as a set of before-after predicates having the form \(\text{Act}_{j}(ag, a, a')\), where \(a\) is the set of variables corresponding to the properties and sensors of \(ag\). Indeed, an action event is responsible for getting the agent to the perception step. Since the actions of an agent can affect its local environment, the set \(a\) can also contain variables describing the environment state. Finally, \(\text{perceptions}\) is the event enabling an
agent to update its perceptions. It is described by the before-after predicate: $\text{Perceive}(ag, rep, rep')$. The local agents behavior described earlier is said “correct”, if the following properties are satisfied.

- **LocProp1**: The behavior of each agent is complied with the perceive-decide-act cycle.
- **LocProp2**: The agent must not be deadlocked in the decision step, i.e. the made decision must enable the agent to perform an action.

$$\text{LocProp2} \equiv \forall ag \bullet ag \in \text{Agents} \land \text{Decide}_i(ag, d, d') = TRUE \Rightarrow \exists \text{Act}_i \bullet \text{Act}_i \in \text{actions} \land G\text{.Act}_i(ag, a) = TRUE$$

- **LocProp3**: The agent must not be deadlocked in the perception step; i.e. the updated representations should allow it to make a decision.

$$\text{LocProp3} \equiv \forall ag \bullet ag \in \text{Agents} \land \text{Perceive}_i(ag, rep, rep') = TRUE \Rightarrow \exists \text{Decide}_i \bullet \text{Decide}_i \in \text{decisions} \land G\text{.Decide}_i(ag, d) = TRUE$$

**Global properties of the macro-level.** At the macro level, the main concern is to prove that the agents behavior, designed at the micro-level, will lead to the desired global properties. The aim is to discover, in the case of proof failure, design errors and thus make the necessary corrections at the micro-level. One of the most relevant global properties that should be proved, when designing self-organizing systems, is robustness. Serugendo ([2]) defines four attributes for the analysis of robustness: 1) Convergence: indicates the system ability to reach its goal; 2) Stability: informs about the system capacity to maintain its goal once reached; 3) Speed of convergence and 4) Scalability: shows if the system is affected by the number of agents.

Besides robustness, resilience represents another relevant property that should be analysed for SO-MAS. Resilience refers to the ability of the system to self-adapt when facing changes and perturbations. The analysis of resilience allows assessment of the aptitude of self-organizing mechanisms to recover from errors without explicitly detecting an error ([2],[3]).

In this paper, we only focus on proving the stability property. We give an example from the foraging ants case study and some guidelines to prove it in the next section. The formalization and proof of the remaining properties is still an ongoing work.

**Refinement Strategy.** The formal development of SO-MAS begins by a very abstract model representing the system as a set of agents operating according to the Perceive-Decide-Act cycle. This abstract model guarantees LocProp1. The first refinement consists in identifying the different actions performed by the agents. This refinement should ensure LocProp2. In the next step, we specify the events corresponding to the decisions that an agent can make. In addition, we describe the rules allowing the agent to decide. In the third refinement, the perceptions of the agents and the necessary events to update them are identified. As a consequence the different events related to the decisions and actions are

---

4 convergence here is different from the convergence of an event in Event-B, i.e. termination.
refined and property \textit{LocProp3} should be satisfied. 
At this level of refinement, the robustness properties can be checked. Further refinements steps are necessary to prove properties related to resilience. In particular, we need to refine the agents local behavior by specifying the self-organisation mechanisms. At this level of refinement, the global properties related to resilience should be proved. In the last refinement step, we identify the different perturbations coming from the environment. This refinement should preserve the global resilience properties.

4 Application to the Foraging Ants

The case study is a formalization of the behavior of a foraging ants colony. The system is composed of several ants moving and searching for food in an environment. Their main goal is to bring all the food placed in the environment to their nest. Ants do not have any information about the locations of the sources of food, but they are able to smell the food which is inside their perception field. The ants interact with one another via the environment by dropping a chemical substance called 	extit{pheromone}. In fact, when an ant discovers a source of food, it takes a part of it and comes back to the nest by depositing pheromone for marking food paths. The perturbations coming from the environment are mainly pheromone evaporation and appearance of obstacles. The behavior of the system at the micro-level is described as follows. Initially, all ants are in the nest. When exploring the environment, the ant updates its representations in its perception field and decides to which location to move. When moving, the ant must avoid obstacles. According to its smells, three cases are possible: 1) the ant smells food: it decides to take the direction for which the smell of food is the strongest; 2) the ant smells only pheromone: it decides to move towards the direction in which the smell of pheromone is stronger; 3) the ant does not smell anything: it chooses its next location randomly. When an ant reaches a source of food on a location, it collects it and comes back to the nest. If some food remains in this location, the ant drops pheromone when coming back. Arriving at the nest, the ant deposits the harvested food and begins another exploration. In addition to the properties \textit{LocProp1}, \textit{LocProp2} and \textit{LocProp3} (described in section 3), the following properties should be verified at the micro-level. \textit{LocInv1}: the ant should avoid obstacles and \textit{LocInv2}: a given location cannot contain both obstacle and food. The main global properties associated with the foraging ants system are described in the following:\footnote{C refers to Convergence, S to Stability and R to Resilience}

\begin{itemize}
  \item C1: The ants are able to reach any source of food,
  \item C2: The ants are able to bring all the food to the nest,
  \item S1: When a source of food is detected, the ants are able to focus on its exploitation and
  \item R1: The ants focusing on exploiting a source of food, are able to continue their foraging activity when this source of food suddenly disappear from the environment.
\end{itemize}

In the remainder of this section, we only focus on the properties related to the correctness (\textit{LocProp1}, \textit{LocProp2}, \textit{LocProp3}, \textit{LocInv1} and \textit{LocInv2}) and the
stability ($S1$) of the system. The proofs of convergence and resilience are still an ongoing work. The next section illustrates the proposed refinement strategy.

4.1 Formalization of the ants local behavior

**Abstract model:** the initial machine $Ants0$ describes an agent (each agent is an ant) operating according to the *Perceive-Decide-Act* cycle. It contains three events *Perceive*, *Decide* and *Act* describing the agent behavioral rules in each step. At this very abstract level, these events are just responsible for switching an agent from one step to another. The current cycle step of each agent is depicted by the variable $stepAgent$ defined as follows.

$$inv1 : stepAgent \in Ants \rightarrow Steps$$

where $Ants$ defines the set of the agents and $Steps$ is defined as follows:

$$axm1 : partition(Steps, \{perceive\}, \{decide\}, \{act\})$$

As an example, we give below the event *Act* modeling the action step. The only action specified at this level is to switch the ant to the perception step.

**EVENT Act**

ANY

ant

WHERE

$qrd12 : ant \in Ants \land stepAgent(ant) = act$

THEN

act1 : stepAgent(ant) := perceive

END

The proof obligations related to this machine concern essentially preservation of the invariant $inv1$ by the three events. All of them are generated and proved automatically under the Rodin platform.

**First refinement:** in the first refinement $Ants1$, we add the variables *QuFood*, *Obstacles* modeling respectively the food and the obstacles distribution in the environment, *currentLoc* and *load* which give respectively the current location and the quantity of food loaded of each ant. Invariants $inv5$ and $inv6$ guarantee the properties $LocInv1$ and $LocInv2$ respectively.

Moreover, the *Act* event is refined by the four following events: 1) *Act_Mov*: the ant moves in the environment, 2) *Act_Mov_Drop_Phero*: the ant moves and drops pheromone when coming back to the nest, 3) *Act_Harv_Food*: the ant picks up food and 4) *Act_Drop_Food*: the ant drops of food at the nest. In the following, the event *Act_Mov* is presented as an action event example.
The parameter \( \text{loc} \) is the next location to which the ant will move. It is the result of the decision process. This decision process will be modeled in the next refinement. The parameter \( \text{decideAct} \) is also an abstract parameter that will be refined in the next step. It indicates what type of decision can lead to the execution of the \( \text{Act Mov} \) event.

The majority of the generated POs are related to proving the refinement correctness (the SIM PO) and the preservation of invariants. With the presented version of the \( \text{Act Mov} \) event, it is impossible to discharge the \( \text{inv5} \) preservation PO (\( \text{inv5} \) states that an ant cannot be in a location containing obstacles). In fact, if \( \text{loc} \) belongs to the set \( \text{Obstacles} \), \( \text{Act Mov} \) will enable \( \text{ant} \) to move to a location containing an obstacle, which is forbidden by \( \text{inv5} \). In order to discharge the \( \text{inv5} \) preservation PO, we need to add the guard \( \text{grd5} : \text{loc} \notin \text{Obstacles} \) to \( \text{Act Mov} \) event. Finally, in order to guarantee the property \( \text{LocProp2} \) for the \( \text{Act Mov} \) event, it is necessary to add another event \( \text{Act Mov Impossible} \) that refines \( \text{Act} \) and allows to take into account the situation where the move to \( \text{loc} \) is not possible because of obstacles. \( \text{Act Mov Impossible} \) will just allow \( \text{ant} \) to return to the perception step. The same reasoning is applied for \( \text{Act Mov Drop Phero} \). For \( \text{Act Harv Food} \), we should consider the case where the food disappears before that the ant takes it.

**Second refinement**: the second refinement \( \text{Ants2} \) serves to create the links between the decision made and the corresponding action. We add the actuators of an \( \text{ant} \): \( \text{paw, exocrinGland, mandible} \) as well as the ant’s characteristic \( \text{nextLocation} \) which is updated when taking a decision. The \( \text{Decide} \) event is split into five events: 1) \( \text{Dec Mov Exp} \): decide to move for exploring the environment, 2) \( \text{Dec Mov Back} \): decide to come back to the nest, 3) \( \text{Dec Mov Drop Back} \): decide to come back while dropping pheromone 4) \( \text{Dec Harv Food} \): decide to take the food, 5) \( \text{Dec Drop Food} \): decide to drop food in the nest. As an example, we give the event \( \text{Dec Mov Exp} \) above.

As a result of event \( \text{Dec Mov Exp} \) execution, the ant chooses its next location and activates its paws. What is necessary now, is to link the activation of the...
paws with the triggering of the move action. Thus, we need to refine the event \textit{Act\_Mov} by adding a \textit{Witness} relating the parameter \textit{decideAct} in the event \textit{Act\_Mov} with the variable \textit{paw}.

\begin{verbatim}
EVENT Act_Mov
REFINES Act_Mov
ANY
ant
WHERE
grd121 : ant ∈ Ants ∧ stepAgent(ant) = perceive ∧ loc ∈ Next(currentLoc(ant))
grd4 : paw(ant) = activate

WITNESSES

decideAct : decideAct = Move ⇔ paw(ant) = activate
loc : loc = nextLocation(ant)

THEN
act12 : stepAgentCycle(ant) := perceive||currentLoc(ant) := nextLocation(ant)
act3 : paw(ant) := disabled

END
\end{verbatim}

\textbf{Third refinement:} at this level of refinement (Ants3), the ants representations about the environment are introduced. Every ant can sense food smell (\textit{food}) as well as pheromone scent (\textit{pheromone}). We introduce also the variable \textit{DePhero} modeling the distribution of pheromone in the environment. The event \textit{Perceive} (here above) is refined by adding the necessary event actions for updating the perceptions of an ant.

\begin{verbatim}
EVENT Perceive
REFINES Perceive
ANY
ant, loc, fp, php
WHERE
grd123 : ant ∈ Ants ∧ stepAgent(ant) = perceive ∧ loc = currentLoc(ant)
grd45 : fp ∈ Locations × Locations → N ∧ fp = FPerce(QuFood)
grd67 : php ∈ Locations × Locations → N ∧ php = PhPerc(DePhero)

THEN
act1 : stepAgentCycle(ant) := decide
act2 : food(ant) := \{loc → fp(loc → dir)|dir ∈ Next(loc)\}
act3 : pheromone(ant) := \{loc → php(loc → dir)|dir ∈ Next(loc)\}

END
\end{verbatim}

\textit{FPerce} (guard \textit{grd45}) and \textit{PhPerc} (guard \textit{grd67}) models the ability of an ant to smell respectively the food and the pheromone situated in its perception field. They are defined in the accompanying context of \textit{Ants3}. Moreover, we split the event \textit{Dec\_Mov\_Exp} into three events: 1)\textit{Dec\_Mov\_Rand}: decide to move to a location chosen randomly because no scent is smelt; 2)\textit{Dec\_Mov\_Fol\_F}: decide to move towards the direction where the food smell is maximum; 3)\textit{Dec\_Mov\_Fol\_Ph}: decide to move towards the direction where the pheromone smell is maximum. This split guarantees the \textit{LocProp3} property for the decision to move. The event \textit{Act\_Mov} is also refined in order to take into account these different decisions.

4.2 Formalization of the ant global properties

The three refinement steps described in the last section have enabled us to specify a correct individual behavior for the ants. Let us now focus on the ability of the modelled behavior to reach the desired global properties. As we already
mentioned, the focus of this paper is on the stability property \((S1)\) which informs about the capability of ants to exploit entirely a source of food detected.

Recall in the machine \(\text{Ants3}\), we have three events describing an exploration movement namely \(\text{Act\_Mov\_Fol\_F}\), \(\text{Act\_Mov\_Fol\_Ph}\), \(\text{Act\_Mov\_Rand}\) plus the event \(\text{Act\_Harv\_Food}\) corresponding to the action of picking up food. All these events are defined according to the parameter \(\text{loc}\) which refers to any location.

In order to prove the stability property, we refine these events by instantiating the parameter \(\text{loc}\) with a precise location of food \(\text{loc1}\). Our aim is to prove that once \(\text{loc1}\) is reached, the quantity of food in it will decrease until reaching zero. In Event-B, this kind of reasoning is possible by proving convergence (or termination) of the event responsible for decreasing this value, i.e. the event \(\text{Act\_Harv\_Food}\). For carrying out the proof of termination in Event-B, we need to use a variant, i.e. a natural number expression or a finite set and prove that event \(\text{Act\_Harv\_Food}\) decreases it in each execution. Finding an implicit variant is trivial under weak fairness assumptions on the actions of this event ([7]). In our case, the nondeterminism introduced by the movement actions makes such an assumption impossible. Indeed, \(\text{Act\_Harv\_Food}\) is not always enabled since once an ant reaches a source of food, the others can need time to reach this source.

For proving convergence, our work is inspired by the proofs done by D. Méry and M. Poppleton in [7] where they demonstrate how to prove convergence under fairness assumption by the use of the Temporal Logic of Actions (TLA) [8] and Event-B.

Let us consider the two states \(P\) and \(Q_{\text{Harvest}}\) describing the quantity of food on \(\text{loc1}\) and defined as follows:

\[
P \triangleq \text{Inv}_{\text{Ants4}} \land \text{Qu\_Food}(\text{loc1}) = n + 1, \quad Q_{\text{Harvest}} \triangleq \text{Inv}_{\text{Ants4}} \land \text{Qu\_Food}(\text{loc1}) = n
\]

\(\text{Inv}_{\text{Ants4}}\) denotes the conjunction of invariants of machine \(\text{Ants4}\). Proving the termination of \(\text{Act\_Harv\_Food}\) is reformulated by the formula:

\[
P \rightsquigarrow Q_{\text{Harvest}}.
\]

We define \(N\) and \(A_{\text{Harvest}}\) as follows.

\[
N \triangleq \text{Act\_Harv\_Food} \lor \text{Act\_Mov\_Fol\_F} \lor \text{Act\_Mov\_Fol\_Ph} \lor \text{Act\_Mov\_Rand}
\]

\[
A_{\text{Harvest}} \triangleq \text{Act\_Harv\_Food}
\]

By applying SF1, we prove \(P \rightsquigarrow Q_{\text{Harvest}}\):

\[
\begin{align*}
\text{SF1.1:} & \quad P \land (N \land Q_{\text{Harvest}}) \Rightarrow (P' \lor Q_{\text{Harvest}}) \\
\text{SF1.2:} & \quad P \land (N \land A_{\text{Harvest}}) \land \text{Qu\_Food}(\text{loc1}) \Rightarrow Q_{\text{Harvest}} \\
\text{SF1.3:} & \quad P \land (\Box(N) \land \text{Qu\_Food}(\text{loc1}) \land \Box(\text{Enabled}(A_{\text{Harvest})) \land \text{Qu\_Food}(\text{loc1}))) \Rightarrow P \rightsquigarrow Q_{\text{Harvest}}
\end{align*}
\]

Condition SF1.1 describes a progress step where either state \(P\) or \(Q_{\text{Harvest}}\) can be produced. Condition SF1.2 describes the inductive step where \(\langle A_{\text{Harvest}} \rangle \land \text{Qu\_Food}(\text{loc1})\) produces state \(Q_{\text{Harvest}}\). Condition SF1.3 ensures that \(\langle A_{\text{Harvest}} \rangle \land \text{Qu\_Food}(\text{loc1})\) will be eventually enabled. Note that both conditions SF1.1 and SF1.2 do not contain any temporal operator. As a consequence, they are expressible in Event-B. SF1.3 is a temporal formula that can be expressed in the \textit{leads to form}. Thus, we can define SF1.3 as:

\[
\begin{align*}
\text{SF1.3:} & \quad P \land (\Box(N) \land \text{Qu\_Food}(\text{loc1}) \land \text{SF\_Qu\_Food}(\text{loc1})(A_{\text{Harvest}})) \Rightarrow P \rightsquigarrow Q_{\text{Harvest}}
\end{align*}
\]
To demonstrate that condition $SF.31$ is true, we need to prove that the formula $\Diamond \text{Enabled}(\text{AHarvest})_{\text{QuFood}}(\text{loc1})$ holds. Ants are able to reach food thanks to their movements for following food. Thus if we assume that once an ant smells food, it will be able to follow it (we do not consider case where food disappears suddenly), we can argue that the event $\text{Act}_{\text{Food}}$ is always eventually $\text{Enabled}$. Consequently, we can prove $SF.31$ under weak fairness assumption.

We consider:

\[
\begin{align*}
Q_{\text{FollowFood}} & \equiv \text{Enabled}(\text{AHarvest})_{\text{QuFood}}(\text{loc1}) \quad \text{and} \quad \text{AFollowFood} \equiv \text{Act}_{\text{FollowFood}}.
\end{align*}
\]

We apply $WF1$:

\[
\begin{align*}
WF.311 & : P \land \Box [N]_{\text{QuFood}(\text{loc1})} \Rightarrow P \rightarrow (P' \lor Q_{\text{FollowFood}}) \\
WF.312 & : [N \land \text{AFollowFood}]_{\text{QuFood}(\text{loc1})} \Rightarrow Q_{\text{FollowFood}} \\
WF.313 & : P \Rightarrow \text{Enabled}(\text{AFollowFood})_{\text{QuFood}(\text{loc1})}
\end{align*}
\]

$WF.311$, $WF.312$ and $WF.313$ do not contain any temporal operator, so that they are directly expressible in $Event-B$.

## 5 Related Work

Related work cited in this section deals in the first part, with the formal modeling and verification of self-organization. The second part is devoted to the presentation of works using $Event-B$ for the development of adaptive systems.

### Formal modeling of self-organizing systems.

In [11], Gardelli uses stochastic Pi-Calculus for modeling SO-MAS for intrusion detection. This formalization was used to perform simulations using the SPIM tool to assess the impact of certain parameters, such as the number of agents and frequency of inspections, on the system behavior. In [12], a hybrid approach for modeling and verifying self-organizing systems has been proposed. This approach uses stochastic simulations to model the system described as Markov chains and the technique of probabilistic model checking for verification. To avoid the state explosion problem, encountered with model-checkers, the authors propose to use approximate model-checking based on simulations. The approach was tested for the problem of collective sorting using the PRISM tool. Konur and colleagues ([13]) use also the PRISM tool and probabilistic model checking to verify the behavior of robot swarm, particularly foraging robots. The authors verify properties expressed by PCTL logic (Probabilistic Computation Tree Logic) for several scenarios. These properties provide information, in particular, on the probability that the swarm acquires a certain amount of energy for a certain number of agents and in a certain amount of time. Simulations were also used to show the correlation between the density of foraging robots in the arena and the amount of energy gained. Most of the works exposed above use the model checking technique to evaluate the behavior of the system and adjust its parameters. Although they were able
to overcome the state explosion problem and prove the effectiveness of their approaches, these works do not offer any guidance to help the designer to find the source of error in case of problems and to correct the local behavior at the micro level. For the purpose of giving more guidance for the designer, we find that the use of Event-B language and its principle of refinement are very useful.

**Formal modeling using the Event-B language.** In [14], the authors propose a formal modeling framework for critical MAS, through a series of refinement step to derive a secure system implementation. Security is guaranteed by satisfying three properties: 1) an agent recovering from a failure cannot participate in a cooperative activity with others, 2) interactions can take place only between interconnected agents and 3) initiated cooperative activities should complete successfully. This framework is applied to model critical activities of an emergency. Event-B modeling for fault tolerant MAS was proposed in [15]. The authors propose a refinement strategy that starts by specifying the main purpose of the system, defines the necessary agents to accomplish it, then introduces the various failures of agents and ends by introducing the communication model and error recovery mechanisms. The refinement process ensures a set of properties, mainly 1) reachability of the main purpose of the system, 2) the integrity between agents local information and global information and 3) efficiency of cooperative activities for error recovery. The work of Hoang and Abrial in [16] was interested in checking liveness properties in the context of the nodes topology discovery in a network. The proposed refinement strategy allows to prove the stability property, indicating that the system will reach a stable state when the environment remains inactive. The system is called stable if the local information about the topology in each node are consistent with the actual network topology.

These works based on the correct by construction approach, often providing a top-down formalization approach, have the particularity of being exempt from the combinatorial explosion problem found with the model checking techniques. They have the advantage of allowing the designer to discover the restrictions to be imposed to ensure the desired properties. We share the same goals as the works presented i.e. ensuring liveness properties and simplifying the development by the use of stepwise refinements. Our refinement strategy was used to guide the modeling of individual behaviors of agents, unlike the proposed refinement strategies that use a top-down development of the entire system. We made this choice to be as closely as possible to the bottom-up nature of self-organizing systems.

### 6 Conclusion

We have presented in this paper a formal modelization for SO-MAS by means of Event-B. In our formalization, we consider the system in two abstraction levels: the micro and macro levels. This abstraction allows to focus the development efforts on a particular aspect of the system. We propose a stepwise refinement strategy to build a correct individual behavior. This refinement strategy is extended in order to prove global properties such as robustness and resilience. Our
proposal was applied to the foraging ants case study. While the proof obligations were used to prove the correctness of the micro level models, it was necessary to turn to TLA in order to prove the stability property at the macro-level. We think that this combination of TLA and Event-B is very promising for formal reasoning about SO-MAS. Our ambitions for future works are summarized in the following four points:

- Reasoning about the convergence of SO-MAS by means of TLA.
- Introduction of the self-organization mechanisms, based on the cooperation in particular, at the proposed refinement strategy of the local agents behavior and the analysis of the impact of these mechanisms on the resilience of the system. For the foraging ants, for example, the objective is to analyse the ability of the ants to improve the rapidity of reaching and exploiting food thanks to their cooperative attitude. To achieve this aim, we plan to use a probabilistic approach coupled with Event-B.
- Definition of design patterns for modeling and refinement of SO-MAS and their application to other case studies.
- Integration of the proposed formal framework within SO-MAS development methods in order to ensure formal proofs at the early stages of the system development. This integration will be made by using model-driven engineering techniques.

References


Capability Relationships in BDI Agents

Ingrid Nunes
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre Brazil
ingridnunes@inf.ufrgs.br

Abstract. The belief-desire-intention (BDI) architecture has been proposed to support the development of rational agents, integrating theoretical foundations of BDI agents, their implementation, and the building of large-scale multi-agent applications. However, the BDI architecture, as initially proposed, does not provide adequate concepts to produce modular software components. The capability concept emerged to address this issue, but the relationships between capabilities have been insufficiently explored to support the development of BDI agents. We thus, in this paper, introduce and analyse three possible relationships among capabilities in BDI agent development — namely association, composition and generalisation — which are widely used in object-oriented software development, and are fundamental to develop software components with low coupling and high cohesion. Our goal with this paper is to promote the exploitation of these and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering.

Keywords: Capability, Modularisation, BDI Architecture, Agent-oriented Development.

1 Introduction

The belief-desire-intention (BDI) architecture is perhaps the most adopted architecture to modelling and implementing rational agents. It has foundations in a model proposed by Bratman [3], which determines human action based on three mental attitudes: beliefs, desires and intentions. Based in this model, Rao and Georgeff [16] proposed the BDI architecture, integrating: (i) theoretical work on BDI agents; (ii) their implementation; and (iii) the building of large-scale applications based on BDI agents. Although their work has been widely used to model and implement BDI agents in theory and practice in academy, there is no real evidence that this approach scales up.

Much work on software engineering aims to deal with the complexity of large-scale enterprise software applications to support their development, and a keyword that drives this research is modularity. Software developed with modular software components — i.e. components with high cohesion and low coupling
properties — are more flexible and easier to reuse and maintain. Although modularity is highly investigated in the context of mainstream software engineering, it has been poorly addressed not only in work on BDI agents, but also by the agent-oriented software engineering community. Research in this context is limited to few approaches, for example, modularisation of crosscutting concerns in agent architectures with aspects [9, 17] and the use of capabilities in BDI agent architectures [6, 4].

We, in this paper, investigate the concept of capability, in order to allow the modular construction of BDI agents, with the aim of supporting the development of large-scale systems based on BDI agents (hereafter, agents). Capabilities are modules that are part of an agent, and they cluster a set of beliefs and plans that together are able to handle events or achieve goals. Therefore, it modularises a particular functional behaviour that can be added to agents. The capability concept is available in some of the BDI agent platforms [10, 12, 15]; however, there is divergence on its implementation, and therefore there is no standard structure for this concept. One commonality shared by different capability implementations is the ability to include capabilities to another, but this relationship also varies in the different available implementations, as well as their implications in the agent reasoning cycle at runtime. Moreover, there is a single type of relationship between capabilities in each implementation. This differs from the object-oriented paradigm, which allows to establish many types of relationships between software objects.

We thus present an initial work on the investigation of different types of relationships that may occur between capabilities, introducing three of them, namely association, composition and generalisation. Besides describing each type of relationship, we analyse how a pair of related capabilities work together in the context of the agent reasoning. These relationships may be used in combination to design and implement an agent, and we show examples of this scenario. The presented relationships provide the basis for a discussion with respect to engineering aspects of agents, which support the construction agent-based systems. Our aim is to promote the exploitation of these and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering.

The remainder of this paper is organised as follows. We first introduce work related to capabilities in Section 2. Then, we describe the different capability relationships in Section 3, and exemplify their combined use in Section 4. We next analyse and compare these relationships in Section 5, also showing how each of the existing BDI platforms that provide the capability concept implement it. Finally, we conclude this paper in Section 6.

2 Related Work

We begin by presenting work that has been done in the context of capabilities. The capability concept was introduced by Busetta et al. [6] and emerged from experiences with multi-agent system development with JACK [10, 1], a BDI
agent platform. The goal was to build modular structures, which could be reused across different agents. In Table 1, we detail the parts that comprise a capability according to this work. Some of which are specific to the JACK platform, such as the explicit specification of perceived events.

This work is the result of practical experience, so Padgham and Lambrix [13] formalised the capability concept, in order to bridge the gap between theory and practice. This formalisation included an indication of how capabilities can affect agent reasoning about its intentions. In order to integrate capabilities to the agent development process, Penserini et al. [14] proposed a tool-supported methodology, which goes from requirements to code. It identifies agent capabilities at the requirement specification phase, based on the analysis models of Tropos [5], and is able to eventually generate code for Jadex [15], another BDI agent platform.

Among the different available platforms to implement BDI agents, such as Jason\(^1\) [2] and the 3APL Platform\(^2\), three implement the capability concept: JACK\(^3\) [10], Jadex\(^4\) [15, 4], and BDI4JADE\(^5\) [12]. As we already discussed how JACK capabilities are implemented, we next detail the other two implementations, which include a capability identifier.

A Jadex capability is composed of: (i) beliefs; (ii) goals; (iii) plans; (iv) events; (v) expressions; (vi) properties; (vii) configurations; and (viii) capabilities. Some of these parts are platform-specific, such as expressions, which are expressions written in a language that follows a Java-like syntax and are used for different

---

Table 1: Capability Specification.

<table>
<thead>
<tr>
<th>PART</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>The capability identifier, i.e. a name.</td>
</tr>
<tr>
<td>Plans</td>
<td>A set of plans.</td>
</tr>
<tr>
<td>Beliefs</td>
<td>A set of beliefs representing a fragment of knowledge base and manipulated by the plans of the capability.</td>
</tr>
<tr>
<td>Belief Visibility Rules</td>
<td>Specification of which beliefs are restricted to the plans of the capability and which ones can be seen and manipulated from outside.</td>
</tr>
<tr>
<td>Exported Events</td>
<td>Specification of event types, generated as a consequence of the activity of the capability, that are visible outside its scope, and their processing algorithm.</td>
</tr>
<tr>
<td>Perceived events</td>
<td>Specification of event types, generated outside the capability, that are relevant to the capability.</td>
</tr>
<tr>
<td>Capabilities</td>
<td>Recursive inclusion of other capabilities.</td>
</tr>
</tbody>
</table>

---

\(^1\) http://jason.sourceforge.net/
\(^2\) http://www.cs.uu.nl/3apl/
\(^3\) http://aosgrp.com/products/jack/
\(^4\) http://www.activecomponents.org
\(^5\) http://www.inf.ufrgs.br/prosoft/bdi4jade/
purposes, e.g., goal parameters or belief values. Beliefs can be used only within the scope of the capability, exported to outside the capability scope, or abstract, meaning that a value of a belief outside the capability may be assigned to this abstract belief. The BDI4JADE capability, on the other hand, is composed of: (i) a belief base; (ii) a plan library; and (iii) other capabilities. These are the explicit capability associations. As BDI4JADE is written in pure Java (no XML files), other properties may be obtained by manipulating the capability parts, besides the described components.

Given this analysis of existing work on capabilities, we next introduce three different types of relationships between capabilities. As said before, all the implementations of the capability concept provide limited relationship types, and after introducing our relationship types, we will revisit these capability implementations in Section 5, indicating the meaning of their capability relationship.

3 Relationships between Capabilities

According to the object-oriented paradigm, a system is composed of software objects, which integrate code and data. Such objects are building blocks to construct complex structures, and can be combined using different forms of relationships. In this section, we analyse three of these relationships — association (Section 3.1), composition (Section 3.2), and inheritance (Section 3.3) — in the context of capabilities.

3.1 Association

Software objects encapsulate both state (represented by attributes) and behaviour (represented by methods), and are accessed through its interface, which is a collection of method signatures. In order for a system to implement functionality, objects collaborate by invoking methods of other objects with which they are associated.

Similarly, capabilities implement some functionality, and have both state (represented by beliefs) and behaviour (represented by plans). The main difference from the object concept is that, while methods that are part of an object interface can be directly invoked by other objects, plans are dispatched within the context of the agent reasoning cycle, and its execution is triggered by a goal or, in some BDI models, an event. As a consequence, in order for an agent behaviour to be the result of the interaction of more than one capability, an important question arises: what is a capability interface?

In a capability, beliefs are a piece of encapsulated knowledge, and are manipulated by the capability’s plans. Consequently, following the principle of information hiding, the manipulation of beliefs are restricted to the capability. Plans, which correspond to methods, cannot be explicitly invoked. Therefore, they are accessible only within the context of the capability, and are not part of the capability interface as well. Goals, on the other hand, indicate the objectives that a capability may achieve, and possibly there are different capability plans
that can be used to achieve such goals. Therefore, goals represent services that a capability may provide to another, and thus comprise its interface. This is illustrated in capabilities of Figure 1, in which goals are in the border of the capabilities. Note, however, that there are goals used only internally, and are not part of the capability interface.

Given that we now have an interface for capabilities — specified in terms of a set of goals that a capability may achieve — we are able to associate capabilities so that they can collaborate. An association is a relationship, shown in Figure 1a, where a source capability $C_S$ uses a target capability $C_T$, by delegating goals to be achieved by $C_T$. In the context of the agent reasoning cycle, it means that during the execution of plans that belong to $C_S$, goals that are part of the $C_T$ interface may be dispatched, and only plans that belong to $C_T$ are candidates to be selected to handle such goals. This is similar to the notion of delegating a goal to another agent, but two agents mean two threads of execution, whereas two capabilities of one agent consist of a single thread of execution.

Consider the scenario in which we are developing an intelligent robot, which is responsible for household duties, such as cleaning the floor and washing clothes. For both these duties, the robot has to move around and, while executing plans for cleaning the floor and washing clothes, the robot has to achieve a subgoal
move($x, y$), i.e. move from a position $x$ to a position $y$. In this case, our robot may have three capabilities — FloorCleaning, Laundry, and Transportation capabilities — and there are association relationships from the FloorCleaning and Laundry capabilities to the Transportation capability, which has an external goal $move(x, y)$, part of its interface.

We present in Figure 1b the visibility of components of the target capability by the source capability. In this figure, and others presented throughout the paper, we show what the capability with a white background can access from the capability with the gray background. All components within the scope of the target capability are hidden and inaccessible by the source capability, except the goals that are part of the target capability interface. Such goals may be dispatched by plans of the source capability. The target capability, on the other hand, is not aware of the source capability.

Although the association relationship is directed, it may be bidirectional. In order to better modularise an agent architecture, functionality associated with two different concerns may be split into two capabilities, and they may use each other to achieve their goals.

### 3.2 Composition

The association relationship allows us to modularise BDI concepts into two capabilities — composed of beliefs, goals, and plans — and each of which should address a different concern, thus having high cohesion. The connection between these capabilities is that the execution of at least one plan of the source capability requires achieving a goal that is part of the target capability. In this case, each capability uses the knowledge captured by their own beliefs to execute their plans.

However, there may be situations in which there should be shared knowledge between capabilities, that is, a capability uses the information stored in other capability’s beliefs in the execution of its plans. In this case, the composition relationship is used, which increases the coupling between the two involved capabilities. This kind of relationship expresses the notion of containment, and its structure is presented in Figure 2a.

An agent may be built by first developing functionality to achieve lower level goals, and then using it to develop higher level functionality. For example, assume that the FloorCleaning capability of the robot agent must have goals, beliefs and plans to both sweep the floor and vacuum the dust, when there are carpets on the floor. As these are two different concerns, they may be modularised into two capabilities, each being composed of the external goals related to their respective duty to be accomplished. The FloorCleaning capability, by having a composition relationship with the Sweeper and the VacuumCleaner capabilities, can thus dispatch external goals of these two capabilities — while executing a plan to clean a room, for instance. This can also be performed using the association relationship, but now there are two differences. First, the Sweeper and the VacuumCleaner capabilities can have plans to handle FloorCleaning’s goals, so if goals are dispatched in plans of this capability, they may be achieved by
plans of the composed capabilities. Second, the FloorCleaning capability may have knowledge stored in its beliefs, such as those related to the environment, and they need to be used to both sweep the floor and to vacuum the dust. So by composing the FloorCleaning capability with the other two, the Sweeper and VacuumCleaner capabilities may access the FloorCleaning’s beliefs in the execution of their plans.

The visibility of the components of the two capabilities involved in a composition relationship, namely the whole and the part, are shown in Figure 2. Figure 2b shows that the whole-capability is able to dispatch external goals of part-capability, but cannot access other components. And Figure 2c details that the part-capability can access both the beliefs and goals of the whole-capability.

This relationship is transitive. Consider a capability C that is part of a capability B, which in turn is part of a capability A. Therefore, C can access beliefs of both B and A in addition to its own beliefs, and A’s goals can be handled by plans of both B and C, in addition to its own plans. As a consequence, different compositions may be performed with capabilities that implement low level behaviour.
3.3 Inheritance

While the association and composition relationships focus on collaborating capabilities, the goal of the inheritance relationship — which will now be discussed — is mainly to promote reuse, by generalising common behaviour in a parent capability and specialising it in children capabilities. This relationship increases the coupling between the involved capabilities, with respect to the other two types of relationships. It is also transitive, that is, a child capability inherits from its parent’s parent.

The development of a multi-agent system may involve building agents that share a common behaviour, but have some particularities that distinguish one from another. In this case, we may need to design a capability with a set of beliefs, goals, and plans, to which other goals, beliefs and plans must be added to develop particularities. The inheritance relationship thus allows to connect this common behaviour to specialised variable behaviour. This relationship is illustrated in Figure 3a.

When a capability extends another, it inherits all the components of the parent capability. Therefore, the components of a child capability can be seen as the union of its components — beliefs, goals, and plans — with its parent’s

![Inheritance Diagram](image-url)
components. Such parent’s components can be accessed within the scope of the child capability, that is, the child capability can: (i) dispatch both external and internal parent’s goals; (ii) access and update parent’s beliefs while executing its plans; (iii) have a goal handled and achieved by the parent’s plans; and (iv) handle and achieve parent’s goals. This full access to the parent capability’s components by the child capability is shown in Figure 3b. The parent capability, in turn, is not aware that there are capabilities that extend its behaviour.

We will now illustrate a situation where inheritance may be used in the context of the development our intelligent robot. Assume that we have physical robots, which are provided with some basic features, such as walking, moving arms, and so on, so that they are able to perform different household duties, depending on the software deployed on them. We are developing robots for both helping in homes and working on laundries. The Laundry capability should have plans to wash clothes in the wash machine and to hand washing, if the robot is for helping at home and, and if it will work on laundries, it should also have components to dry cleaning. Therefore, two capabilities may be designed: Laundry and ProfessionalLaundry. The latter extends the former, adding new beliefs, goals, and plans needed to provide the dry cleaning functionality.

4 Using Capability Relationships

Given that we presented the three capability relationships, we illustrate their use in this section. We gave examples of their individual use in the previous section within the same context, the intelligent robot example. In Section 4.1, we combine the examples previously given by providing a big picture of the design of our intelligent robot. In Section 4.2, we provide further examples of the use of capability relationships in the context of transportation.

4.1 Intelligent Robots

We provided many examples in the context of robot development, where the capability relationships may be applied to modularise agent concerns. We now present an integration of these different examples to show how relationships can be used together in the development of agents. An overview of the design of the intelligent robots example is presented in Figure 4. This is an overview, and therefore this figure does not correspond to the complete design of a system, many agent components are omitted.

We use a simple notation. Capabilities are represented with rectangles, split into four compartments: (i) capability name; (ii) goals; (iii) beliefs; and (iv) plans. For relating capabilities, we use the notation previously introduced. And we represent agents with ellipses, and an agent is an aggregation of capabilities.

The Laundry capability provides the basic functionality for washing clothes, and it is extended by the ProfessionalLaundryCapability — an instance of the latter adds the ability of dry washing to the former. The Laundry capability is associated with the Transportation capability, so that the Laundry
capability can dispatch goals related to transportation. Note that, because the ProfessionalCapability capability extends the Laundry capability, it also inherits the association.

The FloorCleaning capability has a goal (clean), which is not handled by any plan within this capability. It is, however, composed of two other capabilities, each having a plan that can achieve it, so that they can be selected to achieve the clean goal when appropriate (remember that these capabilities have other omitted beliefs, goals and plans). The execution of plans of the Sweeper and VacuumCleaner capabilities also needs goals related to transportation to be achieved, thus both of them are associated with the Transportation capability.

These capabilities are the building blocks to develop agents. A Maid agent (that is used to help at home) is an aggregation of both the FloorCleaning and Laundry capabilities, so that is can perform tasks related to them. The Laudress agent (who performs duties at laundries) must be able to perform other tasks related to washing clothes, therefore it is an aggregation of the ProfessionalLaundry capability, which in turn inherits the behaviour of its parent capability.

4.2 Driver Agents

We now will introduce a second example, which is in the transportation context. The objective is to design agents able to drive cars and motorcycles. As above,
Fig. 5: Example: Intelligent Robots.

we will show an overview of the design, highlighting important parts of it, and
omitting details. This example is illustrated in Figure 5.

The key functionalities associated with driving are implemented as part of
the Driver capability, which has beliefs with respect to the current speed and
location, an external goal drive\((x, y)\) that it is successfully achieved when the
agent has driven from location \(x\) to location \(y\), and internal goals dispatched by
plans whose aim is to achieve the drive\((x, y)\) goal. There are two extensions of
this capability: MotoDriver and CarDriver, which specialise the Drive capa-
bility to add behaviour specific to driving a motorcycle and a car, respectively.
Besides other omitted details, each has its own plans to perform similar tasks,
such as accelerating.

To drive from a location \(x\) to \(y\), the Driver capability must first find a
route between these two locations. This is modularised into the RoutePlanner
capability, which has knowledge needed to calculate a route (maps, congestion
zones, agent preferences, etc.), and different plans to find a route. To be able to
find the route, the Driver capability has an association with the RoutePlanner
capability, and consequently it can dispatch the findRoute\((x, y)\) goal.

Finally, there is a complicated part related to driving, which is the control
of gears. This can be modularised in a separate capability, which needs specific
beliefs, goals and plans to do so. However, it also needs the knowledge that is part
of the Driver capability, and consequently there is a composition relationship between the Driver and GearController capabilities.

In order to build agents able to drive a motorcycle or a car, an agent must aggregate the MotoDriver capability or CarDriver capability, respectively.

5 Discussion

In this section, we discuss relevant issues with respect to the described capability relationships. We first analyse them, point out their main differences and the impact of choosing one or another in Section 5.1. In Section 5.2, we describe details of how capabilities are implemented in existing BDI platforms, and which kind of capability relationship they provide. We next discuss in Section 5.3 other object-oriented concepts, and how they are related to the presented relationships.

5.1 Relationship Analysis and Comparison

We have presented three different kinds of relationships between capabilities, and understanding their differences in order to be able to choose one to be used in agent design is important. We thus in this section make this discussion.

First, a key difference among these relationships is their purpose. Associations should be used when different independent agent parts collaborate to achieve a higher level goal. This is similar to collaborations among agents, but capabilities are within the scope of a single agent, i.e. a single thread. Therefore, it is a design choice to develop two agents, each of which with one capability and collaborating through messages, or to develop a single agent with two capabilities, collaborating by dispatching goals to be achieved by the other capability. Composition is adopted when the agent behaviour can be decomposed into modular structures, but parts depend on the whole, providing the notion of a hierarchical structure. And inheritance is used when there is a need for reusing a common set of beliefs, goals and plans, and then specialising it in different ways.

According to software engineering principles, the lower the coupling between capabilities, the better. Additionally, components of each capability should have high cohesion. These presented relationships have different degrees of coupling between the involved capabilities, so consequently relationships that reduce coupling should be preferred, when possible. We summarise this comparison of the relationships — discussed in the previous sections — in Table 2, which also indicates the visibility of components of capabilities involved in the relationships. For example, when there is an association relationship, the whole-capability has access to the part-capability’s beliefs, while the part-capability has access to the whole-capability’s beliefs, external goals and internal goals. We also emphasise the purpose of each relationship. Therefore, choosing a certain capability relationship is a design choice that not only implies restrictions over the visibility of the capability components, but also expresses the meaning of the relationship.

Now, we will focus on the impact at runtime of choosing different capability relationships. When a capability has access to components of another capability,
it may use these components at runtime. The access to beliefs is already shown in Table 2, and this means that a capability can use and modify knowledge to which it has access. Besides accessing other capability’s knowledge, a capability involved in a relationship may: (i) dispatch goals of another capability when one of its plans is executing; and (ii) execute a plan to achieve a goal of another capability. We show when these two possibilities can happen in Table 3, which are associated with goal visibility. For example, if a whole-capability (of a composition relationship) dispatches one of its goals, this goal may be achieved by the execution of a whole-capability’s plan or a plan of any the part-capabilities (and their parts).

5.2 Capabilities in Existing BDI Platforms

In Section 2, we introduced three BDI agent platforms that provide the capability concept. We will now discuss how each of these platforms provide capability relationships.
The JACK platform explicitly provides a single type of relationship: composition, allowing the construction of a hierarchical structure. Nevertheless, its interpretation is not the same as that adopted in this paper. When this relationship is declared, the visibility of the involved capabilities’ components should also be specified. Beliefs may be imported (i.e. shared with its enclosing agent or capability), exported (i.e. accessible from its parent capability), or private (i.e. local to the capability). Events have the role of goals in JACK, and in this platform capabilities should explicitly declare the kinds of events that it is able to handle or post. When declaring this information, an exports modifier is used to indicate whether events are to be handled only within the scope of the capability or by any other capability.

Although using these modifiers increases the flexibility of the platform, it goes against the principle of information hiding. When a belief is exported, any other capability can access it, and this increases the possibility of breaking the code. Although in object-orientation sometimes attributes are exposed through getters and setters, this still preserves encapsulation, as a getter hides if the value being returned is the value of an attribute or something else. The semantics of handling exported events is similar to that we adopt with the goal visibility in compositions.

Note that using solely capability compositions results in limiting capabilities to be used as hierarchical structures.

Jadex Jadex extended the capability concept of JACK [4], providing a model in which the connection between an outer and an inner capability is established by a uniform visibility mechanism for contained components. The implemented relationship type is also composition, but it is more flexible by allowing the declaration of abstract and exported components.

In Jadex, any component (beliefs, goals, plans and so on) can be used only internally, if no modifier is specified. They can be exported, and thus accessed outside the capability scope. In addition, they may be declared as abstract, and be set up by an outer capability. This way of modelling capabilities is similar to that discussed above, and have the same issues.

Jadex was recently extended by changing its implementation based on XML files to an implementation based on pure Java, as BDI4JADE, making an extensive use of Java annotations. This makes the implementation of capabilities more flexible, as all object-oriented features can be used.

BDI4JADE BDI4JADE provides a flexible implementation as it is implemented in pure Java. Goals are declared as Java classes, and therefore can be used in different capabilities. Moreover, Java modifiers can be used to limit goal visibility, for instance, by using a package visibility.

As the other two agent platforms discussed, it implements only the composition relationship. However, beliefs are always private to the capability, or

http://www.activecomponents.org/
accessible by its included capabilities. A goal is dispatched in a plan with a specification of its scope. There are two possibilities: (i) it can be handled by any plan of any capability; or (ii) it can be handled by the capability whose plan dispatched the goal, or any other included capability. Therefore, this implementation is the closest to the composition relationship described here.

It is also possible to extend capabilities in BDI4JADE as capabilities are Java classes. However, if the belief base or plan library of the parent capability is overridden by the child capability, the inheritance will lose its meaning.

5.3 Further OO Concepts

In this paper, we propose the use of relationships from object orientation to improve the modularity promoted by capabilities. This is just one of the object-oriented mechanisms that support the construction of high-quality software systems from a software engineering point of view. In this section, we discuss other mechanisms that may be adopted.

First, attributes and methods are always associated with an explicitly specified visibility, which can be private, protected, or public. JACK and Jadex, as previously discussed, provide similar concept using the `export` keyword. Here, we do not propose to use of visibility modifiers, except for goals, because exposing capability’s beliefs goes against the principles of encapsulation and information hiding. In some situations, it is needed, and we provide mechanisms that explicitly show why there is a need for sharing beliefs, i.e. when there is a whole-part structure, and the parts involved. Nevertheless, visibility may be helpful to restrict the access of part or child capabilities to components of the whole or parent capabilities, respectively.

Associations between objects usually have a cardinality specified. If this is also applied to capabilities, it will allow capabilities to be associated to more than one instance of a capability. However, dispatching a goal of any of these capabilities will produce the same effect, unless their fragments of knowledge have different states. But this is unreasonable. This is also the case of overriding components of extended capabilities, when using inheritance, or using abstract capabilities. We are not stating that any of these mechanisms should not be used, but they should be carefully analysed before being adopted in the context of capabilities, in order to evaluate their usefulness and their meaning.

Finally, configurations of how capabilities are structured can be investigated, so as to form design patterns [8], or anti-patterns that should be avoided, such as object-oriented code-smells [7].

6 Final Considerations

Modularisation plays a key role in software engineering and is crucial for developing high-quality large-scale software. However, it has limited investigation in agent architectures, or more specifically BDI agents. Our previous studies have
shown that there is a lack of mechanisms that allow modularising fine-grained variability in BDI agents [11].

Capabilities are one of the most important contributions to allow the construction of modularised BDI agent parts, increasing maintainability and promoting reuse. Nevertheless, this concept could be further explored to provide more sophisticated tools to increase the quality of BDI agents from a software engineering point of view, and supporting the construction of large-scale multi-agent systems. In this paper, we investigated the use of three types of relationships between capabilities, which are association, composition and inheritance. Each of which has a particular purpose, and indicates specific access to its components. We showed examples of their use, and discussed the implications of each relationship at runtime. Although some BDI agent platforms provide mechanisms to emulate these relationships, by means of the exportation of capability’s components, they are not in accordance with the principle of information hiding. Furthermore, keeping track of all shared beliefs and capabilities that can handle goals may become an error-prone task, thus making agents susceptible to faults.

The main goal of this paper is to promote the exploitation of capability relationships and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering. In this context, this work has left many open issues to be further discussed, with respect to capabilities and modularisation into agent architectures: (i) does it make sense to add visibility to all BDI agent components? (ii) does it make sense to design and implement abstract capabilities? (iii) is there any situation where there should be cardinality in the association relationship? and (iv) what is the interface of an agent and of a capability?

References


A scalable runtime platform for multiagent-based simulation

Tobias Ahlbrecht, Jürgen Dix, Michael Köster, Philipp Kraus, and Jörg P. Müller

Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld, Germany
firstname.lastname@tu-clausthal.de
http://www.in.tu-clausthal.de

Abstract. Using purely agent-based platforms for any kind of simulation requires to address the following challenges: (1) scalability (efficient scheduling of agent cycles is difficult), (2) efficient memory management (when and which data should be fetched, cached, or written to / from disk), and (3) modelling (no generally accepted meta-models exist: what are essential concepts, what implementation details?). While dedicated professional simulation tools usually provide rich domain libraries and advanced visualisation techniques, and support the simulation of large scenarios, they do not allow for “agentization” of single components. We are trying to bridge this gap by developing a distributed, scalable runtime platform for multiagent simulation, MASeRaTi, addressing the three problems mentioned above. It allows to plug-in both dedicated simulation tools (for the macro view) as well as the agentization of certain components of the system (to allow a micro view). If no agent-related features are used, its performance should be as close as possible to the legacy system used.

Paper type: Technological or Methodological

1 Introduction

In this paper, we describe ongoing work on a distributed runtime platform for multiagent simulation, MASeRaTi, that we are currently developing in a joint project (http://simzentrum.de/en/projects/desim). The idea for MASeRaTi evolved out of two projects, Planets and MAPC.

Agent-based traffic modelling and simulation: We developed ATSim, a simulation architecture that integrates the commercial traffic simulation framework AIMSuN with the multiagent programming system JADE (implemented in JAVA): ATSim was realized within Planets, a project on cooperative traffic management (http://www.tu-c.de/planets).

Agent-based simulation platform: We implemented, in JAVA, an agent-based platform, MASSim, which allows several simulation scenarios to be plugged-in. Remotely running teams of agents can connect to it and play against each
other on the chosen scenario. MASSim has been developed since 2006 and is used to realise the MAPC, an annual contest for multiagent systems.

While the former system centers around a commercial traffic simulation platform (AIMSuN), the latter platform is purely agent-based and had been developed from scratch. Such an agent-based approach allows for maximal freedom in the implementation of arbitrary properties, preferences, and capabilities of the entities. We call this the micro-level: each agent can behave differently and, possibly, interact with any other agent.

The traffic simulation platform AIMSuN, which works easily for tens of thousands of vehicles, however, does not support such a micro-level view. Often we can only make assumptions about the throughput or other macro-features. Therefore, with ATSim, we aimed at a hybrid approach to traffic modelling and integrated the JADE agent platform in order to describe vehicles and vehicle-to-X (V2X) communication within a multiagent-based paradigm. One of the lessons learned during the project was that it is extremely difficult to agentize certain entities (by, e.g. plugging in an agent platform) or to add agent-related features to AIMSuN in a scalable and natural way.

Before presenting the main idea in more details in Section 2, we point to related work (Section 1.1) and comment about the overall structure of this paper.

1.1 Related work
In the last decade a multitude of simulation platforms for multiagent systems have been developed. We describe some of them with their main features and note why they are not the solution to our problem. Shell for Simulated Agent Systems (SeSAm) [22] is an IDE that supports visual programming and facilitates the simulation of multiagent models. SeSAm’s main focus is on education and not on scalability.

GALATEA [9] is a general simulation platform for multiagent systems developed in Java and based on the High Level Architecture [24]. PlaSMA [14] was designed specifically for the logistics domain and builds upon JADE. AnyLogic (http://www.anylogic.com/) is a commercial simulation platform written in Java that allows to model and execute discrete event, system dynamics and agent-based simulations, e.g. using the included graphical modelling language. MATSim (http://www.matsim.org/) was developed for large-scale agent-based simulations in the traffic and transport area. It is open-source and implemented in Java. The open-source simulation platform SUMO [23] was designed to manage large-scale (city-sized) road networks. It is implemented in C++ and supports a microscopic view of the simulation while it is not especially agent-based.

Mason [26] is a general and flexible multiagent toolkit developed for simulations in Java. It allows for dynamically combining models, visualizers, and other mid-run modifications. It is open-source and runs as a single process. NetLogo[30]

---

1To agentize means to transform given legacy code into an agent so that it belongs to a particular multiagent system (MAS). This term was coined in [29]. In [28], Shoham used the term agentification for this.
is a cross-platform multiagent modelling environment that is based on Java and
employs a dialect of the Logo language for modelling. It is intended to be easily
usable while maintaining the capability for complex modelling.

TerraME ([http://www.terrame.org](http://www.terrame.org/)) is a simulation and modelling frame-
work for a terrestrial system which is based on finite, hybrid, cellular automata
or situated agents. We are using a similar architecture (Section 3), but we add
some features for parallelisation and try to define a more flexible model and
architecture structure.

Most frameworks with IDE support are not separable, so the architecture
cannot be split up into a simulation part (e.g., on a High Performance Comput-
ing (HPC) cluster) and a visualisation/modeling part for the UI. Therefore an
enhancement with HPC structure produces a new design of large parts of the
system. Known systems like Repast HPC ([http://repast.sourceforge.net](http://repast.sourceforge.net/))
use the parallelisation structure of the message passing interface MPI ([http://
/www.mcs.anl.gov/research/projects/mpi](http://www.mcs.anl.gov/research/projects/mpi/)) but the scenario source code
must be compiled into platform specific code. Hence, the process of developing
a working simulation requires a lot of knowledge about the system specifics.

Repast HPC defines a parallel working agent simulation framework written
in C++. In addition to our concept, Repast uses a similar structure to spawn
environment and agents over the process and defines local and non-local agents.
Technically, it uses Boost and Boost.MPI to create the communication between
the processes. A dedicated scheduler defines the simulation cycle. A problem of
Repast HPC is the “hard encoding” structure of the C++ classes, which requires
good knowledge about the Repast interface structure. In our architecture, we
separate the agent and scheduling structure into different parts, creating a better
fit of the agent programming paradigm and the underlying scheduler algorithms.

Also, a number of meta models for multiagent-based simulation (MABS)
have been developed so far. AMASON [21] represents a general meta-model
that captures the basic structure and dynamics of a MABS model. It is an ab-
straction and does not provide an implementation. MAIA [15] takes a different
approach by building the model on institutional concepts and analysis. The re-
sulting meta-model is very detailed, focusing on social aspects of multiagent
systems. easyABMS [13] provides an entire methodology to iteratively and vi-


ually develop models from which code for the Repast Simphony toolkit can be
generated. The reference meta model for easyABMS is again very detailed
making it possible to create models with minimal programming effort.

To summarize, we find that most platforms are either written in Java or are
not scalable for other reasons. Many are only used in academia and simply not
designed to run on a high performance computing (HPC) cluster. Common chal-
enges relate to agent runtime representation and communication performance.

### 1.2 Structure of the paper

In Section 2 we discuss our past research (ATSIm and MASSim), draw conclu-
sions and show how it led to the new idea of a highly scalable runtime platform
for simulation purposes. We also give a more detailed description of the main
features of MASeRaTi and how they are to be realized. The main part of this paper is Section 3, where we describe in some detail our simulation platform, including the system meta-model and the platform architecture. Appendix 4 presents a small example on which we are testing our ideas and the scalability of the system as compared to MASSim, a purely agent-based approach implemented in Java. We conclude with Section 5 and give an outlook to the next steps to be taken.

2 Essential features of MASeRaTi

In this section, we first present our own research in developing the platforms ATSim (Subsection 2.1) and MASSim (Subsection 2.2). We elaborate on lessons learned and show how this resulted in the new idea of the scalable runtime platform MASeRaTi (Subsection 2.3).

2.1 Traffic simulation (ATSim)

Most models for simulating today’s traffic management policies and their effects are based on macroscopic physics-based paradigms, see e.g. [17]. These approaches are highly scalable and have proven their effectiveness in practice. However, they require the behaviour of traffic participants to be described in simple physical equations, which is not necessarily the case when considering urban traffic scenarios. Microscopic approaches have been successfully used for freeway traffic flow modelling and control [27], which is usually a simpler problem than urban traffic flow modelling and control, due to less dynamics and better predictability.

In [8], we presented the ATSim simulation architecture that integrates the commercial traffic simulation framework AIMSuN with the multiagent programming system JADE. AIMSuN is used to model and simulate traffic scenarios, whereas JADE is used to implement the informational and motivational states and the decisions of traffic participants (modelled as agents). Thus, all features of AIMSuN (e.g. rich GUI, tools for data collection and data analysis) are available in ATSim, while ATSim allows to simulate the overall behaviour of traffic, and traffic objects can be modelled as agents with goals, plans, and communication with others for local coordination and cooperation.

AIMSuN (Figure 1(a) left side) provides an API for external applications to access its traffic objects via Python or C/C++ programming languages. However, the JADE-based MAS (right side of Figure 1(a)) is implemented in Java. To enable AIMSuN and the MAS to work together in ATSim, we used CORBA as a middleware. Technically we implemented a CORBA service for the MAS and an external application using the AIMSuN API to access the traffic objects simulated by AIMSuN. The CORBA service allows our external application to interact with the MAS directly via object references. For details on the integration architecture, we refer to [8]. Two application scenarios were modelled and evaluated on top of ATSim: The simulation of decentralized adaptive routing
strategies, where vehicle agents learn local routing models based on traffic information [12], and cooperative routing based on vehicle group formation and platooning [16]. The overall system shown in Figure 1(a) was developed in a larger research project and contained additional components for realistic simulation of V2X communication (extending the OMNET++ simulator), and for formulating and deploying traffic control policies; see [11].

Our evaluation of the ATSim platform using a mid-sized scenario (rush hour traffic in Southern Hanover, one hour, approx. 30,000 routes, see [11]) showed that while the agent-based modelling approach is intuitive and suitable, our integration approach runs into scalability issues. Immediate causes identified for this were the computationally expensive representation of agents as Java threads in Jade and the XML-based inter-process communication between Jade and the AIMSUN simulator. In addition, system development and debugging proved difficult because two sets of models and runtime platforms needed to be maintained and synchronised.

2.2 Multi-Agent Programming Contest (MASSim)

The MASSim platform [5,4] is used as a simulation framework for the Multi-Agent Programming Contest (MAPC) [2](http://multiagentcontest.org). Agents are running remotely on different machines and are communicating in XML with the server over TCP/IP. The server computes the statistics, generates visual output and provides interfaces for the simulation data while the simulation is running.

A drawback of dividing the simulation in such a way is the latency of the network that can cause serious delays. Network communication becomes a bottleneck when scaling up; the slowest computer in the network is determining the overall speed of the simulation. Running the simulation in one Java virtual machine leads to a centralised approach that might impede an optimal run (in terms of execution time) of a simulation.
Figure 1(b) depicts the basic components of the MASSim platform. MASSim will mainly serve us as a reference to compare scalability with MASeRaTi right from the beginning (using the available scenarios). We want to ensure that MASeRaTi outperforms MASSim in both computation time and number of agents.

2.3 MASeRaTi: The underlying idea

Our new simulation platform, MASeRaTi [http://tu-c.de/maserati](http://tu-c.de/maserati), aims at combining the versatility of an agent-based approach (the micro-view) with the efficiency and scalability of dedicated simulation platforms (the macro-view). We reconsider the three challenges mentioned in the abstract for using a purely agent-based approach.

**Scalability:** Efficient scheduling of agent cycles is a difficult problem. In agent platforms, usually each agent has his own thread. Using e.g. Java, these threads are realised in the underlying operating system which puts an upper limit of 5000 agents to the system. These threads are handled by the internal scheduler and are therefore not real parallel processes. In the MASeRaTi architecture we develop a micro-kernel where agents truly run in parallel. In this way, we reduce the overhead that comes with each thread significantly. We believe that this allows for a much better scalability than agent systems based on (any) programming language, where all processes are handled by the (black-box) operating system. Additionally, many simulation platforms use a verbose communication language (e.g., XML or FIPA-ACL) for the inter-agent communication that becomes a bottleneck when scaling up. We exploit the efficient synchronisation features of MPI instead.

**Efficient memory management:** Which data should when be fetched from disk (cached, written)? Most agent platforms are based on Java or similar interpreter languages. When using them we have no control over the prefetching or caching of data (agents need to access and reason about their belief state): this is done by the runtime mechanism of the language. We do not know in advance which available agent is active (random access), but we might be able to learn so during the simulation and thereby optimise the caching mechanism. This is the reason why we are using Lua in the way explained in the next section.

**Modelling:** As of now, no generally accepted meta-model for multiagent-based simulations exists. We would like to distinguish between essential concepts and implementation details. What are the agents in the simulation? Which agent features are important?

So the main problem we are tackling is the following: How can we develop a scalable simulation environment, where the individual agents can be suitably programmed and where one can abstract away from specific features? We would like to reason about the macro view (usually supported by dedicated simulation tools) as well as zooming into the micro view when needed. The overhead for supporting the microview should not challenge overall system scalability:
(1) If no agents are needed (no micro-view), the performance of MASeRaTi should be as close to the legacy code (professional simulation tools) as possible.

(2) If no legacy code at all is used, MASeRaTi should still perform better or at least comparable to most of the existing agent platforms (and it should have similar functionality).

Due to general considerations (Amdahl’s law[18]) and the fact that not all processes will be parallelizable, it is not possible to achieve (1) perfectly (no agents: performance of MASeRaTi = performance of legacy code).

In addition to a scalable platform we also provide a meta-model for multi-agent-based simulations (MABS) and address the third challenge. However, the focus in this paper is on the first two challenges. The meta-model serves as a general starting point for the development of a MABS and ensures a certain structure of a simulation that is needed by the underlying platform in order to facilitate scalability. We have chosen Lua mainly because of its efficiency. It allows both object-orientation and functional programming styles and is implemented in native C. For details we refer to Section 3.2.

To conclude, we formulate the following basic requirements for MASeRaTi: (1) the support of a macro and micro view of a simulation, (2) a scalable and efficient infrastructure, and (3) a multiagent-based simulation modelling framework that also supports non-agent components.

3 Overview of the system

The overall architecture of our framework is inspired by concepts from game developing. The state of the art in developing massively multiplayer online role-playing games (MMORPG) consists in using a client-server architecture where the clients are synchronised during game play[10] via a messaging system. Well-known games include Blizzard’s World of Warcraft (WoW) or EA’s SimCity 2013, which supports multiplayer gaming with an “agent-based definition” in its own Glassbox engine(http://andrewwillmott.com/talks/inside-glassbox).

While a game architecture is a good starting point for our purposes, we cannot create a server system with hundreds of nodes, which is powerful enough to handle a MMORPG system. For developing purposes we also need a single node-based system, which can run on a small (desktop) node. After the developing process the source codes must then be transferable to a HPC system.

Our underlying meta-model uses the well established concept of a BDI-agent[28,31] in a variant inspired by the agent programming language Jason[7] combined with the idea of an entity[3] that evolved out of the experiences gathered in the MAPC. Our agent model connects agents to these entities in the simulation world. Agents consist of a body and a mind: While the mind (being responsible for the deliberation cycle, the mental state etc.) does not have to be physically grounded, the entity has to be located in an area of the simulation. Thus, an entity is an object with attributes that an agent can control and that

79
might be influenced by the actions of other agents or the overall simulation. Intuitively, an agent can be viewed as a puppet master that directs one (or more) entities. For all other objects in the simulation world, we use the concept of artifacts [6]. We also provide a basic notion of a computational norm that can be used by the simulation designer to steer the agents’ behaviour. Additionally, all objects can be grouped by using ObjectGroups. See Section 3.3 for details.

3.1 Architecture

Our system is composed of three layers (Fig. 2):

**Micro-kernel (MK):** The micro-kernel is a C++ based system, which defines the basic network parallelisation scheduling algorithms. The layer defines the underlying structure, e.g. plug-in and serialization interface, Prolog interface for the belief base and statistic accumulation interface. The layer describes a meta-model for a parallel simulation (Section 3.2).

**Agent-model layer (AML):** The agent-model layer (Section 3.3) defines the model of an agent-based simulation and is written in Lua (http://www.lua.org/) [20]. Within this layer the relation and entities of an agent-based simulation are created e.g. BDI-agent, world, artifacts, etc. Due to the multiple-paradigm definition of Lua pure object-oriented concepts are not supported directly. Technically speaking, Lua uses only simple data types and (meta-) tables. Fortunately, based on these concepts, we can create an object-oriented structure in Lua itself. This allows us to work in a uniform fashion with UML models at the AML and the scenario layer.

**Scenario layer (SL):** The third layer is the instantiation of the AML with a concrete scenario, e.g., a traffic setting or the MAPC cow scenario. It is represented by dotted boxes in Fig. 2 to emphasize the difference to the AML layer. Section 4 provides an example.
3.2 Micro-kernel

The micro-kernel describes the technical side of the system and is split up into two main structures (Fig. 3(b)). The core part (below) defines the scheduler algorithms, the core and memory management, the network and operating system layers and the plug-in API within a Prolog interpreter. Above these core utilities the Lua interpreter (top) is defined and each class structure on the core can be bound to “Lua objects”. The Lua runtime is instantiated for each process once, so there is no elaborated bootstrapping.

The choice of Lua is affected by the scaling structure and the game developing viewpoint. Lua, a multi paradigm language, has been used for game development for many years ([25]). An advantage of Lua is the small size of its interpreter (around 100 kBytes) and the implementation in native C with the enhancement to append its own data structures into the runtime interpreter with the binding frameworks. The multiparadigm definition of Lua, especially object-oriented and functional [20], can help us to create a flexible metamodel for our simulation model. Lua can also be used with a just-in-time compiler.

The kernel defines basic data structures and algorithms (Fig. 3(a)):
Simulation: A global singleton simulation object, which stores all global operations in the simulation e.g. creating agents or artifacts. It defines the initialization of each simulation; the constructor of the Simulation object must create the world object, agent objects, etc.

Object: Defines the basic structure of each object within the simulation. All objects have got a UUID (Universally Unique Identifier), a statistical map for evaluating statistical object data, the (pre/post)tick methods to run the object and the running time data, which counts the CPU cycles during computation (for optimisation).

Prolog: An interface for using Prolog calls within the simulation.

Each class is derived from the Lua Binding class, so the objects will be mapped into the AML.

The mapping between the micro-kernel and the AML is defined using a language binding concept. The Lua interpreter is written in native C. Based on this structure, a C function can be “pushed” into the Lua runtime. The function will be stored into a global Lua table; the underlying C function is used with a script function call.

Our concept defines the micro-kernel in UML; instantiated C++ objects are mapped into the runtime environment by a Lua binding framework (e.g. Lua Bridge (https://github.com/vinniefalco/LuaBridge) or Luabind (http://www.rasterbar.com/products/luabind.html)). Classes and objects in Lua are not completely separate things, as a class is a table with anonymous functions and properties. If a Lua script creates an object, it calls the constructor, which is defined by a meta-table function, the underlying C++ object will be also created and allocated on the heap. The destructor call to an object deterministically removes the Lua object and its corresponding C++ object. All C++ objects are heap allocated and encapsulated by a “smart pointer”, as this avoids memory leaks. This concept allows consistent binding between the different programming languages and the layer architecture.

Each Object comes from the Communication interface, which allows an object to send any structured data to another object. The central Object inherits to three subclasses. This structures necessary for creating a distributed and scalable platform with optimisation possibility:

Synchronised Object: An object of this type is synchronised over all instances of the micro-kernel (thread and core synchronised). It exists also over all instances and needs a blocking communication. In the agent programming paradigm the world must be synchronised.

Non-Synchronised Object: This object exists only on one instance of the micro-kernel and can be transferred between different instances of the micro kernel. It should be used for agents and norms, because the evaluation is independent from other objects. Using the “execution time” of the tick (time complexity), we can group such objects together.

Data-Type: This object represents a data structure, e.g. a multigraph for the traffic scenario with routing algorithms (Dijkstra, $A^*$ and $D^*$). The data
types will be pushed into the micro-kernel with the plug-in API. The Access-Type creates the connection to the storing devices.

Synchronised and non-synchronized objects are implemented via Boost.MPI structure, and the Access-Type defines the interface to a database or the filesystem for storing / loading object data. The access via the data interface will be defined by the Boost.Serialization library, so we can use a generic interface. Based on the Data-Type we can use the defined plug-in API for math datatypes (Fig. 4), which allows to create a (multi-) graph interface for our traffic scenario, based on Boost-Graph. This enables us to use a differential equation solver like OdeInt(\url{http://www.odeint.com/}) to simulate the macroscopic view in the simulation (e.g. a highway traffic model can be simulated with a differential equation while employing a microscopic agent-based view for an urban traffic area. The “glue” between these two types can be defined by a “sink / source data-type”. A plug-in is defined in a two-layer structure. The plug-in is written in C++ and based on the Lua binding structure mapped into the higher layers. The plug-in interface is based on a native C implementation to avoid problems with name managing in the compiler and linker definition. Plug-ins are stored in a dynamic link library; they are loaded upon start of the kernel.

3.3 Agent-model Layer

The agent-model layer (AML) (depicted in 5) defines a meta-model of an agent-based simulation. It provides the basic structure and serves as a starting point for an implementation. We start by explaining the structure, followed by the overall system behaviour; we end with a general description of the development process. Realization details (pseudo code) can be found in the appendix of [1].

Structure The structure of the meta-model is heavily influenced by the goal of creating a simulation which can be distributed over several nodes or cores. In such a multiagent simulation, the developer has to decide for each object whether it has to be present on every single core or whether it can exist independent of the other objects (we aim for the latter). These two options lead to two approaches: (1) the invocation of functions in the same simulation step (the objects being
Fig. 5. Agent-model layer: UML class diagram

on the same core), or, (2) the sending of messages (objects are not on the same core) after a simulation step. Since the first approach has the drawback to be forced to synchronise all objects, we choose the latter.

The goal of the AML is to simplify the development of multiagent simulations by defining those objects that have to be synchronised and those that run independently. A developer can easily modify the AML to her needs, in particular to redefine the synchronicity of objects.

Figure 5 illustrates the structure of the AML. Mainly, a simulation consists of a singleton Simulation, the non-synchronised object types Agent, Norm, and the synchronised classes Area, Artifact, ObjectGroup. While for the Simulation only one instance is allowed, the other objects can be instantiated several times. All instantiated objects are being executed in a step based fashion and therefore implement a tick method.

Simulation: The simulation class in the AML is the Lua-based counterpart to the simulation class in the MK. It is responsible for the creation, initialisation and deletion of objects, thus it is in full control over the simulation.

Agent: As we aim to simulate as many agents as possible we have to ensure that this part of the model can run independent of the rest. Therefore we define two kinds of agents as non-synchronised objects: a generic agent based on [31] and a more sophisticated BDI agent [28] inspired by Jason [7]. The agent interacts with the environment through entities [3]. In general an agent can have random access to the simulation world, so we can only encapsulate some parts of the agent, namely the internal actions and functions while the effects on the environment have to be synchronised. That is the reason for separating the agent into two parts: the mind (the agent) and the body (the entity). Thus, the generic agent has three methods that are invoked in that order: (1) perceive, (2) think, and (3) act. Inside these methods, we can call the methods of the entity directly while communication between objects has to be realised over a synchronised object (for instance with the means of an artifact). The agent developer has to explicitly specify the variables that have to be synchronised.

BDI Agent: The BDI agent is more sophisticated and consists of a Belief Base representing the current world view, a set of Events describing changes
in the mental state, a set of plans \textit{Plans}, and a set of \textit{Intentions} describing the currently executed plans. Fig. 6 shows an overview of the agent cycle.

For each \textit{tick}, the agent first perceives the environment, and checks for new messages. Based on this information, the belief base gets updated and an event for each update is generated. From the set of events one particular event is selected and a plan that matches this event will be chosen and instantiated. During a simulation run this might result in multiple instantiated plans at the same time and allows the agent to pursue more than one goal in parallel. Being a BDI agent it can only execute one action at a time, but several internal actions per simulation \textit{tick}. The next method selects the next action of an instantiated plan (i.e. the next action of an intention). In contrast to Jason, the agent cycle does not stop here if it was an internal action or a message, i.e., an action that does not affect the environment. Thus, the agent selects the next event (if possible) or next intention (if possible) until it reaches a global timeout (set by the simulation) or an external action is executed that forces a synchronisation, or if the set of events and intentions are both empty. Again, the agent developer has to explicitly tell the simulation platform the variables that have to be synchronised.

\textbf{Artifact:} For all passive objects of a simulation we use the \textit{artifact} methodology defined in \cite{6}. Basically, each artifact has a \texttt{type} and a \texttt{manual} in Prolog (a description of the possible actions associated with it) and a \texttt{use} method that allows an agent to execute a particular action. Due to the generality of this approach the developer decides whether the actions are known by the agents beforehand or not. Additionally, since the artifact is defined as a synchronous object, one can consider a derivation of this object that implements the actions as methods and allows for direct method invocation.

85
Area: So far, we defined the main actors of a simulation but how are they connected among each other? An artifact does not have to be located inside a real simulation, i.e., it does not need a physical position (in contrast, most objects do need one). Therefore, we define an area as a logical or physical space (similar to the term locality introduced by [19]). There can be several areas, subareas, and overlapping areas. In the general case, agents have random access to the environment, so the areas have to be synchronised over all cores of the simulation platform. In some circumstances, however, it is reasonable to create a new class inheriting all properties from the non-synchronised object. Within an area, we define some basic data structures and algorithms for path finding, etc. The most important issue, the connection of the non-synchronised agents with the synchronised areas is realised by the use of entities. Agents perceive the environment and execute actions by using the entities’ sensors and effectors.

Entity: An entity can be seen as the physical body of an agent located inside an area. An agent can register to it, get the sensor data, and execute actions that possibly change the environment. The entity has some effectors and sensors that are easily replaceable by the simulation developer. Since such an entity represents the physical body of an agent and is meant to connect an agent with the environment it has to be synchronised over all cores.

Institution & Norm: An institution is an object that checks for norm violations and compliance. More precisely, it operates as a monitor and is also responsible for sanctioning. But a developer can also decide to separate these two tasks. For the future, we are planning to focus only on three kinds of norms: obligations, permissions, and prohibitions. Additionally, we will only consider exogenous norms (events that occur in at least one area) and not rules that affect the agent’s mind, plans etc. Due to the non-synchronisation, the agent developer has to tell the simulation platform the variables that have to be synchronised.

ObjectGroup: Finally, an ObjectGroup – as the name implies – defines a group of objects. It can be used to group agents, artifacts or other objects. Method calls on an ObjectGroup are forwarded to all group members, i.e., with a single method call, all corresponding methods (with the same type signature) of the group members are invoked. In order to reduce overhead and to avoid circular dependencies we only allow a flat list of members at the moment. However, if a hierarchy is needed, it can be easily implemented.

Agent-model layer behaviour: So how does the overall behaviour look like? Initially the simulation object creates a number of agents, areas, object groups, norms, etc., and changes the global properties in the three phases: preTick, tick, and postTick. It can delete and create new agents during runtime. However, if the simulation developer decides to allow an agent to create another agent, this is consistent with the meta-model. The agent cycles are executed in each tick method, also the artifacts’, norms’ and areas’ main procedures are executed in this phase. The preTick is most often used as a preparation phase and the postTick phase is used for cleaning up.
This section contains some heavy technical machinery and describes even some low level features that are usually not mentioned. However, our main aim is to ensure scalability in an agent-based simulation system. In order to achieve that, we came up with some ideas (using Lua and how to combine it with BDI-like agents) that can only be understood and appreciated on the technical level that we have introduced in this section.

4 Evaluation: Cow scenario

Scalability is an important aim of the platform and therefore has to be evaluated early on. For that reason we chose the cow scenario from the MAPC as a first simulation that is realistic enough in the sense that it enforces the cooperation and coordination of agents. As it is already implemented for the MASSim platform, it can easily serve as a first benchmark.

In addition, we can test the suitability of the proposed meta-model and test a first implementation. Furthermore, the cow scenario contains already some elements of more complex scenarios like the traffic simulation.

The cow scenario was used in MAPC from 2008 to 2010. The task for the agents is to herd cows to a corral. The simulated environment contains two corrals – one for each team – which serve as locations where cows should be directed to. It also contains fences that can be opened using switches. Agents only have a local view of their environment and can therefore only perceive the contents of the cells in a fixed vicinity around them. A screenshot of the visualisation as well as a short description of the scenario properties are depicted in Fig. 7. For a detailed description we refer to [4]. Using the proposed meta-model AML we can now implement the cow scenario in the following way.

Fig. 8 shows how we derived the cow scenario classes from appropriate superclasses of the agent-model layer. The grid of the environment is implemented as an Area. Obstacles are defined by a matrix that blocks certain cells. The two corrals are subareas located inside the main area. Fences will become Artifacts. Similarly, we define a switch as an artifact that controls and changes the state (opened or closed) of a fence when getting activated. The cows are realised by a reactive agent that perceives the local environment and reacts upon it. For such a reactive agent the basic Agent definition together with an entity representing the cow are sufficient, while for the cowboy agents we need a more complex behaviour that facilitates coordination and cooperation. For this reason we use the BDIAgent (recall Fig. 5) class and create an entity for each cowboy agent. Furthermore, for each entity we create a simple MoveEffector that can be used by the entities to alter their position and a ProximitySensor providing the entities with their percepts. Additionally, we have to define the two teams by using the notion of an ObjectGroup. Finally, the simulation creates all agents and entities, assigns them to the two teams and creates the simulation world.

\[\text{Please note, that this is ongoing work. The corresponding Lua code can be found in the appendix of [1].}\]
Fig. 7. The environment is a grid-like world. Agents (red (at top) and blue (at the bottom) circles) are steered by the participants and can move from one cell to an adjacent cell. Obstacles (green circles) block cells. Cows (brown circles) are steered by a flocking algorithm. Cows tend to form herds on free areas, keeping the distance to obstacles. If an agent approaches, cows get frightened and flee. Fences (x-shapes) can be opened by letting an agent stand on a reachable cell adjacent to the button (yellow rectangles). An agent cannot open a fence and then definitely go through it. Instead it needs help from an ally. Cows have to be pushed into the corrals (red and blue rectangles).

To conclude, this preliminary evaluation shows that it is possible to express each aspect of the scenario using the predefined classes without the need to derive further ones from the synchronised or non-synchronised objects. (Nonetheless, doing so still remains a possibility). Regarding the suitability of Lua, it is an extremely flexible language that comes at the cost of a certain degree of usability: any newcomer needs some time to master it. But even then, having appropriate tools and methodologies that support the modelling process is a necessity to ensure an improved workflow and reduced error-proneness.

5 Conclusion and outlook

In this paper, we described ongoing work towards a distributed runtime platform for multiagent simulation. The main contributions of this paper are: (1) an analysis of the state of the art in agent-based simulation platforms, leading to a
set of requirements to be imposed on a simulation platform, focusing on runtime scalability and efficient memory management; (2) the proposal of a novel architecture and design of the MASeRaTi simulation platform, bringing together a robust and highly efficient agent kernel (written in Lua) with a BDI agent interpreter including multiagent concepts such as communication and computational norms; and (3) an initial proof of concept realization featuring a simple application scenario.

The work presented in this paper provides the baseline for further research during which the MASeRaTi system will be extended and improved. Issues such as optimisation of the scheduler and the caching mechanisms sketched in the appendix of [1] will be explored in more detail. Also, systematic experimental evaluation will be carried out using more sophisticated and much larger traffic simulation scenarios. As the ATSim platform introduced in Section 2.1 can deal with a few thousand (vehicle) agents, we expect MASeRaTi to scale up to one million agents. By the time we prepare the version of this paper for the postproceedings, we shall have more information available with respect to evaluation methods, criteria, and metrics, including but not restricted to scalability. Aside, different communication technologies like BitTorrent([http://www.libtorrent.org/](http://www.libtorrent.org/)) for the inter-object communication will be investigated.

Given the three objectives in the abstract, our focus in this paper has been on the first two: scalability and efficient memory management, whereas we only touched the third, modelling. Here, one avenue of research is to develop appropriate modelling tools to support the MASeRaTi architecture. Finally, methodologies for simulation development will be explored, starting from established methodologies such as GAIA, Tropos, or ASPECS.

References


Security Games in the Field: Deployments on a Transit System

Francesco M. Delle Fave, Matthew Brown, Chao Zhang, Eric Shieh, Albert X. Jiang, Heather Rosoff, Milind Tambe, and J. P. Sullivan

University of Southern California
CA 90049, Los Angeles, USA
http://teamcore.usc.edu/default.html
{dellefav,matthew.a.brown,zhan661,eshieh,jiangx,rosoff,tambe}@usc.edu

*Los Angeles County Sheriff’s Department
CA 91754, Los Angeles, USA
jpsulliv@lasd.org

Abstract. This paper proposes the Multi-Operation Patrol Scheduling System (MOPSS), a new system to generate patrols for transit system. MOPSS is based on five contributions. First, MOPSS is the first system to use three fundamentally different adversary models for the threats of fare evasion, terrorism and crime, generating three significantly different types of patrol schedule. Second, to handle uncertain interruptions in the execution of patrol schedules, MOPSS uses Markov decision processes (MDPs) in its scheduling. Third, MOPSS is the first system to account for joint activities between multiple resources, by employing the well known SMART security game model that tackles coordination between defender’s resources. Fourth, we are also the first to deploy a new Opportunistic Security Game model, where the adversary, a criminal, makes opportunistic decisions on when and where to commit crimes. Our fifth, and most important, contribution is the evaluation of MOPSS via real-world deployments, providing data from security games in the field.

Keywords: Security, Game-theory, Real-world deployment

1 Introduction

Research in Stackelberg security games has led to several real-world deployments to aid security at ports, airports and air transportation [14]. Such systems generate unpredictable security allocations (e.g., patrols and checkpoints), while carefully weighing each potential target, considering the scarcity of defender resources and the adversary’s response. In a Stackelberg security game, the defender (e.g., the security agency) commits to her strategy first, taking into account the attacker’s (e.g., a terrorist’s) ability to conduct surveillance before launching his attack [4, 5].
Among the different applications of security games, the problem of patrolling a transit system has gathered significant interest [7, 15]. Due to the large volume of people using it every day, a transit system is a key target for illegal activities such as fare evasion (FE), terrorism (CT) and crime (CR). The security of such a system then, poses a number of challenges. The first challenge is multi-operation patrolling. Whereas most previous work in security games has focused on single threats which could be represented with a single adversary model (e.g., PROTECT, TRUSTS and IRIS)[14], the comprehensive security of a transit system requires different specialized security responses against three threats (FE, CT and CR). The second challenge is execution uncertainty. Security resources are often interrupted during their patrols (e.g., to provide assistance or arrest a suspect). Thus, traditional patrol schedules are often difficult to complete. Current research in security games has proposed the use of Markov decision processes (MDPs) to plan patrols under uncertainty [7]. However, such schedules were not actually deployed in the field, therefore, their real effectiveness has yet to be verified in the real-world. The fourth challenge is accounting for joint activities. In CT patrolling, security resources, such as explosive detective canine (EK9) teams, often patrol train lines in cooperation with other resources. By doing so, their effectiveness is increased. Recently, [12] proposed a new security game model, SMART (Security games with Multiple coordinated Activities and Resources that are Time-dependent), that explicitly represents jointly coordinated activities between defender’s resources. [12]. Yet, similarly to the work of [7] discussed earlier, this framework has still not been deployed in the real-world. The fourth challenge is crime. Literature in criminology describes criminals as opportunistic decision makers [13]. At a specific location, they decide whether to commit a crime based on available opportunities and on the presence (or lack thereof) of security officers. Thus far, this type of adversary—less strategic in planning and more flexible in executing multiple attacks— has not been addressed in previous work, which has focused on strategic single shot attackers [14].

The fifth and most important challenge is that, despite earlier attempts [11], the actual evaluation of the deployed security games applications in the field is still a major open challenge. The reasons are twofold. First, previous applications focused on counter-terrorism, therefore controlled experiments against real adversaries in the field were not feasible. Second, the number of practical constraints related to real-world deployments limited the ability of researchers to conduct head-to-head comparisons.

To address these challenges, this paper introduces five major contributions. Our first contribution is MOPSS, the first Multi-Operation Patrol Scheduling System for patrolling a train line. MOPSS provides an important insight: the multiple threats (FE, CT and CR) in a transit system require such fundamentally different adversary models that they do not fit into state-of-the-art multi-objective or Bayesian security game models suggested earlier [16, 3]. Instead, in MOPSS each of the three threats is modeled as a separate game with its own adversary model. These three game formulations provide security for the
same transit system, require data from the same transit system as input, use
smart-phones to display the schedules and share several algorithmic insights.
Our second contribution addresses execution uncertainty. We deployed MDP-
based patrol schedules in the field, and used sampling-based cross-validation to
handle model uncertainty in such MDPs [6]. Similarly, our third contribution
is the deployment of coordinated schedules for CT patrolling. We incorporate
the framework in [12] to MOPSS, and use it to generate patrols for counter-
terrorism. Fourth, we address crime patrolling. Our contribution is the first ever
deployment of opportunistic security games (OSGs). We model criminals as op-
portunistic players who decide whether to commit a crime at a station based on
two factors, the presence of defender resources and the opportunities for crime
at the station.

Our fourth contribution is the real world evaluation of MOPSS. This eval-
uation constitutes the largest scale evaluation of security games in the field in
terms of duration and number of security officials deployed. As far as we know,
it constitutes the first evaluation of algorithmic game theory in the field at such
a scale. We carefully evaluated each component of MOPSS (FE, CT and CR)
by designing and running field experiments. In the context of fare evasion, we
ran a 21-day experiment, where we compared schedules generated using MOPSS
against competing schedules comprised of a random scheduler augmented with
officers providing real-time knowledge of the current situation. Our results show
that our schedules led to statistically significant improvements over the com-
peting schedules, despite the fact that the latter were improved with real-time
knowledge. For counter-terrorism, we organized a full-scale exercise (FSE), in
which 80 security officers (divided into 14 teams) patrolled 10 stations of a
metro line for 12 hours. The purpose of the exercise was a head-to-head compar-
ison of the MOPSS game-theoretic scheduler against humans. The comparison
was in terms of the schedule generation process, as well as provide a thorough
evaluation of the performance of both schedules as conducted by a number of
security experts. Our results show that MOPSS game-theoretic schedules were
able to perform at least equivalently to (and in fact better than those) generated
by human schedulers. Finally, we ran a two-day proof-of-concept experiment on
crime where two teams of officers patrolled 14 stations of a train line for two
hours. Our results validate our OSG model in the real world, thus showing its
potential to combat crime.

2 Transit Line Patrolling

The Los Angeles Sheriff’s Department (LASD), the security agency responsible
for the security of the Los Angeles Metro System (LA Metro), requested a multi-
operation patrol scheduling system to improve and facilitate the comprehensive
security of each train line. This system should generate randomized schedules
for three different operations each addressing a fundamentally different threat:
Fare evasion patrols (FE): This type of patrol covers both the trains and
the stations of a train line. The purpose is to capture as many fare evaders as
possible to improve the perception of order within an area. Thus, this type of patrolling should favor the locations with a large volume of riders because it would lead to a large number of fare evaders caught.

**Counter-terrorism patrols (CT):** This type of patrol covers the stations of a train line. Each station concentrates a large number of people at a specific place and time. In addition, in Los Angeles, several stations are located within key economic and cultural areas of the city (e.g., tourist locations, business and financial districts). Thus, the effects on society of any successful attack on the metro system would be catastrophic. Terrorists are then strategic adversaries who carefully study the weaknesses of a train line before committing an attack. To optimize security, this type of patrol should cover the different stations while favoring the stations either with large passenger volume and/or located in key areas.

**Crime:** This type of patrol covers the stations of a train line. Crimes can be of different types including robbery, assaults and drug dealing. Each of this crimes is a symptom that the train line’s security is not sufficient. In addition, criminals behave differently than terrorists or fare evaders. They are opportunistic decision makers, they randomly traverse a train line, moving from station to station, seeking opportunities for crime (e.g., riders with smart-phones) [1, 13]. The key purpose of crime patrolling is then to patrol each of these stations, while favoring the stations representing “hot-spots” for crime (i.e., the most attractive stations from a criminal’s perspective).

Given the three operations defined above, the LASD computes patrol schedules, manually, on a daily basis. This task, however, introduces a significant cognitive burden for the human expert schedulers. Thus, to generate more effective schedules in a timely fashion, we introduce MOPSS, described in the next section.

### 3 MOPSS

MOPSS addresses the global security of a transit system. Hence, it presents two key advantages for the LASD. First, it can be used to generate specialized patrols for substantially different threats and second it concentrates all the information relevant to the comprehensive security of each transit line (e.g., crime and ridership statistics). MOPSS is comprised of a centralized planner and a smart-phone application (shown as a demonstration in [9]). The system is shown in Figure 2. The core of MOPSS consists of the three game modules. Each module generates patrols for one operation (FE, CT or CR). Each operation deals with a fundamentally different adversary model (fare evaders, terrorists or criminals), therefore each operation is modeled as a different two-player security game (the defender’s resources represent the security officers). Each module takes as input the information about the requested patrol (i.e., the number of officers, the starting time and the duration) and connects to a database to get the data necessary to build the security game model. Each game is cast as an optimization problem and sent to the SOLVER which contains three algorithms, one for each game.
Fig. 1. MOPSS
Fig. 2. The MOPSS system

[12, 17, 7]. Once the game is solved, the defender’s mixed strategy is sent to the SAMPLER to produce the schedule which is uploaded into the application.

3.1 Fare Evasion Module

This module aims to generate the defender’s (i.e., security officers’) mixed strategies against fare evaders [7]. The idea is to use such strategy to derive patrol schedules that randomly favor the trains and the stations with a large volume of riders. Fare evaders are modeled as daily riders based on statistics.

The key requirement of fare evasion patrolling is to be able to address execution uncertainty. To do so, in the FE module, the mixed strategy for each defender resource $i$ is determined by an MDP denoted by a tuple $\langle S_i, A_i, T_i, R_i \rangle$ where: (i) $S_i$ is a finite set of states ($s_i = (l, \tau)$ where $l$ is a train or a station and $\tau$ is the time step); (ii) $A_i$ is a set of two actions: perform a train or a station check (equivalently do a train or a station check) and (iii) $T_i(s_i, a_i, s_i')$ is the transition probability which can model execution uncertainty such as an officer being delayed while trying to conduct a fare check (e.g., due to arrests) and (iv) $R_i$ is the immediate reward for transition $(s_i, a_i, s_i')$. Although this reward could potentially model more complex domains, it is unrelated to the game-theoretic payoffs, and is not considered in the remainder of this work.

The FE game is then represented as a two player Bayesian zero-sum game (see [7] for the definition of the linear program). Given a resource $i$ and rider $\lambda \in A$ (i.e., defined by their daily itinerary in the train line), the objective is to maximize the expected utility of the defender, defined as $\max \sum_{\lambda \in A} p_{\lambda} u_{\lambda}$ where each utility $u_{\lambda}$ is the defender’s payoff against passenger type $\lambda$, which has a prior $p_{\lambda}$ calculated using ridership statistics (calculated using ridership statistics). Each $u_{\lambda}$ is calculated by the constraint $u_{\lambda} \leq x^T U_{\lambda} e_{\alpha} \forall \lambda, \alpha$ where each utility
$U_\lambda(s_i, a_i, s'_i, \alpha)$ represents the payoff that resource $i$ will get for executing action $a_i$ in state $s_i$ and ending up in $s'_i$, while the attacker plays action $\alpha$ (defined by the base vector $e_\alpha$) and $x$ is the marginal probability that the resource will actually go from $s_i$ to $s'_i$. In other words, $x$ represents the probability that the officer will overlap with a fare evader of type $\lambda$ playing action $\alpha$.

The optimization problem defined above is used by the SOLVER module to produce a mixed strategy represented as a Markov policy $\pi_i$. The SAMPLER then generates a single MDP patrol schedule that is loaded onto the handheld smartphone. An example of such a schedule is shown in Figure 3(a). The figure shows the schedule as it is visualized by the mobile application. The schedule contains two actions: train checks and station checks. Given that there is now a full MDP policy on the smartphone, a schedule can be updated whenever a security officer is delayed, by pushing the “>” button shown in Figure 3(a).

We next turn to instantiating the parameters in this game model for deployment. Fortunately, given fixed train fares and penalties for fare evasion, populating the payoff matrices is straightforward. Furthermore, via observations, we were able to set the transition function $T_i$. However, the delay length, whenever an office was interrupted, seemed to vary significantly, and modeling this variability became important. A continuous-time MDP or modeling multiple fine-grained delays are both extremely expensive. As a practical compromise we use a model considering a single delay whose value is chosen via cross-validation [6]. First, we randomly generate $N$ MDPs, each of which assumes that resource $i$ can experience delays of five different lengths: 6, 12, 18, 24 and 30 minutes (any delay longer than 30 minutes is considered to be beyond repair and a new schedule is generated). Second, we solve each MDP and obtain $N$ Markov policies $\pi_{ik}$ corresponding to each $MDP^k$ which we cross validate by running 100000 Monte Carlo simulations. In each simulation, we sample one strategy for the defender and calculate the resulting expected utility against all $N$ MDPs. Finally, we pick the policy that maximizes the minimum. If the officer gets into a state not directly represented in the MDP, we pick the next available state at their current action.

Fig. 3. Three schedules for each threat of a transit system
3.2 Counter Terrorism Module

The counter-terrorism module aims to generate a defender mixed strategy that can be used to produce schedules that deter terrorists from attacking the stations of a train line [12]. Since stations are often composed of multiple levels, these schedules should then randomly patrol each of these stations while taking the levels into account and while favoring the most important stations. Terrorists are modeled as strategic adversaries who carefully observe the security of a train line before executing an attack.

The key requirement of CT patrolling is to represent joint activities. We achieve this by incorporating the SMART problem framework defined in [12] in the CT component of MOPSS. A SMART problem is a Security Game [8] such that each target \( t \in T \) is assigned a reward \( U_d^t(t) \) and a penalty \( U_a^t(t) \) if \( t \) is covered and uncovered by a defender’s resource. Similarly, each target is assigned a reward \( U_d^t(t) \) and a penalty \( U_a^t(t) \) for the attacker. The defender has a set of \( R \) resources. Each resource chooses an activity from the set \( \mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_K\} \) for each target \( t \in T \). Each resource \( r \in R \) is assigned a graph \( G_r = (T, E_r) \), where the set of vertices \( T \) represents the set of targets to patrol and the set of edges \( E_r \) represents the connectivity between such targets. Each edge \( e \in E_r \) is assigned a time value \( \tau(e) \) representing the time that it takes to one defender resource \( r \) to traverse \( e \).

The attacker’s pure strategy space is the set of all targets, \( T \). A pure strategy for the defender is a set of routes, one route \( X_t \) for each resource. Each route is defined as a sequence of activities \( \alpha \), conducted at a specific target \( t \) with specific duration \( \gamma \). Joint activities are then represented when there exists two routes \( X_i \) and \( X_j \) such that \( t_i = t_j \) and \( |\gamma_i - \gamma_j| \leq W \), i.e. when two activities of two different resources overlap in space and time (within a time window \( W \)). For each activity \( \alpha_i \), \( \text{eff}(\alpha_i) \) represents the individual effectiveness of the activity \( \alpha_i \), which ranges from 0% to 100%, and measures the probability that the defender will be able to successfully prevent an attack on target \( t \). The effectiveness of the joint activity \( \langle \alpha_i, \alpha_j \rangle \) is defined as \( \text{eff}(\alpha_i, \alpha_j) \).

Given these parameters, the expected utilities \( U_d(P_t, t) \) and \( U_a(P_t, t) \) for both players, when the defender is conducting pure strategy \( P_t \) (defined as a joint pure strategy for multiple defender resources), and when the attacker chooses to attack target \( t \) is given as follows:

\[
\omega_t(P_t) = \max_{\{t, \alpha, \gamma\} \in P_t} \{\text{eff}(\alpha), \text{eff}(\alpha_1, \alpha_m)\} \tag{1}
\]

\[
U_d(P_t, t) = \omega_t(P_t)U_d^t(t) + (1 - \omega_t(P_t))U_a^t(t) \tag{2}
\]

\[
U_a(P_t, t) = \omega_t(P_t)U_d^t(t) + (1 - \omega_t(P_t))U_a^t(t) \tag{3}
\]

Here \( \omega_t(P_t) \) defined in Equation (1) represents the effective coverage of the defender on target \( t \) when executing pure strategy \( P_t \).

To solve this problem, we use SMART\(_H\), a branch-and-price, heuristic approach, which we incorporate in the SOLVER component of MOPSS. SMART\(_H\) is based on a branch-and-price framework, it constructs a branch-and-bound
tree, where for each leaf of the tree, the attacker’s target is fixed to a different $t'$. Due to the exponential number of defender pure strategies, the best defender mixed strategy is determined using column generation, which is composed of a master and slave procedure, where the slave iteratively adds a new column (defender strategy) to the master. The objective of the pricing component is to find the best defender mixed strategy $x$ at that leaf, such that the best response of the attacker to $x$ is to attack target $t'$. The structure of the algorithm is illustrated in Figure 4. In the figure, the master solves the non-zero-sum game to get a defender mixed strategy over a small subset of joint patrol pure strategies. After solving the master problem, the duals are retrieved and used as inputs for the slave. The purpose of the slave is to generate a pure strategy which is then added to the master and the entire process is iterated until the optimal solution is found.

![Fig. 4. The column generation algorithm](image)

An example counter-terrorism schedule, as visualized by the mobile application, is shown in Figure 3(b). The schedule describes two actions, observe (patrol a station) and transit (go to a station) each with a specific time and duration. The key challenge to deploy CT schedules is to define an accurate SMART problem instance to accurately encompass the real-world problem. To achieve this, we had to define three types of features. First, we had to define the payoffs of the game. We defined the payoffs for each target (32 in total) in discussions with security experts from the LASD. Each set of payoffs for each station was based on the number of people using the station every day and by the economic impact that losing this station would have on the city. The different levels of a single station had slightly different payoffs which were based on the number of persons present at each specific level of the station every weekday. Second, we had to define the defender different resources, i.e., the team participating to the experiment. These are defined as follows:

- **T teams**: High Visibility Uniformed Patrol teams.
- **HVWT teams**: High Visibility Weapon teams.
- **VIPR teams**: Visible Intermodal Interdiction teams.
- **CRM teams**: Crisis Response Motor teams.
- **EK 9 teams**: Explosive Detection K9 (canine) teams.

Third, we had to define the single and joint effectiveness for both the observe and transit actions. All Transit actions were given a 0 effectiveness, since moving

---

1 We are not able to reveal the value of these payoffs due to an agreement with the LASD.
from one station to another (i.e., riding the trains or taking the car) will not have any effect on the security of the stations. Most teams were assigned the same positive individual effectiveness of 0.7, except the VIPR team which has a greater individual effectiveness because it is composed of officers from multiple agencies carrying heavy weapons. VIPR teams, T-teams and HVWT teams typically work alone. Hence, to define their effectiveness values, their individual effectiveness is positive while their joint effectiveness is null (any joint effectiveness value below 0.7 would induce the same type of behavior, but we chose 0 since it is a clear indicator of the type of behavior that we want to obtain). The CRM teams are assigned a joint effectiveness greater than their individual effectiveness because they can perform all type of activities, but, typically, they prefer joint over individual activities. In contrast, EK9 teams typically work only in cooperation with other teams, therefore they are assigned a null individual effectiveness and a positive joint effectiveness of 0.75.

3.3 Crime Module

The crime module aims to generate a defender mixed strategy to prevent crime on a train line. The idea is to generate schedules that take criminal behavior into account and attempt to predict the stations that are more likely to be affected by crime. Crime statistics are used to characterize the behavior of criminals and the attractiveness that they attribute to each station of the train line. The key difference with the previous modules is that criminals behave differently than fare evaders and terrorists. They are less strategic in planning crimes and more flexible in committing them than is assumed in a Stackelberg game. They opportunistically and repeatedly seek targets and react to real-time information at execution-time, rather than strategically planning their crimes in advance.

Crime schedules are computed using an OSG [17]. An OSG is similar to a Stackelberg game in that the defender commits to a patrol strategy first, after which the criminal chooses the station(s) to attack given their belief about the defender’s deployment. In an OSG, the defender’s actions are computed using a Markov chain, which assigns probabilities for how the defender should move through the train line. The criminal’s behavior is defined by a quantal-biased random walk, i.e., the next station to visit for potentially committing a crime is determined according to the quantal response model [14]. This model takes as input information the attractiveness $\text{Att}(i)$ of each station $i$ and the criminal’s belief about the defender’s strategy which is updated using real-time observations. Station attractiveness is a measure based on crime statistics about the availability of opportunities for committing crime as well as how likely criminals are to seize such opportunities. The behavior models for both the defender and the criminal are combined to form a Markov chain with transition matrix $T_s$, which along with the rewards to the defender, define an OSG that can be solved to generate an optimal defender strategy. To solve an OSG, we iteratively calculate the defender expected utility $V_d$ over all the possible states of the Markov
chain for a number of crime opportunities $k$ as follows:

$$Obj = \lim_{K \to \infty} \sum_{k=0}^{K} V_d(k+1)$$

$$= r_d \cdot (I - (1 - \alpha)T_s)^{-1}X_1,$$  \hspace{1cm} (4)

where $R_d$ is a vector defining the utility of each state of the Markov chain in terms of the payoff $u_d$ for the defender and the attractiveness $Att(i)$; $I$ is the identity matrix; $\alpha$ is the probability of leaving the train line after an attack and $X_1$ is the initial coverage probability over all the possible states of the Markov chain. By maximizing $Obj$ (i.e., minimizing the total amount of crime in the metro), we obtain a transition matrix $T_s^*$. This matrix is then used to compute the defender’s Markov strategy $\pi$.

The maximization of Equation 4 is a nonlinear optimization problem. Therefore, to scale up to the number of states necessary to represent a real train line we use the Compact OPportunistic security game State algorithm (COPS) [17] in the SOLVER module. COPS returns a number of coverage probabilities for the different stations of the train line. These are then sent to the SAMPLER module which generates a schedule. An example of a schedule for crime patrolling is shown in Figure 3(c). It describes three actions, go north (i.e., take the next northbound train), go south (i.e., take the next southbound train) and stay (i.e., patrol a specific station).

To deploy crime schedules, two key challenges had to be addressed. The first challenge deals with defining of the attractiveness parameter. In our work, we define the attractiveness $Att(i)$ of station $i$ following the statistical model presented in [13]. Formally, $Att(i) = 1 - \exp^{-aN(i)}$, where $N(i)$ is the number of past crimes at station $i$ (based on actual crime statistics received from the LASD) and $a$ is a weighting coefficient. The second challenge is the parameterization of the criminal behavior model, which consists of defining the quantal-biased random walk. In our crime tests (Section 4.3), we defined the criminal behavior in collaboration with both security agencies and criminologists.

### 4 Real World Evaluation

In collaboration with the Los Angeles Sheriff’s Department (LASD), we designed three types of real world tests, one for each of the three operations defined in Section 2. Each of these tests allows us to evaluate different aspects of game-theoretic patrolling. This evaluation introduces the following novelties: (i) in fare evasion, we present the first real world deployment of game-theoretic schedules and analyze their performance against real adversaries (fare evaders); similarly, (ii) in counter-terrorism, we present the first real world head-to-head comparison between game-theoretic and human generated schedules. Finally, (iii) in crime, we introduce the first deployment of OSGs. The crime tests provide the first real world data showing the benefits of game-theoretic scheduling when facing opportunistic attackers.
4.1 Fare Evasion Experiments

The purpose of the fare evasion experiments is to derive a quantitative analysis of the performance of MOPSS. Fare evasion patrols allow the officers to interact daily with the riders of the system. The outcome of these interactions can then be used to compare different types of schedules (e.g., number of fare evaders caught).

**Setup**: Our experiments ran for 21 weekdays, with each day consisting of a team of two security officers patrolling a line of 22 stations for at most 120 minutes (some days our tests ended early due to the officers being reassigned). The team was provided with one of two types of schedules: (i) MOPSS; or (ii) UR+HINT. UR+HINT used a uniform random approach to generate a schedule but allowed Human INTeLLigence to be used to augment this schedule in real time using situational awareness. UR+HINT was chosen for two reasons: first, it avoided human bias of being predictable, and second, it simultaneously allowed humans to use their information and intelligence to improve the uniform random schedule. The officers were not told which schedule they were using as not to bias their performance. Patrols were run during both the morning and the afternoon. MOPSS schedules were deployed for 11 days of testing while UR+HINT schedules were deployed for 10 days, resulting in 855 and 765 patrol minutes, respectively. We divided the data into 30 minute segments to ensure that we could use all the data collected, even when the patrols were shorter than 120 minutes.

The performance of each type of schedule was determined in two ways. First, we counted the number of passengers checked and the number of captures at the end of each patrol. The captures were defined as the sum of the number of warnings, citations, and arrests. Passengers without a valid ticket could be given a warning or cited for a violation on the discretion of the officer. Second, we measured the degree of satisfaction of the officers with respect to each type of schedule. To achieve this, during each patrol, we counted the number of times that the update function was used voluntarily. More specifically, each update was labeled as VOLUNTARY or INVOLUNTARY. INVOLUNTARY updates consisted of the officer requesting a new schedule because they were delayed (e.g., from issuing citations or arresting a suspect) and were unable to perform the next action on their schedule. VOLUNTARY updates consisted of the officer updating the current schedule because they did not like the current action. Officers were allowed to choose a new location that they considered more
fruitful for catching fare evaders and request a new schedule from there. Since too many VOLUNTARY deviations would essentially change the nature of the schedules, officers were allowed only two such updates during each patrol for both types of schedule. To ensure impartiality, we allowed officers to use VOLUNTARY and INVOLUNTARY updates for both MOPSS and UR+HINT, but as we will see below they used VOLUNTARY updates almost every day with the UR+HINT schedules (hence augmentation with human intelligence), but never in the MOPSS schedules.

Finally, notice that MOPSS game-theoretic schedules are essentially testing a maximin strategy. MOPSS computes a Stackelberg strategy, but in the short time that we had to perform tests, the passengers would not be able to first surveil and then adapt to the patrol schedules of the officers as would be assumed by a Stackelberg game. However, since our security game is defined to be zero sum, the resulting Stackelberg strategy is equivalent to a maximin strategy, which makes no assumption on the adversary’s surveillance of the defender’s mixed strategy. Thus, these experiments compare the benefit of using a maximin strategy against other (non-game-theoretic) approaches for generating patrol schedules.

**Results:** Our results are shown in Figure 5 and in Table 1. Figure 5 shows eight boxplots depicting the data that we collected during each patrol, using both MOPSS and UR+HINT schedules. Respectively, the four figures present data collected on captures (Figure 5(a)), warnings (Figure 5(b)), violations (Figure 5(c)), and passengers checked (Figure 5(d)) per 30 minutes of patrolling. For each boxplot, the top and bottom of the box represent the 75th and 25th percentiles, respectively, while the middle line indicates the median. The “+” data points indicate statistical outliers, while the whiskers show the most extreme non-outlier data points. Each of the four figures (captures, warnings, violations and passengers checked) shows that the data collected using MOPSS schedules had higher values than the data collected using UR+HINT schedules.

Table 1 shows the average number of captures, warnings, violations and passengers checked per 30 minutes of patrolling over the 21 days. Table 2 shows the number of days of patrol and the number of VOLUNTARY and INVOLUNTARY deviations. On average, MOPSS schedules led to 15.52 captures against 9.55 captures obtained using UR+HINT schedules. To confirm the statistical significance of these results, we ran a number of unpaired student t-tests ($p = 0.05$) and verified, for each metric, that the difference in the results was statistically significant. This is a key result, as MOPSS schedules were more effective despite officers VOLUNTARILY deviating from the UR+HINT schedules, and thus augmented the schedules with real time knowledge, on 8 out of the 10 days. Furthermore, the magnitude of the difference is practically significant: cumulatively over a period of days MOPSS would capture a much larger total number of fare evaders. In addition, the fact that the officers never deviated from MOPSS schedules, except for INVOLUNTARY delays, validates our MDP model and

---

2 MOPSS schedules also led to two arrests.
shows its usefulness in producing flexible patrol schedules that can be used in the real world.

<table>
<thead>
<tr>
<th># of Days</th>
<th># Captures</th>
<th># Warnings</th>
<th># Violations</th>
<th># Passengers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOPSS</td>
<td>11</td>
<td>15.52</td>
<td>10.42</td>
<td>5.03</td>
</tr>
<tr>
<td>UR+HINT</td>
<td>10</td>
<td>9.55</td>
<td>6.48</td>
<td>3.07</td>
</tr>
</tbody>
</table>

Table 1. Average, over each metric, of the results obtained in Figure 5

<table>
<thead>
<tr>
<th># VOLUNTARY deviations</th>
<th># INVOLUNTARY deviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOPSS</td>
<td>0</td>
</tr>
<tr>
<td>UR+HINT</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2. Number of voluntary and involuntary deviations for each type of schedule.

As discussed in Section 3.1, MOPSS schedules randomly favor locations with higher volumes of passengers. Our results confirm the accuracy of our model as both Figure 5(d) and Table 1 show that MOPSS schedules led the officers to check more passengers than UR+HINT schedules. This raises the question of whether a static type of schedule, which only deploys officers at the most crowded locations, would lead to similar or even better results than those obtained with MOPSS. Given the limited amount of time that we had to conduct our experiments, we were unable to compare MOPSS schedules against a static deployment. However, effective randomization was one of the main reasons for LASD to collaborate on these experiments – security agencies know that static schedules become predictable in the long term (see [14] discuss the benefits of randomization in detail). After a certain amount of time, the passengers would know where the officers are located and could exploit this information to avoid paying the fare.

4.2 Counter-Terrorism Experiment

The purpose of this experiment is to run a head-to-head comparison between MOPSS and a manual allocation, the standard methodology adopted by several security agencies. Security agencies refer to this type of experiment as a mass transit full scale exercise (FSE). A FSE is a training exercise where multiple security agencies analyze the way their resources cooperate to secure a specific area while simulating a critical scenario. This scenario typically describes a “high level” threat, e.g., intelligence reports confirming that a terrorist attack might take place in the Los Angeles Metro System. The FSE consists of simulating the
response to this threat, i.e., increasing the number of resources patrolling a train line on a daily basis to improve the quality of the security.

**Setup:** The FSE consisted of patrolling 10 stations of one train line of the LA Metro system for 12 hours. Each station on the train line is composed of three levels (street level, platform level and mezzanine) except station 1 which is composed of 5 levels (2 more platform levels). The exercise involved multiple security agencies, each participating with a number of resources. Overall, 80 security personnel were involved. These resources were divided into 14 teams, each with different abilities (see Section 3.2).

The exercise was divided into 3 different “sorties”, each consisting of three hours of patrolling and one hour of debriefing. Human-generated schedules were used during the first sortie while MOPSS schedules were used during the second and the third sorties. The first two sorties were used to run the head-to-head comparison. Hence, the sorties were ran under the same settings: the same number of officers had to cover the 10 stations for a cumulative time of 450 minutes. The two sorties were ran during off-peak times (9h00 to 12h00 and 13h00 to 16h00, respectively), hence the type and the number of riders of the train lines could be considered to be, approximately, the same. The purpose of Sortie 3 was to test whether the officers were capable of following MOPSS schedules for a longer period (900 minutes instead of 450) and during peak time. We found out that the officers were actually able to follow the schedules. Thus, since the purpose of this Sortie was unrelated to our comparison, we will focus on Sorties 1 and 2 in the remainder of this section. Each type of schedule was generated as follows:

MOPSS schedules: The schedules were generated by (i) instantiating a CT game using the specifics of the FSE discussed earlier; (ii) solving this problem instance using the SOLVER and (iii) sampling a pure strategy in the SAMPLER to generate the patrol schedule for each of the different resources involved. Specifically, we ran the SMART in the SOLVER component, considering 14 resources and 32 targets. The algorithm produced a mixed strategy which was then sampled to generate a pure strategy in the SAMPLER. This pure strategy contains a schedule for each resource.

Manual Schedules: The schedules were generated by human expert schedulers of the LASD. They were generated using a two-step process. First, each station was assigned a coverage duration of 45 minutes (i.e., \( \frac{1}{10} \)th of the time). The idea was to have the officers perform three observe actions at each station. Second, the human expert schedulers assigned teams to each station so that each station was covered for exactly 45 minutes. Joint team activities were used 6 times in six different stations. This simple two-step process was adopted to avoid the cognitive burden involved with leveraging the effectiveness of each team to cover the different stations individually or while coordinating with other teams. Despite its simplicity, this process was difficult for the human expert schedulers. It involved several discussions and required one entire day of work.
**Results:** We first analyze the type of schedules generated as a result of using either MOPSS or manual scheduling. Then, we evaluate the results obtained by deploying the schedules during Sorties 1 and 2 and measuring their performance in the real-world.

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MOPSS</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3. Count of Individual Activities

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOPSS</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4. Count of Joint Activities

The numbers of individual and joint activities for both the schedules generated during the FSE are shown in Tables 3 and 4. In both tables we can see that the number of individual (IA) and joint (JA) activities for both approaches are the same (IA: both 24; JA: both 6). All the joint activities in the MOPSS schedules are performed by CRM and EK9 teams, i.e., the teams with a positive joint effectiveness. This is similar to the behavior of the manual generated schedules, where joint activities are mostly performed by EK9 and CRM teams (once by the VIPR team). The remaining individual activities are performed by the T team, the HVWT team and the VIPR team.

There are two important differences between the two types of schedules. First, MOPSS sent the most effective VIPR team to the most important stations because its individual effectiveness is greater than the effectiveness of other teams. This was not seen in the human schedule. Second, the schedules generated using MOPSS assigned the different teams to cover all the different levels of the different stations, whereas manual schedules did not specify such levels. The reason for this is that human schedulers were not able to reach this level of detail and thus they preferred to leave the decision of which level to patrol to the teams once they were deployed. In addition, the effort required to generate the schedules using MOPSS was much lower than the effort required to generate manual schedules, which required one day of work due to its significant cognitive burden. Since typically such patrols would be conducted day-in and day-out for several days in high-threat periods, the savings of human effort achieved by game-theoretic schedulers are thus very significant.
Each type of security allocation (either manual or game-theoretic based on MOPSS) was evaluated by security experts. A team of security experts (SEs) was placed at each station for the entire length of the exercise. Their task was to observe and evaluate the officers’ patrolling activity during each sortie, and determine how their behavior was affecting the quality of the security within each station. In what follows, we report the conclusions of their analysis. The SEs did not know what type of schedules (so as to not bias their evaluation). To translate the observers’ observations into a comparable value, each observer was asked to fill out a questionnaire every 30 minutes. The objective was to define a number of key sentences that could help to qualify the way in which the security officers had been patrolling the station in the last 30 minutes. Each questionnaire contained 11 assertions about the level of security within the station. The assertions were defined in collaboration with a team of SEs from the LASD and with social scientists. Each SE had to determine his level of agreement with each assertion, which was defined in the integer interval \{0, 6\}, where 0 meant a strong disagreement, whereas 6 meant a strong agreement.

Figures 6(a) and 6(b) show the results that we obtained. Figure 6(a) shows the weighted average agreement obtained for each assertion calculated over all the stations (the average was calculated considering each station’s corresponding weight). Figure 6(b) shows the average agreement obtained for each station calculated over all the assertions. The error bars in both figures show the standard error of the mean calculated for each specific assertion (in Figure 6(a)) and station (in Figure 6(b)). As we can see the difference between some data points of the two approaches do not seem to be statistically significant. A student t-test confirmed this trend. This is expected, since we were only able to collect data for few hours of a single day. Nonetheless, we can still acquire some interesting information about the performance of game-theoretic schedules in the field, by analyzing the results that are statistically significant.

In Figure 6(a), we can see that MOPSS schedules seem to yield a higher level of agreement than manual schedules over all questions. As shown in the
figure, the difference is significant only for assertions $Q_1$, $Q_2$, $Q_8$ and $Q_9$. These four assertions correspond to very general statements about the security at each station which address the efficiency of the schedules, their ability to provide a strong feeling of safety and to allow the officers to patrol each area as much as needed.

Similarly, in Figure 6(b), we can see that the average agreement is higher for MOPSS schedules over Manual schedules for stations $S_1$, $S_2$, $S_3$, $S_4$, $S_8$, $S_9$ and $S_{10}$. Some of these stations ($S_1$, $S_8$ and $S_9$) are the ones assigned a higher set of payoffs, as discussed above. Hence, they correspond to the ones given a higher coverage by MOPSS.

These results indicate that game-theoretic schedules were evaluated as more effective than manual schedules. By analyzing the differences between the schedules, we can infer that this happened for two key reasons. First, as discussed earlier, manual schedules were generated by leaving the decision of which level of a station to patrol to each deployed team. The officers then, were not able to properly coordinate over the different levels to patrol and therefore they ended up patrolling the same levels. Second, MOPSS produced a schedule which more effectively scheduled the VIPR team, i.e., the team with the highest effectiveness (0.8) for covering each target. More specifically, the VIPR team patrolled all the most important stations at key levels. In contrast, manual schedules assigned the VIPR team, without accounting for its effectiveness. This made an impact on the security evaluators, which considered the game-theoretic allocation more effective than the manual allocation, because it was leveraging the abilities of the resources in a way that human experts could not achieve.

### 4.3 Crime Experiment

Our crime experiment was designed to be a proof-of-concept of MOPSS crime component. As discussed in Section 3.3, OSGs are a new framework to represent opportunistic adversaries. The purpose of our experiment is then to validate this new framework in the real world to ascertain its ability to generate effective schedules against crime. The experiment was organized as follows:

**Setup:** We ran tests for two days with each test consisting of a two hours patrol involving two teams of two security officers. Each team had to patrol seven stations of a particular LA Metro train line using schedules generated using MOPSS. MOPSS generated the schedules by converting crime statistics into a set of coverage probabilities for the different stations. Figure 7 shows such probabilities and correlates them to the crime statistics for each of the 14 stations to patrol. In the figure, the x-axis enumerates the 14 stations to patrol. The bar graphs (y-axis on the right) show, for each station, the total number of crimes that happened during 2012 and 2013. Finally, the line graph shows the different coverage probabilities calculated for each station (y-axis on the left). In the figure, the stations with a larger coverage probability (stations 5 to 10) are either the stations with a large number of crimes (stations 5 and 8) or the adjacent stations (Stations 6, 7, 9 and 10). The latter stations are given a large coverage probability because the OSG model anticipates the possibility
that criminals will choose stations 6, 7, 9 and 10 anticipating that stations 5 and 8 will be frequently patrolled by security officers [17]. Hence, these coverage probabilities show how game theory allows to build real world patrol schedules. 

**Results:** During the tests, the officers were able to write 5 citations and make 2 arrests. In general, they were able to understand and follow the schedule easily. Overall, these tests indicate that the CR module in MOPSS can produce effective schedules that would work in the real world.

![Fig. 7. Crime Statistics and Coverage Probabilities](image)

5 Lessons learned

The work presented in this paper is the result of a long term collaboration between university researchers and personnel from different security agencies including decision makers, planners and operators. To interact with such security agencies, we took inspiration from the lessons presented in [11]. We discussed the strengths and weaknesses of every aspect of MOPSS and emphasized the requirement of learning from the field to ascertain the performance of our system. In addition, The field experience allowed us to discover two new insights regarding real-world applied research in security games: (i) testing this research in the field requires a period of “immersion” and (ii) users are a key factor when running field experiments.

The first insight is a key lesson for running field experiments. Any real world test of a security game based system will involve real security officers protecting a critical area for a long period of time. To succeed in such an experiment, researchers should immerse themselves in order to deeply understand the way officers and, more generally, a security agency operate every day. A period of “immersion”, as we did for both the FE and the CT experiments, also ensures that the security agencies do not think researchers as ivory tower occupants leading to easier acceptance of technology. To test MOPSS, we spent several months observing the different security agencies patrolling the LA Metro to understand how they operate so as to set up effective field experiments.

The second insight comes from our interactions with the security personnel. These officers are the end users of our system. Thus, it is critical that they understand exactly the benefits of game-theoretic scheduling. Not doing this could severely affect the results of the evaluation. As an example, at the beginning of our FE tests (Section 4.1), the officers required a number of days to understand
that their schedules could be updated without having to request a new allocation to the dispatch.

6 Summary

This paper steps beyond deployment to provide results on security games in the field, a challenge not addressed by existing literature in security games. Readers will notice that the paper does not contain any simulation results as all of our results are based on real world experiments. We presented MOPSS, a novel game-theoretic scheduling system for patrolling a train line. MOPSS introduced five contributions not addressed in previous applied systems, including both TRUSTS [16] and the system in [15].

The first contribution is multi-operation patrolling. Thus far, all existing game-theoretic scheduling systems [14] (in particular TRUSTS) and the system in [15] were focused on a single mission. In contrast, MOPSS is the first deployed system to use three significantly different adversary models to develop three different patrol schedules for the threats of fare evasion, terrorism and crime. In contrast with previous work suggesting such threats could be modeled as a multi-objective security game [3]. A fundamental contribution of this paper is the insight that these different threat types lead to fundamentally different adversary models that cannot be folded into a single security game framework. MOPSS then is built upon these three adversary models. The second contribution deals with uncertain interruptions in the execution of patrol schedules. Existing systems, including TRUSTS [16], generated patrols that were often interrupted and left incomplete. This led to the use of MDPs for planning defender patrols in security games [7]. MOPSS exploits this idea to generate patrols for fare evasion. The third contribution is that MOPSS is the first system to generate patrols for counter-terrorism which accounts for joint coordinated activities between defender resources. This is achieved by incorporating the framework in [12] within both the SOLVER and the CT-Game in MOPSS. As a fourth contribution, MOPSS is the first system to deploy the Opportunistic Security Game model, where the adversary makes opportunistic decisions to commit crimes.

Finally, the fifth, and most important, contribution is the evaluation of MOPSS via real-world deployments. We ran three field experiments showing the benefits of game-theoretic scheduling in the real world. To the best of our knowledge, this evaluation constitutes the first evaluation of security games and, most importantly, the largest evaluation of algorithmic game theory, in the field. Existing literature on game theory in the field has focused on showing equilibrium concepts in the human and animal activities [10, 2]. Our work shares their enthusiasm of taking game theory to the field, but fundamentally focuses on algorithmic deployments and the impact of such algorithms. Most importantly, our work opens the door of applied research in security games to the realm of field evaluation. Given the maturity that such research has acquired in the recent years and its strong connection with real world patrolling problems, we argue
that field deployment should become a key area for future research in security games.

References

The AORTA Architecture: Integrating Organizational Reasoning in Jason

Andreas Schmidt Jensen¹, Virginia Dignum², and Jørgen Villadsen¹

¹ Technical University of Denmark, Kongens Lyngby, Denmark
{ascje,jovi}@dtu.dk
² Delft University of Technology, Delft, The Netherlands
m.v.dignum@tudelft.nl

Abstract. Open systems are characterized by a diversity of heterogeneous and autonomous agents that act according to private goals, and with a behavior that is hard to predict. They can be regulated through organizations similar to human organizations, which regulate the agents’ behavior space and describe the expected behavior of the agents. Agents need to be able to reason about the regulations, so that they can act within the expected boundaries and work towards the objectives of the organization. In this paper, we propose the AORTA¹ architecture for making agents organization-aware. It is designed such that it provides organizational reasoning capabilities to agents implemented in existing agent programming languages without being tied to a specific organizational model. We show how it can be integrated in the Jason agent programming language, and discuss how the agents can coordinate their organizational tasks using AORTA.

1 Introduction

Open systems rely on organizational structures to guide and regulate agents, because these systems have no control over the internal architecture of the agents. This means that the agents must be able to reason about the organizational structures in order to know what to do in the system and how to do it. Regulations are often specified as organizational models, usually using roles that abstract away from specific agent implementations such that any agent will be able to enact a given role. Roles may restrict enacting agents’ behavior space, such that it coincides with the expectations of the system. The system can then be regulated, for example, by blocking certain actions (for example through a middleware, such as S-MOISE⁺ [9]), or by enabling the agents to reason about the expectations of the system.

¹ Adding Organizational Reasoning to Agents
Agents that can reason about organizations are organization-aware [14]. This includes understanding the organizational specification, acting using organizational primitives, and cooperating with other agents in the organization to complete personal or organizational objectives. From the agent’s perspective, there are two sides of organizational reasoning. First, how can it contribute to the objectives of the organization, and second, how can it take advantage of the organization, once it is a part of it.

AORTA (Adding Organizational Reasoning to Agents) [12] is an organizational reasoning component that can be integrated into the agent’s reasoning mechanism, allowing it to reason about (and act upon) regulations specified by an organizational model using simple reasoning rules. AORTA assumes a preexisting organization, is independent from the agent, and focuses on reasoning rules that specify how the agent reasons about the specification. The organization is completely separated from the agent, as shown in figure 1, meaning that the architecture of the agent is independent from the organizational model, and the agent is free to decide on how to use AORTA in its reasoning. The separation is possible because AORTA is tailored based on an organizational metamodel, designed to support different organizational models.

In this paper, we propose the AORTA architecture for making agents organization-aware. It is designed such that it can provide organizational reasoning capabilities to agents implemented for existing agent platforms. We present an integration of AORTA in the well-known agent platform Jason [1], and show how it lets Jason-agents decide how to use their capabilities to achieve their organizational objectives, and furthermore, how they are able to coordinate their tasks.

We consider software architecture as the highest level of abstraction of a software system. The AORTA architecture is designed as a component that can be integrated into existing agent platforms. Existing agents are linked to an AORTA-agent, which features an organizational reasoning cycle that performs organizational reasoning, providing the existing agent with organizational reasoning capabilities. Furthermore, the organizational reasoning is specified in an AORTA-program in which organizational actions and coordination mechanisms for each agent can be defined by the developer.

The rest of the paper is organized as follows. We begin, in section 2, with a description of the organizational metamodel, and briefly discuss a simple scenario, which we later implement in AORTA and Jason. In section 3, we present

---

2 The implementation of the AORTA architecture is available as open source at [http://www2.compute.dtu.dk/~ascje/AORTA/](http://www2.compute.dtu.dk/~ascje/AORTA/).
the AORTA architecture. Section 4 describes the integration with Jason. We discuss related work in section 5 and conclude the paper in section 6.

2 Organizational model

Organizational models are used in multi-agent systems to give agents an explicit representation of an organization. Different models are proposed in the literature (e.g. MOISE+ [9], OperA [5], ISLANDER [6]). A common trait is the use of roles, abstracting implementation details away from expectations, and objectives, defining the desired outcome of the organization.

Reasoning in AORTA is based on an organizational metamodel, which supports different organizational models. The metamodel is based on roles and objectives.

A role, role(r, O) has a name, r, and is responsible for a set of objectives, O. An objective is denoted objective(o). Roles can form a dependency relation over an objective, dependency(r1, r2, o), such that r1 depends on r2 for the completion of an objective o. Objectives may be partially ordered, order(o1, o2), indicating that certain objective o1 must be completed before objective o2. Role enactment is denoted rea(a, r), which means that agent a enacts role r. Furthermore, active(o) denotes objective o is active (an objective is active if it has not yet been completed and all objectives it depends on have been completed).

Existing organizational models can be mapped to this metamodel. For example, if MOISE+ is being used, an objective is a goal, which is part of a mission, and a role would be responsible for missions it is permitted or obligated to pursue. Note that since the metamodel is currently based on roles and objectives, and has no notion of norms, it is not yet possible to reason about norms that are enforced.

2.1 First responders

We consider a scenario of first responders at a fight between groups of people, some of them being injured and requiring medical attention.

After a match between Manchester United and Manchester City, the fans are fighting and some of them are badly hurt. The authorities have been contacted, and a group number of medics and police officers (the first-responders) have arrived. The medics are supposed to help the injured, while the police officers are supposed to break up the fight. The fans may try to prevent medics from rescuing injured fans from the other team.

The organizational specification is shown in figure 2. For this paper, we assume that the agents entering the organization are cooperative, that is, they will pursue organizational objectives and cooperate with the other agents in the organization. It is, however, simple enough to consider self-interested agents as well; they will just be more likely to pursue their personal objectives rather than those of the organization.
Agents in the scenario will have to reason about which role(s) to enact, how to achieve and coordinate their objectives, and how to complete objectives that the agents are not capable of achieving themselves (i.e., by delegating to another, more capable agent).

3 The AORTA architecture

Classical BDI agents are represented by sets of beliefs, desires and intentions, where desires are possible states of affairs that the agent might want to realize, and intentions are those states of affairs that the agent has committed to (attempt to) realize. A similar representation can be made for organizational reasoning: the agent holds beliefs about the organization (its specification and instantiation) and can use that for reasoning about organizational objectives that are possible (or required) to be achieved, roles that can be enacted, norms that are enforced, and so on. An integration of the organization within the agent, makes the agent more likely to take both the organization and its own beliefs into account in its reasoning. Furthermore, by representing the organization as beliefs, the organizational structure can be changed, if necessary. For example, if the organization changes (reorganization), or if the agent finds out that it has wrong beliefs about the organization.

AORTA provides organizational reasoning capabilities to agents, and extends classical BDI reasoning, allowing the agents to reason about organizational matters. Organizational reasoning is divided into organizational option generation, organizational action deliberation and organizational coordination. An organizational option is something that the agent should consider, such as an active objective, or a role that can be enacted or deacted [11]. For instance, initially in the scenario, the medics will only search for injured people. When all areas have been searched, this objective has been completed and a new objective, rescuing the injured, will be possible. An organizational action is the execution of an organizational option: actually enacting a role or committing to an organizational objective. This creates the expectation (for the organization) that the agent should somehow believe it is able to (help) achieving it, either by itself, by cooperating with other agents, or by delegating it to one or more agents in
Fig. 3. The Organizational Reasoning Component of AORTA.

the dependency relation of its role. Note that self-interested or deceitful agents might know that they cannot achieve an organizational objective, but will commit to it anyway to disturb the organization. Organizational coordination is organization-level coordination, which is based on the agent’s mental state.

The organizational reasoning component of AORTA is depicted in figure 3. The agent (assumed to be a BDI agent) has a mental state, which is coupled to AORTA. Based on the mental state, AORTA can determine which organizational options to choose, and the organizational actions might change the mental state. For instance, in order to consider the available organizational options, AORTA uses the agent’s capabilities and intentions. Furthermore, intentions may influence the reasoning, e.g., when the intention to coordinate a task requires use of the organizational model. Finally, AORTA lets agents commit to objectives: an organizational action leads to change in the agent’s intentions, corresponding to the fact that the agent commits to the objective. The coordination component sends messages using the mailbox, and incoming messages can change the organizational structure.

3.1 Mental state

BDI agents usually have knowledge bases containing their beliefs and intentions. AORTA-agents are agents that contain an AORTA-component, which means that they not only have belief and intention bases, they also have knowledge bases for the organizational aspect. Each knowledge base will hold different kinds of formulas depending on their purpose.

Definition 1 (Knowledge bases). The AORTA knowledge bases are based on a predicate language, $\mathcal{L}$, with typical formula $\phi$ and operators $\land, \neg, \forall$. The agent’s belief base and intention base are denoted $\Sigma_a$ and $\Gamma_a$, respectively. The language of the organization is denoted $\mathcal{L}_{org}$, and $\mathcal{L}_{org} \subseteq \mathcal{L}$. The organizational specification and options are denoted $\Sigma_o$ and $\Gamma_o$, respectively. We then have the following
knowledge bases:

\[ \Sigma_o, \Gamma_o \subseteq L^{org} \quad \Sigma_a, \Gamma_a \subseteq L \]

We define different kinds of formulas for each knowledge base, which allows us to target specific knowledge bases in different situations.

**Definition 2 (Formulas).** AORTA uses reasoning formulas, \( L_R \), with typical element \( \rho \), which are based on organizational formulas, option formulas, belief formulas and goal formulas.

\[ \rho ::= \top | \text{org}(\phi) | \text{opt}(\phi) | \text{bel}(\phi) | \text{goal}(\phi) | \neg \rho | \rho_1 \land \rho_2 \]

Organizational formulas, \( \text{org}(\phi) \), queries the organizational specification, option formulas, \( \text{opt}(\phi) \), queries the options base, belief formulas, \( \text{bel}(\phi) \), queries the belief base and goal formulas, \( \text{goal}(\phi) \), queries the intention (or goal) base. We can use the formulas to specify things such as:

\[ \text{org(objective(injuredFound))} \land \neg \text{bel(injuredFound)} \]

where the first part of the conjunction queries the organizational specification, \( \Sigma_o \), and the second part queries the agent’s belief base, \( \Sigma_a \). The formula queries whether there is an organizational objective (to find victims), which the agent currently does not believe it has achieved.

**Definition 3 (Mental state).** The AORTA mental state, \( MS \), is a tuple of knowledge bases:

\[ MS = \langle \Sigma_a, \Gamma_a, \Sigma_o, \Gamma_o \rangle \]

The implementation of the mental state is based on tuProlog [4], which is a Java-based lightweight implementation of ISO-Prolog. We chose tuProlog because of its efficiency and straightforward interface in Java, allowing us to query a Prolog database without requiring any external system-dependent libraries. Each AORTA-agent has its own instance of tuProlog, comprising its entire mental state. That is, all knowledge bases of an agent are implemented in a single Prolog instance by wrapping each rule in a predicate depending on its nature. For example, the reasoning formula \( \text{bel}(a \land b) \land \neg \text{org}(c \land d) \) is converted to the following Prolog query: \( \text{bel}(a), \text{bel}(b), \neg \text{org}(c), \neg \text{org}(d) \). This translation makes querying straightforward, while still keeping the distinction between the different knowledge bases.

Note that we let AORTA-agents have their own mental state, rather than integrating AORTA into the knowledge bases of an agent in an existing platform. This means that the belief base and goal base of AORTA must be synchronized with the agent, which could lead to pitfalls in an integration process (especially if the knowledge bases are not properly synchronized). However, our aim is to enable AORTA to be integrated with most of the existing agent platforms, and since it requires only that formulas must be converted between the language of AORTA and the agent platform in question, we find that it makes the implementation of AORTA simpler to understand.
3.2 Acting and coordinating

At the center of AORTA-agents are the organization-specific actions. While an agent will have access to a number of domain-specific actions (such as a medic performing a life-saving action), an AORTA-agent will furthermore be able to consider certain organizational options (what happens by enacting a certain role, pursuing an objective), or performing organizational actions (enacting a role, committing to an objective).

**Definition 4 (Organization-specific actions).** The set of options with typical element $a_O$ is denoted $\text{Opt}$ and the set of actions with typical element $a_A$ is denoted $\text{Act}$.

\[
\begin{align*}
    a_O & ::= \text{consider}(\phi) \mid \text{disregard}(\phi) \\
    a_A & ::= \text{enact}(\rho) \mid \text{deact}(\rho) \mid \text{commit}(\phi) \mid \text{drop}(\phi)
\end{align*}
\]

Actions are executed using a transition function, $T_O$ and $T_A$, respectively. Each action is only applicable in certain states. For example, $\text{consider}(\phi)$ can only be applied if $\Sigma_o \models \phi$ in the current state, and the effect is that $\phi$ is added to $\Gamma_o$. Role enactment, $\text{enact}(\rho)$, is applicable only when $\rho$ is the name of a role, the agent does not currently enact that role. Committing to an objective, $\text{commit}(\phi)$, is possible only if $\phi$ is an organizational objective, and $\phi$ is not already a belief or a goal\(^3\). $\text{disregard}(\phi)$, $\text{deact}(\rho)$ and $\text{drop}(\phi)$ simply remove the respective formula from the appropriate knowledge base.

Notice the correspondence between elements in $\text{Opt}$ and $\text{Act}$: if the agent considers enacting a role, the enact action allows it to enact that role. However, once the role is enacted, the option is no longer an option. Since the agent now enacts the role, it seems appropriate to remove the option from $\Gamma_o$. This is done using an option removal function, $O$, which removes options, when they are no longer applicable (that is, when their respective organizational action would be undefined).

We are now in a position to introduce organizational reasoning rules: option and action rules. These rules enable the agent to decide which organization-specific actions to perform.

**Definition 5 (Reasoning rules).** The sets of option rules $\mathcal{R}_O$ and action rules $\mathcal{R}_A$ are defined as follows.

\[
\begin{align*}
    \mathcal{R}_O & = \{ \rho \implies a_O \mid \rho \in L_R, a_O \in \text{Opt} \} \\
    \mathcal{R}_A & = \{ \rho \implies a_A \mid \rho \in L_R, a_A \in \text{Act} \}
\end{align*}
\]

Finally, since each agent has its own organizational state, they need to be able to coordinate and synchronize their organizational knowledge. While such

\[^3\] The correspondence between goals and beliefs is based on achievement goals in the GOAL agent programming language [7], which are defined such that $\phi$ is an achievement goal iff $\phi$ is a goal and $\phi$ is not currently believed.
coordination can happen in different ways, we choose to use organizational messages. In order to determine whether a message is intended for AORTA, organizational messages are wrapped in an organizational wrapper, \( \text{om} \), which is an unary predicate with the message as a single term.

**Definition 6 (Organizational Messages).** An organizational message is defined as

\[
\text{msg}(\alpha, \text{om}(M)),
\]

where \( \text{om} \) is the organizational wrapper, and \( M \) is the message. In outgoing messages, \( \alpha \) corresponds to the set of recipient agents, and in incoming messages, \( \alpha \) is the sender.

Each agent can then specify how to coordinate using a set of coordination rules, which specifies certain criteria for when and with whom to coordinate.

**Definition 7 (Coordination rules).** A coordination rule is a triple,

\[
(c, \phi, m),
\]

where \( c \) is the trigger for coordination and is a set of positive or negative reasoning formulas, \( \phi \) defines the set of agents to coordinate with, and \( m \) is the message.

The coordination trigger \( c \) can, e.g., be the set \( \{\text{bel}(\text{injuredFound})\} \), which will trigger at a point where \( \Sigma_{\alpha} \models \text{injuredFound} \) is true and \( \Sigma_{\alpha} \models \neg\text{injuredFound} \) was true in the previous state.

### 3.3 AORTA reasoning cycle

The configuration of an AORTA-agent consists of the agent’s knowledge bases, a number of option, action and coordination rules, and a message box for incoming (inbox) and outgoing (outbox) organizational messages. The initial state consists of a set of initial beliefs and goals, and the organizational specification.

The agent has a number of state transition rules available, which can be used to change its state. A reasoning cycle in AORTA is executed using a strategy that decides which transition rules to execute.

The agent has transition rules for execution of option and action rules, called \( \text{OPT} \) and \( \text{ACT} \), a transition rule for external updates, \( \text{EXT} \), and two rules for coordination, \( \text{COORD} \) and \( \text{CHK} \).

\( \text{OPT} \) can be applied to an option rule in a given state, \( \rho \implies a_O \), if \( \rho \) is entailed and the option transition function, \( T_O \), is defined for \( a_O \).

\( \text{ACT} \) works similarly for action rules, using the action transition function, \( T_A \), and the option removal function, \( O \).

\( \text{EXT} \) changes the agent’s mental state to accommodate updates from outside AORTA. For example, if the agent perceives something, \( \text{EXT} \) adds the percept to the belief base.
options {
  \[\text{org}(\text{role}(\text{R},\text{Os})), \text{bel}(\text{me}(\text{Me})), \text{member}(\text{D},\text{Os}), \text{cap}(\text{O}))\] => consider(\text{role}(\text{Role},\text{Os}))
  \[\text{bel}(\text{me}(\text{Me})), \text{org}(\text{role}(\text{R},\text{Os})), \text{rea}(\text{Me},\text{R}), \text{member}(\text{D},\text{Os}), \text{objective}(\text{O}), \text{active}(\text{O}))\]
  => consider(\text{objective}(\text{O}))
}

actions {
  \[\text{opt}(\text{role}(\text{Role},..))\] => enact(\text{R})
  \[\text{opt}(\text{objective}(\text{O})), \text{org}(\text{role}(\text{R},\text{Os})), \text{member}(\text{D},\text{Os}), \text{rea}(\text{Me},\text{R}), \text{bel}(\text{me}(\text{Me})))\] => commit(\text{O})
}

coordination {
  \[\text{+bel}(\text{visited}(\text{R}))\] : \[\text{org}(\text{rea}(\text{A},\text{medic}))\] => send(\text{A},\text{bel}(\text{visited}(\text{R})))
  \[\text{+goal}(\text{X})\] : \[\text{bel}(\text{me}(\text{Me})), \text{org}(\text{rea}(\text{Me},\text{R1})), \text{dependency}(\text{R1},\text{R2},\text{X}), \text{rea}(\text{A},\text{R2}))\]
  => send(\text{A},\text{goal}(\text{X}))
  \[\text{+bel}(\text{O})\] : \[\text{org}(\text{role}(\text{R},\text{Os})), \text{objective}(\text{O}), \text{member}(\text{D},\text{Os}), \text{rea}(\text{A},\text{R}))\] => send(\text{A},\text{bel}(\text{O}))
  \[\text{+org}(\text{rea}(\text{A},\text{R}))\] : \[\text{bel}(\text{agent}(\text{Ag}))\] => send(\text{Ag},\text{org}(\text{rea}(\text{A},\text{R})))
}

Fig. 4. An example of an AORTA program.

COORD is applied to coordination rules, \((c, \phi, m)\), when \(c\) is triggered by the
state, and the set of agents entailed by \(\phi\) is not empty. The message \(m\) is
then sent to each agent.

CHK takes new messages from the incoming message queue and adds them to
the appropriate knowledge base\(^4\).

For the purpose of this paper, we use a single linear strategy, which executes
the state transition rules in a predefined order.

**Definition 8 (Linear strategy).** The linear strategy is defined as follows:

\[(\text{CHK})^*(\text{EXT})(\text{OPT})(\text{ACT})(\text{COORD})^*\],

where \((\text{RULE})^*\) denotes that \text{RULE} is executed until the agent’s state no longer
changes.

The strategy executes each of the transition rules, as explained above, changing
the agent’s state. The linear strategy is rather simple, but it is possible to
implement strategies, which e.g. allows the agent to explore different paths before
choosing one.

### 3.4 AORTA programs

An AORTA program consists of three sections: \(\text{options}, \text{actions}\) and \(\text{coordination}\).
An example program, which can be used in the first responders scenario, is shown
in figure 4.

Options and actions are of the form \(\lbrack \phi \rbrack \Rightarrow a\), where \(\lbrack \phi \rbrack\) consists of a comma-separated list of reasoning formulas. The content of each reasoning formula (i.e.,
the query) is Prolog code. For example, the action rule

\[\text{opt}(\text{role}(\text{R},..))\] => enact(\text{R}).

---

\(^4\) For simplicity, we assume that the agents will not consider whether a sender is
trustworthy, and thus whether a message is reliable.
states that if role(R, _ ) is an option (i.e. entailed by \( \Gamma_o \)), the agent should enact \( R \). Note that this is a simplification of the reasoning process required by agents to decide whether or not to enact a role in an organization. It is, however, possible to incorporate more sophisticated reasoning, e.g., by using the notion of social power. For example, in [3], various forms of power agents may have over each other are identified and formalized as rules. These power relations can be used in the reasoning process by adding the rules to the agents’ organizational state.

The coordination section consists of coordination triples, of the form \([c] : [\phi] \Rightarrow \text{send}(Ag, \psi)\), where \( c \) is a list of reasoning formulas, with either + or - in front of each, denoting that the trigger or its negation is now entailed by the agent’s mental state. \( \phi \) is identical to \( \phi \) in option and action rules. \( Ag \) corresponds to a variable in \( \phi \) or \( c \), and \( \psi \) is the message to be sent. Thus, the following rule

\[
[+\text{org(rea(A,R))}] : [\text{bel(me(A),agent(Ag))}] \Rightarrow \text{send}(Ag, \text{org(rea(A,R)))}
\]

states that when the agent enacts a role, it should inform all other agents in the system.

The implementation of OPT and ACT is deterministic: the rules in each section are simply processed linearly, and the first matching rule is executed. COORD is implemented such that every triggered triple in a state will be executed in a single step.

3.5 Implementation overview

The architecture is depicted in figure 5. The system is implemented in the class Aorta, which contains a list of the agents in the system and a reference to the original organizational specification. Each AORTA-agent is associated with an instance of AortaAgent, which contains the agent’s state, AgentState, and in which the reasoning cycle is implemented. The reasoning cycle performs two steps: executing the strategy and sending messages from the outbox.

3.6 Integration considerations

The agent state contains the agent’s the knowledge bases, rules and message boxes. Furthermore, it contains an ExternalAgent and an AortaBridge. The external agent corresponds to the message box and knowledge bases of the agent using AORTA. That is, whenever the agent commits to a new goal or updates its beliefs, these changes are propagated via the external agent into AORTA using EXT. The bridge lets AORTA manipulate the agent’s mental state. For example, successful execution of commit(\( \phi \)) will add \( \phi \) to the agent’s goal base using the bridge.

When integrating AORTA into an existing agent platform, there are thus three things to take care of.

Bridge AORTA uses the bridge to send updates to the agent’s goal and belief bases, so an agent platform-specific bridge should be implemented (by implementing the AortaBridge interface), such that the knowledge bases can be synchronized.
Fig. 5. Implementation overview with the most important classes. A filled arrowhead indicates an association between classes. An unfilled arrowhead indicates inheritance.

**External agent** When the agent updates its goal or belief base, it should inform AORTA by invoking the appropriate methods of *ExternalAgent*.

**Translation** AORTA makes use of tuProlog, so the contents of the agent’s knowledge bases should be translated into Java objects supported by tuProlog.

4 Jason+AORTA

We now show how AORTA can be implemented in an existing agent platform, the Jason platform [1]. Jason is a Java-based interpreter for an extended version of AgentSpeak. Jason is based on the beliefs-desires-intentions (BDI) model, is open source and highly extensible, making it a reasonable choice for the integration of AORTA.

The AgentSpeak language is a Prolog-like logic programming language, which allows the developer to create a plan library for each agent in a system. A plan in AgentSpeak is of the form

\[ +\text{triggering event : context <- body.} \]

If an event matches a trigger, the context is matched with the current state of the agent. If the context matches the current state, the body is executed; otherwise the engine continues to match contexts of other plans with the same trigger. If no plan is applicable, the event fails. Triggering events can amongst other things be addition or deletion of beliefs (+l and −l) and addition or deletion of goals.
Fig. 6. Jason+AORTA. A filled arrowhead indicates an association between classes. An unfilled arrowhead indicates inheritance.

(+!l and −!l). The body contains a sequence of actions the agent should perform and goals to adopt. When adopting a goal in the body of a plan, the agent will attempt to achieve the new goal before continuing executing the current plan.

Note that when a plan for a goal has been completed, the goal is considered finished. This means that it will be removed from the agent’s mental state. Since commit(φ) is only defined if φ is not already a goal and is not believed by the agent, the agent will be able to commit to a goal multiple times, until it believes it has been achieved.

The AORTA integration in Jason is shown in figure 6. The integration consists of an extended agent architecture, which implements the actual integration with AORTA, and an infrastructure, which makes it possible to create an AORTA-project in Jason without having to deal with the specifics of the integration. This is done by specifying the infrastructure as follows:

```
MAS projectname {
    infrastructure: AORTA(organization(location, type))
    ...
}
```

The infrastructure takes two parameters: location refers to the location of the organizational specification, and type refers to the type of organizational model (currently, the a generic organization based on the metamodel is supported).
AORTA does not make any changes to the Jason language, and any existing implementations of multi-agent systems in Jason should be compatible with Jason+AORTA. The integration does two things: (1) when the belief base or goal base of the AORTA-agent changes, these changes are propagated to the Jason-agent (via AortaJasonBridge), and an addition/deletion event is triggered and (2) when the Jason-agent’s mental state changes, AORTA receives those changes (via the ExternalAgent). The Jason-agent is connected to the ExternalAgent in three places:

AortaAgentArch Organizational messages are filtered and sent to AORTA for processing. The normal procedure for checking an agent’s mailbox is extended to check whether incoming messages are wrapped in the organizational wrapper.

AortaBB Whenever the Jason-agent’s belief base is changed (i.e., a belief is added or removed), those beliefs are sent to AORTA to ensure synchrony between the mental states.

AortaGoalListener When a goal changes state (i.e., when a plan for it has started, failed, or stopped), the goal listener is responsible for sending the changes to AORTA.

Furthermore, Jason formulas are converted to AORTA formulas. Note that while Jason supports annotations on literals (e.g., denoting the source of a belief, injuredFound[source(alice)]), they are lost in conversion to AORTA formulas, since they are not supported. This should generally not be a problem, since formulas will not propagate back and forth between the systems. That is, if a belief originates from Jason, it will be sent to AORTA, which will not send it back to Jason, e.g. +injuredFound[source(alice)] → bel(injuredFound) → +injuredFound does not happen.

The AORTA reasoning cycle is executed in Jason via the method reasoning-CycleStarted() in AortaAgentArch, which is called in the beginning of a Jason reasoning cycle. This means that the agent will execute the AORTA reasoning strategy in the beginning of each cycle.

4.1 The first responders scenario

We now discuss how AORTA can be used to let agents participate in the first responders scenario. We use the Blocks World for Teams [13] testbed to simulate the first responders scenario by considering the drop zone being the ambulance, colored blocks being injured fans, and agents playing the roles of fans, medics and police officers. Fans are fighting just outside some of the rooms and they can stop the medic from rescuing injured fans by entering a room just before the medic does so. Police officers will look for areas where fans are standing, while medics will check the rooms to find injured fans.

Consider an agent, Bob, playing the role of a medic (Σo |= rea(bob, medic)), using the program in figure 4. He is considering the objective injuredFound (Γo |= objective(injuredFound)), to which he has not yet committed. The following action rule can then be executed.
Bob commits to finding the injured, which leads to the subgoal of visiting room1. When he believes he has visited the room (when he is inside the room), both goals will finish, since +injuredFound waited on the completion of +visited(room1). Since the main goal, injuredFound, has not yet been completed, Bob can execute the same action rule again, thus committing to the goal once more. Since there are no more rooms to visit, only the second plan is applicable, and he believes that all the injured fans have been found.

When injuredFound is achieved, several things happen. First, the following coordination mechanism is triggered:

\[
[+\text{bel}(O)] : [\text{org}(R,Os), \text{objective}(O), \text{member}(O,Os), \text{rea}(A,R))] \Rightarrow \text{send}(A, \text{bel}(O))
\]
Since $\text{bel}(\text{injuredFound})$ is added to the agent’s mental state, and $\text{injuredFound}$ is an objective, the agent will inform all agents responsible for that objective, that it has been completed. Second, the next objective, $\text{injuredSaved}$, becomes an option, and $\text{Bob}$ will then commit to completing it. The flow of execution is similar to that of figure 7 and will not be described in detail.

If, during the rescue, a room is blocked by a fan, the agent may adopt a goal, $\text{removeBlocker}$, which will trigger the following coordination mechanism:

$$\text{[+goal(X)]} : [\text{bel(me(Me))}, \text{org(rea(Me,R1)}, \text{dependency(R1,R2,X), rea(A,R2)}]$$

$$\Rightarrow \text{send(A, goal(X))}$$

Since the agent commits to a goal for which there is a dependency, he sends a request to the agents enacting the role R2 (in this case the $\text{officer}$ role). An officer should then commit to achieving the goal, and inform the medic when it has been done.

### 5 Related work

The $\text{MOISE}^+$ model is based on three organizational dimensions: the structural, functional and deontic dimensions [9]. Development of organized multi-agent systems using the $\text{MOISE}^+$ model is separated into a system and an agent level. The system level, $\text{S-MOISE}^+$, provides an interface (a middleware) between the agents and the organization using a special agent, the OrgManager, to change the organizational state, ensuring organizational consistency. The agent level, $\text{J-MOISE}^+$, joins Jason and $\text{MOISE}^+$, by making organizational actions available to agents, such that they can reason about (and change, using the OrgManager) an organization.

Similar to AORTA-agents, agents in $\text{J-MOISE}^+$ receive objectives (missions) that they can achieve using Jason plans. The main difference is that the organization-oriented reasoning is done as a part of the agent’s normal reasoning process, whereas AORTA-agents perform the organizational reasoning inside AORTA, and then decides how to complete their objectives at a different level. The main advantage of keeping the reasoning apart in AORTA is that it allows agents on different agent platforms to perform the same kind of organizational reasoning without any extra development required.

The ORA4MAS (Organizational Artifacts for Multi-Agent Systems) approach [8] is another attempt to build a bridge between an organization and the agents in it. It is a general approach suitable for different kinds of organizational models. They use artifacts, which they claim brings the control back to the agents (as compared to using a middleware), since the agents can, via their autonomy, choose whether to interact with the organizational artifacts of the system.

We argue that the ultimate way of bringing the control back to the agents is to allow the agents themselves to perform the organizational reasoning. By integrating AORTA in agents, they are provided with organizational reasoning capabilities, while they are still able to, e.g., decide not to commit to certain organizational objectives.
6 Conclusion and future work

We have described the AORTA architecture and have shown how it can be integrated in the Jason platform. The example shows how Jason-agents gain capabilities to reason about which organizational objectives to commit to, and how to coordinate completing them.

AORTA lets the developer focus on implementing the agents’ domain-specific capabilities, while commitment to organizational objectives, coordination, and communication can be done entirely by AORTA. Furthermore, since AORTA can be integrated in different agent platforms, the same AORTA programs can be used for several different implementations in different agent programming languages. The use of the simple, generic language makes AORTA readily useful, however, the support of an existing, and more powerful, organizational language, such as Moise or OperA, is a natural extension to the architecture.

The decoupling of AORTA and the agent platform means that synchronization is required. However, the linear strategy makes sure that external changes are synchronized before options and actions are considered (via the Ext transition rule). As mentioned, the requirement is a translation between AORTA formulas and the formulas of the connected agent (e.g. AgentSpeak formulas). Furthermore, organizational reasoning is done in AORTA and is thus separated from the agent’s normal reasoning. This is because the organizational state is only available to AORTA, as it is not shared with the agent. This means that the agent cannot reason about organizational matters, such as role enactment and organizational objectives without using the rules of AORTA. However, if necessary, in the case of Jason, it is possible to allow this kind of reasoning by introducing an internal action, e.g. \texttt{.org(Fml)} which succeeds if Fml can be translated to an AORTA formula and is entailed by the organizational state.

In the future, we plan to investigate other strategies that could improve the reasoning, such as a strategy that explores different paths of execution, and makes a decision based on this. Furthermore, since agents may have objectives that do not coincide with the organizational objectives, they need a way to decide which objectives to pursue, for example using a preference ordering [2] or individual agent preferences [10].

Finally, we are investigating how to incorporate norms in the semantics, such that the agents are able to deliberately follow paths that violate the organization, while possibly being sanctioned by other agents in the organization.

References


Abstract. Continuous improvement is a procedure to improve products, services or processes. In the Software Engineering domain, software process improvement means understanding existing development processes and changing them to increase product quality and reduce development costs and time. In this paper, we present the Medee Improvement Cycle, which adopts this approach to improve development methods for Organization Centered Multiagent Systems (OC-MAS). Such a cycle is anchored in the Medee Method Framework, which provides means for building methods through the combination of method fragments sourced from existing Agent Oriented Software Engineering methods (AOSE methods) and Agent Organization models (AO models). The Medee Improvement Cycle allows to continuous evolving MAS methods and fragments, taking into account a set of quality attributes, such as understandability, visibility, supportability, acceptability and robustness. We have shown through the case study how to apply this cycle to evolve fragments through their usage, instead of assuming that we have already the definitive version of them from the beginning.

1 INTRODUCTION

Organization-centered multiagent systems (OC-MAS) are systems whose basic conceptual entity is the agent organization as a whole, composed of a set of goals, norms, and functionalities, as well as an internal structure of components, like subsystems [18].

Such approach adopts a sociological and organizational vision for modeling MAS, involving organizations, teams and inter-agent relationships notions. Research in this area usually provides Agent Organization models (AO models) to support the specification of organizational aspects during MAS development and possibly changing them during MAS execution, such as MOISE+ [16] and OperA [13]. Nevertheless, these models do not address a structured MAS development cycle in terms of phases, tasks, and work products, as extensively accepted by the software industry [19].

Moreover, although some existing Agent-Oriented Software Engineering (AOSE) methods, such as Gaia [25] and Ingenias [20], propose the development of MAS based on the notion of agent organization, they deal with organization specification at design time, preventing the modification of the organization core aspects during
runtime. At the best of our knowledge, just an embryonic work on supporting the OC-MAS development cycle is available [24].

In order to fill this gap and provide reuse of existing AOSE methods and AO models, Casare et al [9] propose the Medee Method Framework (Medee for short), which allows the development of OC-MAS in a disciplined way, even though some AO models are not currently incorporated into AOSE methods. In order to do that, such a method framework proposes the composition of MAS situational methods out of method fragments according to a given project situation, by applying the principles proposed by Situational Method Engineering [7] [14]. The proposed approach provides a high degree of reuse and flexibility, allowing the composition of new methods based on software industry standards for method description, such as SPEM [19]. Furthermore, it allows the user to leverage advantages of both AOSE methods and AO models in order to develop OC-MAS.

Given that such situational methods are built on demand for immediate use and stored for further reuse, it is desirable that both methods and fragments could be improved in a continued way. Therefore, the definition of a cycle for guiding the continuous improvement of such methods and fragments would reinforce the development of OC-MAS.

In this paper we present the Medee Improvement Cycle, a continuous cycle for evolving fragments and situational methods for MAS. The usage of such a cycle is illustrated in a case study conducted to improve fragments sourced from the MOISE+ organization model. The Medee Improvement Cycle is based on the idea that, although processes, methods, and tools are essential to the development of MAS, they should be built upon a continuous software process improvement in order to focus on product quality (e.g. OC-MAS applications quality) as well as on reducing MAS development costs and time [21][22]. In brief, this cycle covers a whole improvement process for MAS situational method: from tailoring a method according to the project characteristics to learning from the results how to evolve the method itself, in a way that such lessons learned could give rise to method improvement.

Nevertheless, before explaining the Medee Improvement Cycle in details, which is done in Section 4, in Section 2 we briefly present the Medee Method Framework and in Section 3 we present the fragments sourced from MOISE+ that we have used. Our case study is presented in Section 5, and we discuss the advancements achieved with our approach in Section 6. Finally, we conclude the paper in Section 7.

2 MEDEE METHOD FRAMEWORK

The Medee Method Framework supports the composition of MAS methods on demand, especially the ones for developing OC-MAS. In brief, it consists of a repository containing method fragments sourced from several AOSE methods and AO models, as well as a process for populating such a repository and a model for composing situational methods out of fragments according to a given MAS project situation. In order to do that, this framework encompasses three components: the Medee Method Repository, the Medee Delivery Process, and the Medee Composition Model. Together,
these components cover most of a typical situational method procedure - from managing the method repository to building and publishing the situational method - in a seamless way, since they are based on the same conceptual model, i.e. the Medee Conceptual Model, as illustrated in Fig. 1. Moreover, this figure highlights that situational methods are published as HTML pages.

Fig. 1. The Medee Method Framework main components and functionalities

AOSE methods and AO models involve particular aspects, such as specific system architecture, development platform, design and programming languages. The Medee Method Framework takes these aspects into account in an integrated way, from the method repository management to the situational composition. Firstly, the Medee user can elaborate method fragments in a standard way, for instance by using common MAS development roles like MAS Developer and MAS Tester, as well as categorizing them according to the underpinned MAS component (e.g. agent, environment, organization), the MAS nature (e.g. open, closed), the design language (e.g. UML, AUML), and the programming language (e.g. Java, AgentSpeak), among other criteria provided by a semiotic taxonomy for MAS fragments.

Secondly, the Medee user can clearly state the project characteristics in terms of people, problem, product, and resource factors. Such characterization takes into account AOSE aspects, like the project team previous experience with developing MAS, the agent architecture to be used, like BDI, and the kind of product to be delivered, such as OC-MAS or agent-centered MAS. Finally, issues like how to proceed for elaborating fragments, characterizing the project, selecting fragments and putting
them together in a situational method are described in great details in the Medee Delivery Process. This latter is published as a website and offers three phases: Method Element Capture, Method Fragment Elaboration, and Medee Method Composition phases.

A detailed description of the Medee Method Framework is available at the Medee website\(^1\) and also at [9]. Currently, the Medee Method Repository stores 64 (sixty-four) fragments sourced from AOSE methods such as Gaia, Tropos [6], PASSI [11] and Ingenias, from AO models like MOISE+ and Opera, and from general-purpose development methods such as USDP (Unified Software Development Process) [17]. This repository can be easily extended with fragments sourced from other AOSE methods since the Medee Delivery Process provides step-by-step tasks for the method repository population. Moreover, new fragments can be categorized according to more than 25 semiotic criteria provided by the Medee Composition Model. Such functionalities allow the user to manage the method repository in a consistent and disciplined way, despite the number of stored fragments.

3 METHOD FRAGMENTS FOR MOISE+

MOISE+ is a well-established AO model tailored for specifying OC-MAS. It describes a MAS organization in terms of three dimensions: structural, functional and a deontic dimensions. For each one of these dimensions MOISE+ proposes one homonym specification.

The method fragments sourced from MOISE+ consisted of the smallest fragments that compose a MAS situational method, which contain tasks that involve steps, input work products, output work product, and development roles, as illustrated in Fig. 2 (right side). It should be noted that Fig. 2 shows a screenshot of the Medee website prior to the case study. Indeed, it depicts the fragment MMF Analyze Organization with MOISE+, which was sourced from MOISE+ along with other four fragments: MMF Design Agent Organizational Behavior with MOISE+, MMF Design Organization with MOISE+, MMF Implement Agent with MOISE+, and MMF Implement Organization with MOISE+ (see Fig. 2, left side). These five fragments could take part in situational methods for developing OC-MAS projects.

It is important to observe that MOISE+ offers a conceptual framework and syntax to organizational specification, but it does not describe the work that should be done - as such activities, task or steps - to produce such specifications. Therefore, the fragments sourced from MOISE+ resulted from the analysis and interpretation made during a previous research presented in [8].

These fragments consisted of an important step towards the development of methods for OC-MAS. Nonetheless, they deserve to be improved through utilization. One way of doing that is using the Medee Improvement Cycle, as done during the case study presented in this paper.

---

\(^1\) http://medee.poli.usp.br/.
The Medee improvement cycle consists of an initial step towards the process improvement for developing MAS. It is anchored in an empirical procedure for continuous evolving MAS situational methods and fragments based on previous project experience.

This cycle is built upon two approaches proposed into distinct research areas: (i) the iterative procedure for building situational methods from the Situational Method Engineering field [7] [14], and (ii) paradigms for software improvement through experimentation proposed in the Software Engineering area, namely Quality Improvement Paradigm (QIP) [1] [2] and Goal Question Metric (GQM) [3]. QIP is an evolutionary software quality process that provides a mechanism for software improvement through experimentation and reuse, based on project experience. Such a paradigm proposes to treat software development as empirical experiments in order to learn with them and thus improve the way to build software.

GQM is a goal-driven approach for collecting data around a particular experiment. It encompasses three main notions: measurement goal, questions of interest, and metrics. In brief, such approach allows a set of goals related to project improvement targets to be identified and refined in terms of questions and metrics.

The Medee Improvement Cycle underpins seven steps that can be applied in two distinct scenarios, depending on the improvement target: a whole situational method or some method fragments. The first scenario involves the seven steps, while the second involves only five of them. The case study presented in Section 5 is concerned

---

2 MMF stands for Medee Method Fragment, MPS stands for Medee work Product Slot, MTV stands for Medee Task Variability, MPV stands for Medee Product Variability.
with the second scenario. Interested readers can find a case study involving the first scenario in [8]. Fig. 3 shows enclosed in solid bold line the five steps for improving fragments and, out of this rectangle, the other two steps involved on improving methods. The seven steps are described in the following.

**Step 1 - Characterize MAS project situation** using the Medee Composition Model. This step allows a Medee user to better understand and characterize the factors involved in the MAS project, mainly those related to AOSE aspects. Therefore, in this step s/he can clearly state project characteristics in terms of people, problem, product, and resource factors. Possible examples are: the project team has no previous experience of developing MAS, although having some skills related to agent-oriented methods and UML; the product to be delivered involves an organization-centered approach. This step is performed only while running the cycle for situational methods (first scenario).

**Step 2 - Set MAS measurement goals** with a GQM model. It consists of establishing a goal-driven model based on the GQM paradigm, according to the MAS improvement targets selected for the empirical procedure. The following method quality attributes proposed by Sommerville [22] were adopted as a backbone to define the measurement goals: understandability, supportability, visibility, acceptability, reliability, robustness, rapidity, and maintainability. For instance, a goal related to fragment understandability could measure how easy it is to understand its elements (e.g. task, work product, roles), while another related to supportability could measure how easy it is to navigate in the website that describes methods and fragments. Although such quality attributes provide a steady basis for starting specifying measurement goals, they may be extended and refined according to a given set of method improvement
Section 5 presents the goal-driven model instantiated to measure the method fragments sourced from MOISE+ during the case study, involving goals concerned with understandability, supportability and visibility.

**Step 3 - Compose MAS situational method.** This step is performed only while running the cycle for situational methods. It consists of generating a situational method according to the current MAS project situation, by executing the Medee Method Composition phase of the Medee Delivery Process. Brandão et al [5] recently provided some automated support for selecting fragments in Medee in order to facilitate situational composition.

**Step 4 - Collect metrics** after analyzing a set of method fragments or using a situational method. It consists of using situational methods or fragments and gathering the metrics specified through the goal-driven model. Every usage of fragments or methods will be considered as an experiment. Moreover, this step involves designing questionnaires that are filled out by the participants of the experiment, as well as validating data provided by them. Examples of how metrics can be collected and validated are presented and discussed in Section 5.

**Step 5 - Analyze the measurement goals.** It consists of identifying the strengths and weakness of situational methods or fragments, through the assessment of the previously collected questionnaires’ answers. Examples of how to perform such analysis concerning MOISE+ fragments are presented in Section 5.

**Step 6 - Packaging experience** to improve the Method Repository. It consists of describing the lessons learned during the experiment in terms of improvement opportunities, in such a way that it could be used to update fragments and/or situational methods, as well as other building blocks underpinned by the Method Repository itself, like the Medee Glossary. Such updating is performed during the next step.

**Step 7 - Manage the Method Repository.** It consists of populating the Medee Method Repository with new elements as well as modifying/updating already stored elements - like method fragments and Medee methods - based on lessons learned during an experiment, as illustrated in Section 5. This step is mainly underpinned by two phases of the Medee Delivery Process [9] - Method Element Capture and Method Fragment Elaboration phases - both described in great detail at the Medee website.

Lessons learned can give rise to improvement opportunities in several ways, such as: (i) understandability concerns can drive method fragment re-elaboration or the creation of new guidelines and examples; (ii) fragment acceptability or reliability concerns can drive method fragments re-classification, and (iii) rapidity concerns are related to the project required effort, can be captured as estimation consideration and associated either with the whole method or the corresponding fragments. Section 5 presents MOISE+ fragments’ improvement done during the case study, while [8] illustrates improvements concerning a situational method.
Moreover, the Medee Improvement Cycle can be used to evolve the Medee Delivery Process itself, and not only fragments and situational methods. Examples of measurement goals and questions of interest that could be used in such an improvement scenario are:

**Goal 1**: Analyze the Medee Delivery Process for the purpose of evaluation with respect to the **usability**

- **Q1**: How ease is it to select method fragments according to the MAS project situation?
- **Q2**: How ease is it to put together the selected fragments in order to compose the situational method?

**Goal 2**: Analyze the Medee Delivery Process for the purpose of evaluation with respect to its **rapidity/efficiency**.

- **Q3**: How fast can the method engineer compose the situational method?

Summing up, these seven steps embedded in the Medee Improvement Cycle offer an evolving process for methods and fragments, from method tailoring to the updating of the method repository based on the lessons learned. As described in the next section, the lessons learned were packaged and integrated in the repository for further use in a seamless way.

5 **CASE STUDY**

The purpose of this case study was to investigate the use of the Medee Improvement Cycle for evolving fragments sourced from MOISE+. In a few words, it consisted of performing the step-by-step of such a cycle to improve MOISE+ fragments based on lessons learned.

Moreover, we would like to investigate how aware Medee users are about the improvements. Therefore, some steps of the improvement cycle were performed twice namely, steps 2, 4, and 5.

This case study was conducted in 2013 and involved undergraduate students, MOISE+ authors and MAS researchers skilled in MOISE+ notions and method engineering, totaling eight people.

The remainder of this section describes the five steps executed during this experiment, starting with Step 2 (Set measurement goals) and closing the cycle with Step 7 (Manage the method repository). It should be observed that evolving the fragments sourced from MOISE+ means also evolving the situational methods that include them as well as evolving of the method repository as a whole.
5.1 Setting the measurement goals (step 2)

This step consisted of defining a goal-driven model based on the GQM approach for evaluating the method fragments sourced from MOISE+ in terms of three quality attributes: understandability, visibility, and supportability. Therefore, it involved defining measurement goals by identifying the objects of study, issues, and viewpoints taken into account in this experiment, as well as detailing them through questions of interest and metrics. Firstly, the five MOISE+ fragments presented in Section 3 were considered as objects of study. Second, the issues consisted of the quality attributes understandability, visibility, and supportability. Thirdly, the viewpoint entities encompassed MAS developers, MOISE+ experts and method engineers.

Finally, these goals were refined through eleven questions of interest and related metrics. Some questions of interest took into account the developer viewpoint, while other considered the MOISE+ expert and Method Engineer viewpoints. In the following we present the three goals, and the associate questions and metrics.

**Goal 1:** Analyze the MOISE+ Method fragment for the purpose of evaluation with respect to the understandability.

Q1: To what extent has the fragment facilitated the understanding of MOISE+ aspects (e.g. concepts, specifications, implementation)?

**Metric 1:** Ranging from 1 (not helpful at all) to 5 (very useful).

Q2: To what extent has the Medee Glossary helped to understand the elements encompassed in the fragment (e.g. tasks, work products, roles)?

**Metric 2:** From 1 (not helpful at all) to 5 (very useful).

Q3: How easy is it to understand the work that should be performed when adopting the fragment? In other words, is it easy to understand the task(s) and steps encompassed in the fragment?

**Metric 3:** From 1 (unclear) to 5 (very clear)

Q4: How easy is it to understand the work product encompassed in the fragment?

**Metric 4:** From 1 (not easy at all) to 5 (very easy).

**Goal 2:** Analyze the MOISE+ Method fragment for the purpose of evaluation with respect to the visibility.

Q5: To what extent the development phase (e.g. analysis, design) during which the fragment is expected to be used is clearly stated?

**Metric 5:** From 1 (unclear) to 5 (very clear).

Q6: To what extent the work product that should be generated by the fragment is clearly stated?

**Metric 6:** From 1 (unclear) to 5 (very clear)

Q7: To what extent the fragment inputs are clearly stated?

**Metric 7:** From 1 (unclear) to 5 (very clear).

Q8: To what extent the development role(s) assigned to the fragment are clearly stated?

**Metric 8:** From 1 (unclear) to 5 (very clear).
Q9: To what extent the MAS aspects involved in the fragment are clearly stated (e.g. MAS component, MAS nature)?

**Metric 9**: From 1 (unclear) to 5 (very clear).

**Goal 3**: Analyze the MOISE+ Method fragment for evaluation purposes with respect to **supportability**.

Q10: How easy is it to navigate in the website that describes the fragment?

**Metric 10**: From 1 (not easy at all) to 5 (very easy).

Q11: To what extent the guidance proposed by the fragment (e.g. examples, whitepapers, concepts) could help task execution and/or work product generation?

**Metric 11**: From 1 (not helpful at all) to 5 (very useful).

### 5.2 Collecting Metrics after Fragments Usage (step 4)

Having defined the goal-driven model, a questionnaire was designed for each of the five MOISE+ fragments. Along with aforementioned goals, questions of interest and metrics, the designed questionnaires asked for additional comments. Furthermore, the participants had inspected these fragments and analyzed them against the MOISE+ literature [15] [15]. Next, they filled out the five questionnaires. Questionnaires involving the Developer viewpoint were filled out by students, while those relating to the MOISE+ expert and Method Engineer viewpoints were filled out by MAS researchers skilled in OC-MAS and familiar with MOISE+ specifications and one of the MOISE+ authors.

Finally, to ensure completeness and consistency, the data provided in these questionnaires were validated through interviews with the students and researchers. The collected metrics are presented in Table 1.

### 5.3 Analyzing the measurement goals (step 5)

This step consisted of analyzing the three measurement goals through the collected metrics. Table 1 (last row) shows a consolidated perspective of such metrics by the three goals, as well as perspectives broken by the five MOISE+ fragment (last column).

<table>
<thead>
<tr>
<th>Frag#</th>
<th>Goal 1 Understandability</th>
<th>Goal 2 Visibility</th>
<th>Goal 3 Supportability</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>4.3</td>
<td>4.4</td>
<td>4.6</td>
<td>4.4</td>
</tr>
<tr>
<td>#2</td>
<td>4.3</td>
<td>4.2</td>
<td>4.3</td>
<td>4.3</td>
</tr>
<tr>
<td>#3</td>
<td>4.0</td>
<td>4.4</td>
<td>4.5</td>
<td>4.3</td>
</tr>
<tr>
<td>#4</td>
<td>4.8</td>
<td>4.7</td>
<td>4.8</td>
<td>4.7</td>
</tr>
<tr>
<td>#5</td>
<td>4.2</td>
<td>4.1</td>
<td>4.6</td>
<td>4.3</td>
</tr>
<tr>
<td>Total</td>
<td>4.3</td>
<td>4.3</td>
<td>4.5</td>
<td>4.4</td>
</tr>
</tbody>
</table>

The metrics regarding Goal 1 – Understandability - have shown that MOISE+ fragments were quite easy to understand (4.3 points in a 1 to 5 scale). However, some
comments stated that the two fragments related to the analysis and design of the organizational specification could be made more understandable if tasks and steps were more explicit about which of the three MOISE+ specifications they were concerned with (i.e. Structural, Functional, Deontic specifications).

Furthermore, this experiment showed that fragments’ elements - like roles, input and output work products - had a quite well visibility (4.3 in a 1 to 5 scale). Nonetheless, some aspects related to Medee development roles were missing, such as role responsibility, while the work products could be more visible if they could also be accessed directly, besides embedded in tasks.

Finally, the metrics related to Goal 3 – Supportability - have shown that MOISE+ fragments offered a suitable collection of examples, whitepapers, and concepts, as well as an easy navigation through the Medee website (4.5 in a 1 to 5 scale). However, some comments suggested that fragments could provide definitions for concepts related to the agent-oriented paradigm to help newcomers.

5.4 Packaging experience for improving fragments (step 6)

This step consisted of describing improving opportunities in a way that such description could be used to manage fragments and/or the Method Repository itself, which is effectively done in the step 7.

Due to paper length limitations, this section describes only a couple of opportunities related to the two fragments concerning the analysis and design of OC-MAS, as well as some improvement related to the Method Repository as a whole.

Opportunities for improving MAS organization analysis and design

— Improve the comprehension about the work to be done, since tasks are mixing up several MOISE+ concepts pertaining to different MOISE+ specification (functional, structural and deontic).
— State in a clear way each one of the MOISE+ specifications should be created or modified through the tasks/steps underpinned by the fragments.
— Recommend the use of MAS User Requirement specification during the design of the MOISE+ organization, as it is recommended during the organization analysis.

Opportunities for improving the Method Repository.

— Offer the definition of concepts related to the Agent-Oriented Paradigm.
— Make development roles characteristics more explicit.
— Make work products characteristics more explicit.

5.5 Managing the Method Repository (step 7)

This step consisted of updating the Medee Method Repository according to the improvement opportunities previously identified. Such an update encompassed manag-
ing two MOISE+ fragments. Also, it involved managing some aspects related to the building blocks underpinned by the Method Repository itself, such as making more explicit the Medee development roles and the Medee work products, and expanding the Medee Glossary by including MAS concepts, as explained in the sequence.

**Updating fragments for MAS organization analysis and design**

It consisted of modifying two fragments, MMF Analyze MAS Organization with MOISE+ and MMF Design MAS Organization with MOISE+, by describing the work required to deal with MOISE+ specifications in a way that each task were focused on one single specification (i.e. Functional, Structural and Deontic). Therefore, the task called MTV Analyze MAS Organization was replaced by three new tasks: MTV Analyze MAS Functional Specification, MTV Analyze MAS Structural Specification and MTV Analyze MAS Deontic Specification.

![Fig. 4. Workflow for MMF Analyzing MAS Organization with MOISE+](image)

As illustrated in Fig. 4, the new task in charge of analyzing the functional dimension of a MOISE+ organization takes a User Requirement (e.g. the one proposed by Tropos) as input and produces the MOISE+ Functional Specification as output. In a similar way, the new tasks in charge of analyzing structural and deontic MOISE+ dimensions produce the homonym MOISE+ specifications as outputs. Furthermore, the improved fragment for analyzing MOISE+ organization was built upon these new tasks and thus clearly states the specification created by each one of its tasks. Moreover, as illustrated in Fig. 4, as soon as a specification is available, it can be used as an input in the next task.
Such an approach promotes the coherence and consistence of the MOISE+ specifications generated by this fragment. A similar approach was adopted to update the MMF Design MAS Organization with MOISE+.

Updating the Method Repository Building Blocks

On one hand, it consisted of modifying the Method Repository navigation tree in order to present the Medee Development Roles and the Medee Work Product Framework in an explicit way, as illustrated in Fig. 5 (right side). In such a way, these elements can be accessed directly, and not only through method fragments.

On the other hand, it consisted of extending the Medee Glossary by creating new concepts in order to facilitate the comprehension of the agent-oriented paradigm main notions. Examples of these new concepts are BDI agents, agent autonomy, and multi-agent systems, as depicted in Fig. 6.

Therefore, after these modifications Medee users should better understand the agent oriented paradigm, as well as easily discover the whole set of MAS development roles and MAS work products currently available in the Method Repository.

Summing up the Medee Method Repository Improvements

During this experiment we have improved several elements of the Medee Method Repository through modifications based on lessons learned. As illustrated in Figs. 4, 5 and 6, such modifications are ready for use since this repository has been updated and the Medee website related pages have been generated again as part of the method repository management procedure.

Therefore, from now on the Medee repository stores a glossary containing concepts that facilitate the comprehension of the agent-oriented paradigm, a navigation tree that presents in an explicit way the MAS development roles and MAS work products available, and method fragments that state in a clear way each one of the three
MOISE+ specifications should be created/updated through the tasks and steps underpinned by them.

Fig. 6. Improving the Medee Glossary

5.6 Repeating the Medee Improvement Cycle

As previously mentioned, we have performed twice some steps of the Medee Improvement Cycle to investigate in which extent the presented improvement were perceived by students and MAS researchers.

This second round took into account a narrower scope, since it concerned mainly the two improved MOISE+ fragments, those related to the analysis and design of organizations. Therefore, this round included the following steps: (i) Setting measurement goals, (ii) Collecting metrics after using method fragments, and (iii) Analyzing the measurement goals.

Table 2. Collected GQM metrics round 2

<table>
<thead>
<tr>
<th>Goal</th>
<th>Frag#1</th>
<th>Frag#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal 1: Understandability</td>
<td>4.6</td>
<td>4.8</td>
</tr>
<tr>
<td>Goal 2: Visibility</td>
<td>4.6</td>
<td>4.6</td>
</tr>
<tr>
<td>Goal 3: Supportability</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td>Total</td>
<td>4.6</td>
<td>4.7</td>
</tr>
</tbody>
</table>

We have used a smaller version of the goal-driven model previously developed, by limiting the objects of study to the Fragments #1 and #2, respectively, MMF Analyze Organization with MOISE+ (Enhanced) and MMF Design Organization with MOISE+ (Enhanced). Thus, metrics were collected through two questionnaires involving the three goals and related questions of interest, and filled out by the same participants.

Table 2 presents a consolidated perspective of collected metrics by the three goals (last row), and perspectives broken by the two MOISE+ fragments (last column). Just by looking at the quantitative aspects one may think that improvement was marginal.
Nevertheless, the comments were very important to evaluate the improvement perception. Participants said that their understanding about MOISE+ work products increased a lot, as well as the steps that must be followed to execute the tasks involved in their generation.

6 DISCUSSION

As described in the course of this paper, the Medee Improvement Cycle allows to continuous evolving MAS methods and fragments, taking into account a set of quality attributes, such as understandability, visibility, supportability, acceptability and robustness. We have shown through the case study how to apply this cycle to evolve fragments through their usage, instead of assuming that we have already the definitive version of them from the beginning.

Furthermore, our approach encompasses several aspects that constitute advancements in the way we can improve methods and fragments for AOSE. Firstly, it is concerned with evolving both MAS methods and fragments based on lessons learned, and not only evaluating and comparing them, as proposed in [4] [10] [12] [23]. Our goal is to continuously improving method and fragments, instead of comparing them quantitatively.

Secondly, it provides an integrated approach to update fragments, methods, and other Method Repository building blocks, based on industry standards for describing methods [19]. As illustrated in the case study, fragment improvements were easily incorporated to the method repository for immediately reuse, which could involve composing new situational method or changing existing ones.

Although no one is able to ensure that has the best development method for a given project situation neither in traditional software engineering field in general nor in AOSE field, the Medee Improvement Cycle is an approach that could help achieving such a goal.

To the best of our knowledge, in the AOSE field there is no currently such a broad approach for evolving methods, fragments, and method repository building blocks in an integrated way.

7 CONCLUSIONS

Situational Method Engineering is an engineering discipline where methods are built on demand according to the project characteristics for immediate use and for further reuse. In such a context, method improvement is strongly desirable since it allows lessons learned from method usage to give rise to a continuous process for evolving method based on quality attributes, like understandability, supportability, visibility, and robustness, among others.

In this paper we have presented the Medee Improvement Cycle, a continuous process improvement approach to deal with MAS methods. Our approach may be applied for both whole methods or single fragments, and uses industry standards [19] for
evolving them in a seamless manner. Moreover, it offers a controlled and disciplined way to learn from experience. Therefore, it can be used for reinforcing the development of MAS in the academy as well as in the software industry.

We show its applicability by presenting in which manners some fragments sourced from a well-established AO model, the MOISE+ model, could be improved towards a better understandability, visibility and supportability. In a few words, at the end of the improvement process we had facilitated the understanding of MOISE+ concepts as well as enhanced the visibility of the step-by-step in which MOISE+ specifications could be produced during the development of an OC-MAS project. Also, we have updated the Method Repository turning it easily to be navigated.

8 ACKNOWLEDGMENTS

We would like to thank the participants of the experiment reported in this paper. Anarosa A. F. Brandão is partially supported by grant #010/2640-5, São Paulo Research Foundation (FAPESP). Jaime S. Sichman is partially supported by CNPq, Brazil. Sara Casare is partially supported by IBM Brasil.

9 REFERENCES

Towards Process-Oriented Modelling and Creation of Multi-Agent Systems

Tobias Küster, Axel Heßler, and Sahin Albayrak
DAI-Labor, Technische Universität Berlin, Germany
tobias.kuester@dai-labor.de

Abstract. Different ways of integrating business processes and agents have been proposed, but using restricted process models or targeting only single agents, none of them is truly convincing. Nevertheless, business processes have many notions in common with agents and would be well suited for modelling complex multi-agent systems. In this paper, we combine concepts of two existing approaches to a mapping from business process diagrams to readily executable agent components. The results are well-structured and extensible, and at the same time account for nearly the entire expressiveness of the process modelling notation.

Keywords: Technological, Methodological

1 Introduction

In recent times, different approaches for modelling agents and multi-agent systems using business process diagrams and related notations have been introduced (e.g., [8,16]). However, none of these approaches is really compelling. Often, very simple workflow models are used, or if a more expressive process modelling notation is chosen, then only a limited subset of the language is covered. Furthermore, usually only single agents are targeted, while interactions between agents – which could very well be modelled using many process notations – are not regarded.

This is unfortunate, since process diagrams share many concepts and abstractions with multi-agent systems – in particular sophisticated notations such as the Business Process Model and Notation (BPMN) [18]. Those notations can be used for modelling the intertwined workflows of different participants in a process, as well as their interactions and communication, or their reactions to external events. The focus lies much more on what has to be done and less on how it is implemented. Thus, despite the shortcomings of existing approaches, BPMN and related notations appear to be very well suited for modelling agents and particularly multi-agent systems.

In this paper we take a look at some of the existing approaches – particularly the WADE extension to the JADE agent framework [8], and a mapping from BPMN to the agent-oriented scripting language JADL [16] – and combine the strong sides of both into a new approach. The result is a mapping from BPMN diagrams to behaviour components for the JIAC multi-agent framework [17].
In this way, the core components of the agents can easily be modelled with
and generated from BPMN process diagrams. Thus, we are helping to close the
gap between design and implementation of multi-agent systems [6]. The resulting
Java classes are similarly structured and as extensible as those of WADE, but
they exhibit the expressiveness of BPMN, including communication between
agents and event-handling, both as part of the workflow and for triggering the
process.

The remainder of this paper is structured as follows: First, we discuss some re-
lated work, most notably the WADE framework and the mapping from BPMN to
JADL, with their benefits and shortcomings. Then, in Section 3, we take a closer
look at BPMN and the JIAC framework, and how they fit together. Thereafte-
er, we describe how BPMN processes can be mapped to semantically equivalent
JIAC Agent Beans (Section 4), and how the transformation was implemented
(Section 5). In Section 6, the mapping is illustrated using an example, before we
finally wrap up and discuss our results.

## 2 Related Work

Different approaches for combining process modelling and agent-oriented soft-
ware development have been devised. Some using BPMN, others using sim-
pler notations; some using code generations, others employing interpreting ap-
proaches. Each of those have their strengths and weaknesses.

In the following we discuss several works that are highly relevant to the ap-
proach described in this paper: The original mapping from BPMN to BPEL, a
mapping from BPMN to JIAC’s scripting language JADL, the WADE frame-
work, mapping workflows to JADE behaviours, and GO-BPMN, a combination
of BPMN and goal hierarchies.

### 2.1 Transformation from BPMN to BPEL

One of the motivations for developing BPMN was to provide a standardised
graphical notation for BPEL, the Business Process Executable Language. Con-
sequently, a mapping from BPMN to BPEL is part of the BPMN specifi-
cation [18, Chapter 14], and a number of alternative or extended mappings have
been proposed by various other authors (see for example [19]).

In many aspects, the mapping is very straightforward: Each pool is mappe-
d to a BPEL process (which can be deployed as a Web service), and the several
events and activities within are mapped to the workflow of the process. The process is
made up mostly of Web service calls, assignments and flow control, but can also
contain, e.g., event handling based on timing and incoming messages. Given a
sufficiently detailed BPMN diagram, the resulting BPEL process can be readily
executable.

Still, there are enough elements in BPMN for which no mapping to BPEL
is given. Thus, while BPMN was created with the mapping to BPEL in mind,
it is not just a visualisation for BPEL but a distinct, self-contained language –
and in fact more expressive than BPEL itself. Among the elements that are not mapped to BPEL are somewhat obscure elements such as the ad-hoc subprocess, or the complex gateway, but also many types of events and tasks.

### 2.2 Transformation from BPMN to JADL

In prior work of mapping BPMN to agents [12], JIAC’s service-oriented scripting language JADL [13] was used as the target of the transformation.

Being conceptually close to BPEL, the mapping is similar, and the process can be mapped very directly to different language elements of JADL. For instance, like BPEL, JADL has dedicated language elements for complex actions such as invoking other services, or for sending and receiving messages, making the generated code compact and easy to comprehend.

Each pool in the BPMN process is mapped to a JADL service, and the service’s input parameters and result types are derived from the pool’s start- and end events [16]. Further, for each start event, a Drools rule is created, starting the respective JADL service on the occurrence of the given event (e.g., an incoming message, or a given time). Also, for each participant in the BPMN process, an agent configuration file is created, setting up the individual agents, each equipped with an Interpreter Bean and Rule Engine Bean, together with the generated JADL services and Drools rules.

Alternatively, the JADL services and rules created from the BPMN processes can be added to a running JIAC agent, thus dynamically changing its behaviour.

### 2.3 WADE: Workflows for JADE

A different approach, from which some of the concepts in this work have been drawn, is WADE (Workflows and Agents Development Environment), which is an extension to the JADE multi-agent framework [2]. Using WADE, certain aspects of the behaviour of a JADE agent can be modelled using a simple workflow notation [8,7]. The workflows basically consist of only two elements: Activities and Transitions.

Using the Wolf tool [9], JADE behaviour classes can be generated from those workflow models. The generated Java classes show a clear distinction between the workflow (the order of the activities, together with conditions and guards) and the several activities. Each of them is mapped to an individual Java method that can either refer to existing functionalities or be implemented by the developer. Using this separation, generated workflows can safely be altered or extended.

However, the expressiveness of WADE is restricted by the simplistic workflow notation, which allows only the most basic workflows to be modelled. While the transitions can be annotated with guards (conditions), it seems impossible to model parallel execution and synchronisation, let alone more advanced concepts such as event handling or messaging. In fact, each workflow diagram covers only the behaviour of an isolated agent; to our knowledge, interactions between agents can not be modelled.
Later, WADE has been extended to provide better support for long-running business processes and event handling [3].

2.4 GO-BPMN and Go4Flex

In GO-BPMN (Goal-oriented BPMN), process models are combined with a goal-hierarchy and executed by agents [10]. The authors highlight the high flexibility of the system, and the prospects of parallelisation, but they also write that testing the system is difficult due to possible side-effects of the processes regarding other goals [5].

The individual processes (the “leafs” in the goal hierarchy) are described as BPMN processes; however, only a subset of BPMN is used. Particularly, each diagram shows only a single pool, and thus, as in the case of WADE, no communication and interaction can be modelled, but just the behaviour of a single agent. While using goals for connecting the individual processes is quite promising, in our opinion process diagrams can more efficiently be used at a higher level of abstraction, e.g., for providing an overview of the system as a whole, instead of for isolated behaviours of individual agents.

A similar approach is Go4Flex, or GPMN [4]. Like GO-BPMN, Go4Flex uses goal hierarchies with BPMN processes being the leafs. Both the goals and the processes are interpreted by Jadex agents [21]. The authors also present a mapping from FIPA/AUML interaction diagrams [1] to BPMN processes [20].

3 A Closer Look at BPMN and JIAC

As we have seen, there are numerous approaches, but to the best of our knowledge none of them makes full use of the expressiveness of BPMN or a similarly powerful process notation. This is unfortunate, since BPMN provides many notions that could very well be used for modelling high-level multi-agent behaviour.

In the following, we will take a closer look at the BPMN language and the JIAC agent framework, being the domain and co-domain of the mapping discussed in the next section of this paper.

3.1 BPMN

The Business Process Model and Notation [18] is a workflow representation that can be used both as a description language for real-world processes, and as a high-level modelling language for computer programs. It can be seen as a combination of UML’s Activity Diagrams and Sequence Diagrams, depicting both the actors’ internal processes and their interactions. An example diagram is shown in Figure 1.

BPMN diagrams can be understood at three levels of abstraction:

1. The diagrams are made up of a few easily recognisable elements, i.e., events (circles), activities (boxes) and gateways (diamonds), connected by sequence- and message flows and situated in one or more pools.
2. These basic elements are further distinguished using sets of marker icons, e.g., message, timer, and error events, or parallel and exclusive gateways.

3. Each element features a number of additional attributes that are hidden from the diagram and contain most of the information that is necessary for automated code generation, e.g., properties and assignments.

Consequently, the essence of a BPMN diagram is easily understood by all business partners, including those who have great knowledge in their domain but little understanding of programming and multi-agent systems. At the same time, BPMN diagrams provide enough information for the generation of executable programs.

A variety of notational elements make BPMN diagrams well suited for the design of distributed systems in general and multi-agent systems in particular. The process diagrams are subdivided into pools, each representing one participant in the process. Using message flows for communication between pools, even complex interaction protocols can be modelled clearly. Further, the notation supports features such as event- and error handling, compensation, transactions and ad-hoc behaviour.

While the semantics of some elements of BPMN – particularly those not covered in the official mapping from BPMN to BPEL [18, Chapter 14] – are not clearly defined, there is an increasing number of approaches describing the semantics of BPMN using, e.g., Petri nets [11], and version 2.0 of the specification made things clearer, too.

The reason why Petri nets are not used in the first place is: While Petri nets have very clear semantics, and basically everything can be expressed as a Petri net, some high-level constructs that are directly supported by BPMN (e.g., event handling and cancellation) would require huge, incomprehensible
Petri nets. Thus, while Petri nets are well suited for the formal specification of a workflow, they are not the best choice for modelling.

BPMN is neither the first process modelling notation, nor will it be the last. However, given its high level of adoption in practical process modelling [22], it has proven to be a good choice for modelling distributed computing systems, combining a high-level overview of the system with all the necessary details about its implementation and execution.

3.2 JIAC

JIAC V (Java-based Intelligent Agent Componentware, version 5) is a multi-agent development framework and runtime environment [17]. Among others, JIAC features message-based inter-agent communication, tuple-space based agent memory, transparent distribution of agents and services, and provides support for dynamic reconfiguration in distributed environments, such as component exchange at runtime. Individual JIAC agents are situated within Agent Nodes, i.e., runtime containers, which also provide support for migration. The agents’ behaviours and capabilities are defined in a number of so-called Agent Beans that are controlled by the agent’s life cycle. The different structures and elements of a JIAC multi-agent system are shown in Figure 2.

![Fig. 2. Components of a JIAC multi-agent system and individual agents.](image)

Each JIAC agent is equipped with a Communication Bean, allowing agents to send and receive messages to and from other agents or groups of agents (multi-casting to message channels). The messages are not restricted to FIPA\(^1\) messages and can have any serialisable data as payload. Other commonly used Agent Beans are the Rule Engine Bean, integrating a Drools\(^2\) rule engine into the agent’s memory for reactive behaviour, and the Interpreter Bean, providing an interpreter for the service-oriented scripting language JADL [13].

Besides these and other predefined Agent Beans, the programmer is free to add application-specific Beans to the agent. Each such Agent Bean can

---
\(^1\) Foundation for Intelligent Physical Agents: [http://www.fipa.org/](http://www.fipa.org/)

\(^2\) JBoss Drools: [http://www.jboss.org/drools/](http://www.jboss.org/drools/)
— implement a number of *life-cycle* methods, which are executed when the agent changes its life-cycle state, such as initialized, or started,
— implement an *execute*-method, which is called automatically at regular intervals once the agent is running (i.e., cyclic behaviour),
— attach *observers* to the agent’s memory, being called, e.g., each time the agent receives a message or its world model is updated, and
— contribute *action*-methods, or services, which are exposed to the directory and can be invoked by other agents or other Beans of the same agent.

Using these four mechanisms, it is possible to define all of the agents’ capabilities and behaviours. For details on programming JIAC Agent Beans, we refer readers to the JIAC Programmers’ Manual [14].

4 A Mapping from BPMN to JIAC Agent Beans

While the mapping from BPMN to JADL is well suited for modelling high-level behaviour or services, traditional JIAC Agent Beans were still advantageous — and often necessary — for defining the better part of the agent’s behaviour, for instance when it comes to the integration with user interfaces or external libraries. Consequently, complementary to the mapping to JADL, a mapping to JIAC Agent Beans was developed [23].

The mapping is conceptually close to WADE: Each Pool in the BPMN diagram is mapped to one Agent Bean, i.e., a Java class, with one method for the workflow, and one method for each individual activity of the process. The *workflow method* acts as an entry point to executing the process, while the several *activity methods* are invoked by the workflow method in accordance with the ordering of the activities in the process.

In the following, we will describe the several aspects of the mapping in detail. Finally, we will briefly illustrate how process modelling can be integrated into the overall development method.

4.1 Workflow Method

The workflow method is made up of calls to several activity methods, being arranged into sequences, if-else statements and loops. While this requires the process to be structured properly (see Section 5), the result is structured and understandable, resembling manually written code, i.e., using conditions and loops instead of *goto*-like successor-relations. Thus, if necessary, the generated code can still be easily extended or altered by hand.

At the same time, BPMN allows for much more expressive workflows to be modelled, compared to the rather minimalistic workflow notation used in WADE. In particular, the following concepts of BPMN are covered by the mapping:

---

3 In the following, we will use the term “workflow” for the order the individual activities are executed in the process, and the term “process” for the whole ensemble of activities and their ordering, events, variables, etc.
- Parallel execution (BPMN’s AND-Gateway) is mapped to multiple threads being started and joined.
- Subprocesses (composite activities) are mapped to internal classes following the same schema as the main class, with workflow- and activity methods for the activities embedded into the subprocess.
- Event handler (intermediate events attached to an activity) are also mapped to threads, running concurrently to the thread executing the activity itself, and interrupting this thread in case the respective event occurs.
- The same pattern is applied to event-based XOR-gateways; in this case the main thread will wait until one of the events has been triggered.

4.2 Properties and Assignments

BPMN specifies a number of non-visual attributes, such as properties (i.e., variables) and assignments. Properties can be declared in the scope of whole processes or individual activities (both atomic tasks and composite subprocesses). When declared in the scope of a process or subprocess, the property is visible to all elements (transitively) contained therein.

Accordingly, properties are mapped to Java variables in different scopes in the Agent Bean, reflecting their visibility in the BPMN diagram. Properties of the process are mapped to variables in the scope of the Agent Bean class, properties of a subprocess to variables in the scope of the embedded subprocess class, and properties of an activity to local variables in the scope of the activity method.

Assignments are always bound to an activity or event, and are included in the respective activity method. In BPMN, assignments can have an assign-time of either ‘before’ or ‘after’, determining whether the assignment has to be applied before or after the actual activity is executed (see below).

4.3 Activity Methods

The several activity methods have neither parameters nor a return value and always follow the same schema:

1. Properties: First, for each property in the scope of the activity one Java variable is declared, using the respective data type.
2. Start Assignments: Then, assignments of the activity with assign-time ‘before’ are applied, e.g., for setting the input parameters of a service call.
3. Activity Body: Now, the code corresponding to the actual activity is carried out, e.g., invoking a service, sending a message, or executing a user-defined code-snippet.
4. End Assignments: Finally, assignments with assign-time ‘after’ are applied, e.g., for binding the return value of a service call to a variable.
5. Loop: If the activity’s loop attribute is set, the content of the activity method is repeated in a loop as long as a given condition is satisfied.
Similar to the mapping to JADL, we can make use of JIAC’s communication infrastructure, by mapping message events and send and receive tasks to sending and receiving JIAC messages, while service tasks are mapped to the invocation of a JIAC action (i.e., a service). Script tasks allow the developer to attach a custom snippet of Java code to the task. Further, timer events are mapped to a temporary suspension of the execution.

There are more types of tasks and events in BPMN, for which no mapping has been devised yet, but these are the most common and important ones. Elements that will be covered in the near future include the rule event, evaluating a given Java condition, as well as the user task, presenting a generic input dialogue to the user.

4.4 Event Handler

As mentioned above, event handlers (i.e., intermediate events attached to an activity’s boundary) are mapped to threads running in parallel to the actual activity, interrupting it in case the event has been triggered. To realise this behaviour, the activity itself is wrapped in another thread, and a reference is passed to the event handler thread, running in a loop and periodically checking whether the respective event has occurred (e.g., whether a message has arrived, or whether a given time has passed). If so, a marker flag is set and the activity thread is interrupted.

In the workflow method, both threads are started, and the activity thread is joined. Finally, when the activity has been completed or aborted, the event handler thread is stopped and the workflow is routed accordingly to whether the event handler has been triggered or not.

4.5 Start Events and Starter Rules

Finally, the processes’ start events have to be mapped to mechanisms for starting the process on the occurrence of the respective events. In the mapping to JADL, a number of Drools rules are created for this purpose. Using Agent Beans, these ‘starter rules’ can be integrated directly into the code, making use of the mechanisms introduced in Section 3.2.

- If the process has a start event with unspecified type, or none type, then the workflow method is invoked in the Agent Bean’s doStart() method (one of the life-cycle methods), being called when the agent is started.
- For a timer start event, the Agent Bean is given an execute() method, regularly checking the current time against the time the process was last started, invoking the workflow method at a given time or interval.
- A message start event results in a message observer being attached to the agent’s memory when the Agent Bean is started, which will then invoke the workflow method every time a matching JIAC message is received.
Finally, in case of a service start event, the workflow method is marked with the annotation \texttt{@Expose}, exposing the workflow method as a JIAC action to be discovered and invoked by other agents.\footnote{There is, as such, no service start event in BPMN. We use this term to distinguish message start events, where the message is in fact a service request.}

Besides creating these mechanisms, a service start event also results in the workflow method’s input parameters being updated to correspond to the specified service parameters. Analogously, a service end event results in the workflow method’s return value being set accordingly.

### 4.6 Development Method

In previous work, we presented a method for integrating process modelling into the overall multi-agent system development cycle\footnote{[16]}\cite{16}. While this was aimed at the mapping from BPMN to JADL, most of the ideas and concepts can be carried over to the mapping to JIAC Agent Beans as well.

In a nutshell, we see process modelling as the next step after use case analysis. For each of the previously identified use case diagrams, one BPMN process diagram is created, holding one pool for each of the actors involved in the respective use case. Those diagrams should describe the behaviour and particularly the interaction of the several roles at a relatively high level of abstraction, illustrating the system behaviour without cluttering the diagrams with algorithmic details. The mapping then translates the pools to behaviours, encapsulated into Agent Beans, while each of the actors corresponds to a different agent role exhibiting those behaviours. Next, the generated JIAC Agent Beans can be extended with additional code not suited for inclusion in the process diagrams, and the agent roles are aggregated to concrete agents and the multi-agent system is set up.

### 5 Implementation

The first version of the mapping was implemented in the course of a diploma thesis\footnote{[23]}\cite{23} as an extension to the BPMN editor VSDT (Visual Service Design Tool). The VSDT was developed with the goal in mind, to provide transformations from BPMN to diverse executable languages\footnote{[15]}\cite{15}. It also allows for the import of existing services, simulation/interpretation of process diagrams, and the generation of descriptive texts in written English from the process. Besides being a BPMN editor, the VSDT can also be used for creating the use case diagrams for connecting the different process diagrams that make up the entire system.

The first step in mapping BPMN to Agent Beans – or any structured programming language – is to structure the process graph to a tree of sequences, decision blocks, loops, etc. To this end, a number of pattern matching rules are used,
identifying different structures in the workflow and substituting them with dedicated structural elements. This functionality is provided by the VSDT’s transformation framework and can be reused for the different target languages [15]. Thus, only the actual mapping of individual process elements to fragments of Java code, as specified in the previous section, had to be implemented.

This element mapping has been separated into two stages. First, the structured process model is translated to an intermediate model, being a high-level representation of the structure of a JIAC Agent Bean. This is done by traversing the process model, which now has a tree-structure, and thereby creating and assembling the respective elements of the Agent Bean model. Then, this model can be translated straightforwardly to executable Java code using a number of templates for the JET framework.5 Using JET and JMerge, parts of the generated Agent Bean code can safely be modified and merged in case the process model changes and has to be re-generated.

6 Example

In this section we will illustrate several aspects of the mapping by means of the simple example diagram from Section 3, shown in Figure 1.

The BPMN diagram consists of two pools, each representing an agent role: Client, and Taxi. The client’s process is exposed and started as a service, expecting a customer ID, current location, desired destination and time of arrival, and returning the ID of the taxi selected for the tour, if any.

The interaction between the two starts by the client sending a request (customer ID, location, destination, desired time of arrival) to all available taxis, which evaluate the request and decide whether to accept it. If so, they send a response (taxi ID, estimated time of arrival, price) back to the client. Meanwhile, the client enters a looping subprocess, listening to responses and memorising the best response, until after 30 seconds the subprocess is interrupted by the attached timer event. The client then sends a notification to the selected taxi. The taxis listen to incoming message, either preparing to pick up the guest if the notification is received, or ending the process after waiting for a few more seconds. Note that the several properties (variables) and assignments are not visible in the diagram.

The resulting Agent Bean for the Client role is shown in Figure 3, along with the client’s part of the process diagram for reference. The entire code was automatically generated and only slightly shortened to improve readability and to better fit into the figure. The full code also contains JavaDoc comments (not shown here) with descriptions to the bean class and each of the activity methods, taken from the description attribute of the respective BPMN elements.

As can be seen, the control-flow of the process is reflected in the workflow() method, which is also exposed as a JIAC action, or service. The workflow method is dominated by the threads for running the subprocess and the attached event

5 JET (Java Emitter Templates) is part of the Eclipse Model To Text (M2T) project: http://www.eclipse.org/modeling/m2t/
Fig. 3. Example: Taxi Request Service. Corresponding parts in the process diagram and the code are numbered correspondingly.

handler, but also contains an if-else-statement for the gateway at the end of the process. The activities *send request* and *notify taxis* are mapped to two similar methods for sending JIAC messages to the specified message groups.

The code for, e.g., sending and receiving messages is quite extensive, and there are several components, such as the event handler classes, that are needed again and again for different workflows. Consequently, these parts are provided by the superclass *AbstractWorkflowBean*, allowing the generated code to be much more compact and readable.

The subprocess is mapped to the inner class *WaitForReplies_Sub*, also forming a new variable scope for its properties. The class follows the same schema as the outer workflow class. It features another workflow method (*run()* in this case) and three activity methods, most notably the *receiveResponse* method, where the client checks its memory for messages arriving at the specified message...
group channel. In accordance with the loop-condition of the original subprocess, the content of the workflow method is executed in an infinite loop. The subprocess itself is run in a thread, which will eventually be interrupted by the event handler thread, thus breaking out of the loop.

The Agent Bean for the Taxi role is similarly structured, and thus is not shown here. The main difference is that its workflow method is not exposed as an action, but is invoked by a memory observer listening for the request messages sent by the client role. The observer is attached to the agent’s memory in the doStart() method (one of the life-cycle methods, which is started when the agent is started). The workflow method itself is rather straightforward, with an if-statement representing the first gateway, and an event-handler for the second. The logic for the evaluate request task can either be provided via the task’s script attribute, or it can be implemented in the generated Java code.

6.1 Discussion

Using the domain-specific scripting language JADL, agent behaviours can be expressed in a very compact and readable way, but the overall expressiveness (e.g., the supported event types) is limited by the scripting language. JIAC Agent Beans, on the other hand, have the full expressiveness of the Java language at their disposal. Thus, basically everything that can be modelled in a BPMN diagram can be mapped to an Agent Bean.

While the resulting workflow method for complex processes can become somewhat bulky – particularly if event handling is used – its structured form as well as the separation into workflow methods and activity methods keeps the resulting code reasonably clear. Like in WADE, individual activity methods can be altered or extended without risk of losing the changes after the code is generated anew. The reason why this is important is that while BPMN is well suited for high-level behaviour, graphically modelling low-level algorithms and such would be too laborious. This way, those can be added to the generated code.

One potential problem might be raised by the extensive use of Java threads for event handling. We are currently investigating ways of integrating the event handling into the agent’s main thread. Another alternative would be to move away from the current workflow methods towards a more interpreter-like approach, memorizing the current state of the process and executing one activity method in each step of the agent’s execution cycle. Particularly for long-running processes this might be beneficial.

Regarding the high expressiveness of the generated Agent Beans and the good performance of compiled Java code when compared to the interpreted JADL scripts, the mapping from BPMN to JIAC Agent Beans is suited best for modelling and generating core components of the multi-agent system, while the mapping to JADL is of much use for creating dynamic behaviours and services to be deployed and changed at runtime.
7 Conclusion

In this paper, we have presented an approach of creating multi-agent systems from process models, combining the mapping from BPMN to JADL [16] with ideas borrowed from WADE [8]. The result is a transformation from BPMN process diagrams to JIAC Agent Beans, generating one method for the workflow as a whole, and one method for each individual activity. The resulting Agent Bean classes are highly expressive and at the same time well structured and readable. Being based on the wide-spread Business Process Modeling Notation, the process diagrams are easily to understand and the mapping also supports important aspects such as communication and interaction and event handling, which are particularly suited for being modelled visually.

Of course, it depends on the application to be developed whether process modelling in general and BPMN in particular are appropriate ways for designing the system. Still, using the mapping proposed and exemplified in this paper, it is possible to model complex and distributed multi-agent systems by means of BPMN and to generate readily executable agent behaviours from the process diagrams. Also, while we decided to use JIAC in this work, the bulk of the mapping could be applied to other agent frameworks, as well.

7.1 Future Work

While the mapping can already be used for generating useful agent behaviours, it is not yet completed. First, there are still aspects of BPMN that are not covered by the mapping, such as some of the less common event types. Second, there are aspects of agents that cannot yet be modelled adequately with BPMN.

One such issue that we want to tackle in the future is the modelling of goals and other kinds of dynamic behaviour by means of BPMN. Without those, the resulting agent systems, strictly following the process diagram, are rather procedural and inflexible. One promising approach is to use the ad-hoc subprocess for this task, but this is still work in progress.

Complementary to the transformation to JIAC code, we are currently working on a process interpreter agent bean. Similar to the JADL interpreter agent, this will allow to pass processes to the agent at runtime and to have that agent execute one or more of the roles in that process. Without the additional layer of abstraction of the scripting language, this approach is expected to have the same expressive power as the generated JIAC bean while at the same time being more dynamic.

References


Abstract. Although we have many agent oriented methods, organizational and environmental system dimensions are not yet analysed and implemented as first class entities. Due to evolution of the development platforms, we are able to consider these dimensions in all the development phases. In this paper we present Prometheus AEOlus method, that allows the integrated development of three systems dimensions: agent, environment and organization. This method was based on both Prometheus method and JaCaMo framework and aims to reduce the conceptual gap between the analysis and implementation phases.

Keywords: AOSE, Organization, Environment, Prometheus, JaCaMo, code generation

1 Introduction

As proposed in [10], multi-agent system (MAS) can be formed by four dimensions: agents, environment, protocols of interaction, and organization. However, the methods\(^1\) provided by the agent oriented software engineering (AOSE) field focus essentially on the agent dimension. In these methods, some environment and organizational concepts are used mainly in the early stages to clarify the problem to be solved. Along the method, these concepts are analysed and, in the implementation phase, they disappear and are replaced by agent program primitives. For instance, methods like Prometheus [14] uses the organizational concept of role to describe part of the agent behaviour. During the analyses phase roles are grouped to give rise to the agents. However, the roles will not be properly coded, but the agents originated by a group of these roles will.

We have thus a gap problem between analysis and development during the phases of AOSE methods. One reason for this gap is that the most used development platforms (i.e. Jade [1] and Jadex [4]), do not deal with organizational and environmental concepts as first class entities. However, we have now programming platforms that consider organization and environment as first class entities.

\(^1\) We use method instead methodology as suggested by [5].
entities, like the JaCaMo [2] and Janus [11] frameworks, and thus this gap could be reduced. Therefore AOSE methods which also deal with these concepts as first class entities can be improved.

Aiming the code generation for this four dimensions, we developed the Prometheus AEOlus method. Prometheus AEOlus is an extension to Prometheus method, in which we included concepts to improve the modelling and code generation of the environment and organization dimensions. In this paper we present the main concepts related to Prometheus AEOlus method. The paper is organized as follows: some AOSE methods are analysed and a state of the art discussion are introduced in Section 2; in Section 3, we present the technologies used to develop Prometheus AEOlus method; in Section 4 we present the concepts and the metamodel defined for Prometheus AEOlus; in Section 5 we show how these concepts are considered in the method; in Section 6 we present some guidelines used to translate these specifications into code; and in Section 7 we briefly discuss the experiments performed to test the method and some future works.

2 State of the Art

Given the existence of many methods, we select some of them in order to identify how they deal with organizational and environmental concepts. The selected methods are well-know, largely used by AOSE community and provide tools which allow code generation from the specification. We selected ASPECS [6], Ingenias [15], O-MaSe [9], PASSI [7], and Tropos [13].

As presented in Table 1, these methods basically deal with two organizational concepts: goals and roles. Goals are used to define the overall system behaviour. Roles are specified to achieve these systems goals, and each role defines a part of an agent behaviour. Some of them also use the group concept, that allow the roles to be structured in coherent sets.

Concerning the environment, excepting ASPECS that do not handle any environmental concept, these methods deal with the concepts of actions, perceptions and two kinds of external entities: actors, which represent users or other systems; or resources, which are external objects or tools used by agents. Actions and perceptions are analysed by the agent point of view, that is, we can specify an action performed from the agent without caring about what these action changes in the environment and, in the same way, we specify a perception received by the agent without caring about how this perception was generated. An actor is an external entity that can perform some operations in the system, and a resource is an external entity in which an agent can perform an action. In both these cases, the method does not address how these events and actions are produced outside the agents.

Moreover, each method supports a tool that allows the translation from the specifications to code in a specific language. Essentially JADE, Jack and Jadex languages are used and, typically, most of system functionality must be coded by the developer since they generate just a skeleton code in the target language.
Table 1: Organizational and environment concepts used by AOSE methods.

Furthermore, no commonly target language allows environment and organization implementation\(^2\). The Janus platform used by ASPECS is one case that allows the implementation of some organizational concepts. Janus was indeed developed specifically to deal with the very concepts of ASPECS.

Thereby, environmental and organizational concepts are just used in these methods to support the agent analysis and, even the organizational ASPECS concepts, are not used considering more detailed models of these dimensions. Despite the existence of tools that allows the code generation, most used target platforms does not support these concepts implementation.

3 Background: Prometheus and JaCaMo

Two main technologies are used in Prometheus AEOUs development: the Prometheus method and the JaCaMo framework. Prometheus [18] is a method that proposes a detailed process for specifying and designing agent oriented software systems. Prometheus method defines a range of structured work products, graphical or textual, that are produced along three developments phases: system specification, architectural design and detailed design. The first phase, system specification, focus on identifying goals and basics functionalities of the system. The architectural design phase uses the work products produced in the previous phase to determine which agent types the system will contain and the interactions among them. The last phase, detailed design, focus on the internals of each agent to specify how they will accomplish their tasks. Due to its maturity

\(^2\) Although Jack has a specific package with organizational concepts, the method do not use this package in the code generation process.
concerning the agent design and analysis, Prometheus was used as the starting point for the Prometheus AEOlus method, in which the environmental and organizational analyses and design are improved.

To improve these dimensions in Prometheus, we decided to use some concepts provided by JaCaMo framework. JaCaMo [2] is a framework for multi-agent programming that combines three separate technologies: the Jason language for programming autonomous agents; the CArtAgO framework for programming environment artifacts; and the Moise organizational model for programming multi-agent organizations. JaCaMo allows the integrated development of these three dimensions (agent, environment and organization) using specific concepts and abstractions for each one.

In the agent dimension, JaCaMo uses abstractions inspired by the BDI architecture, and implemented in the Jason programming language. A Jason [3] agent is an entity composed of a set of beliefs, goals, and plans, and it is able to perform a set of actions. These can be external actions, which change the environment, or internal actions, which change only the internal state of the agent.

In the environment dimension, JaCaMo uses the CArtAgO [16] framework. CArtAgO is based on the Agents and Artifacts (A&A) model [17] and it allows the programming of software environments. Such environments are composed of one or more workspaces, which one composed of a set of artifacts. Artifacts are tools or resources and they provide sets of operations, that can be used by the agents, and observable properties and observable events, that can be perceived by the agents. Observable properties can be updated by the operation execution likewise the observable events are specified by it.

Finally, in the organizational dimension, JaCaMo uses the Moise [12] organizational framework. Moise specifies i) a structural specification, that points out the roles within the organization. Roles define the agent expected behaviour in the system and they can be arranged into groups and subgroups; ii) a functional specification, where the relation among organization’s goals, called social scheme, and sets of goals the agents can commit to, called missions; iii) and the normative specification, that binds roles to missions through norms.

Since our objective are both to allow the code generation and to reduce the gap between the analyses and development stages, we decided to use JaCaMo concepts to improve the Prometheus development process to ensure that all the concepts used during the design and analysis stages will be the same used in the implementation stage. To the best of our knowledge, JaCaMo is the first approach that allows the integrated development of these three dimensions (agent, environment and organization).

4 Prometheus AEOlus Metamodel

The Prometheus AEOlus metamodel was defined by the union of Prometheus [8] and JaCaMo [2] metamodels. To merge these metamodels is not a straightforward process, since they are developed in different projects with distinct ob-
jectives. We initially have thus to compare both metamodels analysing which concepts will be used in the final metamodel. Figure 1 presents this first integrated metamodel with Prometheus and JaCaMo concepts.

![Fig. 1: Prometheus AEOlus preliminary metamodel](image)

Starting with the Prometheus metamodel, the following concepts were included for the environment dimension: *Workspace, Artifact, Operation, ObservableEvents, and ObservableProperties*. For the organization, the included concepts are the following: *Role, Group, Goal, Norm, Mission, and SocialScheme*. We also included some agent concepts which are important to best align JaCaMo and Prometheus metamodels: *ExternalAction, InternalAction, and TriggerEvent*.

We then identify related and conflicting concepts. Related concepts are concepts that have similar meanings, hence only one of them is needed in the metamodel. That is the case of actions, since when an agent perform an external action it is executing an operation in an environmental artifact. Thus the agent *ExternalAction* concept has the same meaning as the environment concept *Operation*. In the same way, when an artifact updates an observable property or generates and observable event in the environment, the agent will receive it as a perception, and both *ObservableProperty* and *ObservableEvent* environmental concepts are related to the agent concept *Percept*. The *Actor* concept is also related to the *Artifact* concept, since *Artifact* also represents all the external entities interacting to the system. The JaCaMo organizational concept of *Role* was clearly related to the *Role* concept from Prometheus. Similarly, the JaCaMo organizational *Goal* concept is related to the Prometheus *Goal* concept. In this case, to allow the specification of both organizational and personal goals, we
chose to divide it in two types: individualGoals, that are the agent’s personal goals, and organizationalGoals, that are goals defined by the organization.

Conflicting concepts are those that do not share the same meaning in Prometheus and JaCaMo, like the Prometheus Data concept. Data concept represents both agent’s beliefs and environmental resources, which is already considered by the JaCaMo Artifact concept. Thus, we replaced the Data concept by the Belief concept, that only represents the agents beliefs.

As the result of the analysis of the metamodels, we have the Prometheus AEOlus metamodel, presented in the Figure 2. In this Figure, white concepts came from Prometheus metamodel; gray concepts are concepts from JaCaMo metamodel and black concepts are those existent in Prometheus that changed their meaning in the Prometheus AEOlus metamodel.

Fig. 2: Prometheus AEOlus metamodel

In the Prometheus AEOlus method agents can perform two actions types: InternalActions, that only changes the agent’s internals, and Operations, that is performed on an environmental Artifact. Artifacts are grouped in Workspaces and, when an operation is executed, they generate some Perception to the Agents. These Perception can become a new Belief in the agents beliefs base or a TriggerEvent that starts a Plan.

Agents have Plans to achieve their Goals. Goals can be composed of Subgoals and they and they can be either IndividualGoals or OrganizationalGoals.
Organizational Goals are grouped in Missions, that structures the organization’s Social Scheme, and they are assigned to the Agent by the Role it plays in the organization. Missions are bind to the Roles by a norm, that specify obligation or permissions to achieve a specific Organizational Goal. These Roles can be structured in Groups and Subgroups.

5 Prometheus AEOlus Method

The Prometheus AEOlus method uses an interactive incremental process based on the Prometheus process. Prometheus AEOlus uses the same three development phases used in Prometheus (system specification, architectural design and detailed design) and a fourth phase called implementation, where some entities defined in the previous phase are refined to allow the code generation for the JaCaMo framework. Figure 3 presents all these phases and the work products produced in each one. In this Figure, work products presented in white are those existent in Prometheus method and used in Prometheus AEOlus too; in black, are those existent in Prometheus but changed in Prometheus AEOlus; and in grey are new work products defined only in Prometheus AEOlus method.

![Prometheus AEOlus process diagram](image)

**Fig. 3: Prometheus AEOlus overview**

Four graphical work products are introduced in the architectural design phase of the Prometheus AEOlus method (Structural, Missions, Normative and Environment Overview) and one textual work product was introduced in the implementation phase (Artifact Descriptor). These work products complements the environmental and organizational specification.

Due to the limited space, in this article we only present the work products used for the organizational and environmental specification. These work products
are presented using an example based on Multi-Agent Programming Contest 2013 Edition\textsuperscript{3}, called Agents on Mars. Each work product is presented with its respective notation.

In the Agents on Mars scenario, two teams are competing to find the best water wells and occupy the Mars best zones. The environment is represented by a graph where each vertex has a number representing its value. A zone is a subgraph with at least two nodes and each zone has a value, determined by the sum of the vertices values. The main goal is to maximize the score, computed by summing up the values of the zones occupied by the team and its current money. The money is increased when the team executes some activities, like probe vertices, survey edges and attack enemies.

5.1 Organizational Modelling

The organizational specification starts with the goal overview diagram. This diagram is used by both Prometheus and Prometheus AEOlhus methods in the system specification phase. In this phase we aim to build a detailed and clear definition of the system, answering the question “What the system should to do?”. Thus, in the goal overview we summarize all identified system’s goals and subgoals in an AND/OR tree. In this tree we can visualize dependencies among the goals. For example, in the goal overview in Figure 4, the main system goal is To Maximize Score and it is decomposed in three subgoals: Occupy good zones, Defend zones and Get Money. All these subgoals can be pursued at the same time since they are decomposed using an AND operator. The goal Occupy good zones is also decomposed using the AND operator with a precedence order, suggesting the sequence in which the subgoals must be pursued. That is, first the Figure out the map subgoal is achieved, then Define good zones is achieved, and finally Place the agents is achieved. Likewise, the subgoal Defend zones is decomposed using the AND operator with a precedence order to be achieved. The goal Get money is decomposed using an OR operator and any of its subgoals (Probe vertices, Attack enemies and Survey edges) can be selected.

Fig. 4: Goal Overview Diagram

\textsuperscript{3} http://multiagentcontest.org/
Later, in the architectural design phase, the goals specified in the goal overview diagram are arranged in coherent subsets called missions. These missions are demonstrated in the missions diagram, presented in Figure 5. Each mission is assigned to an agent by the role it plays in the system. Thus, a mission must be composed of a consistent set of goals, since the agent that assumes a mission must be able to achieve all the goals of the mission. For example, the mission called Occupy is composed of Define good zones and Place the agents goals. Thus, the agent that assume this mission must achieve both goals.

![Mission Diagram](image)

Fig. 5: Mission Diagram

Also in the architectural design phase, the structural diagram shows the system’s roles, how these roles are grouped and the links among them. A role is defined when a specific behaviour is necessary in the system. These behaviour can be defined based on the system’s initial description. For example, in the Agent on Mars scenario description, we can identify five roles: Sentinel, Inspector, Explorer, Repairer and Saboteur. A sixth role, called Leader, is defined as a project choice, since we chose to use a centralized approach for decision making. Figure 6 shows the structural diagram for this example. In this Figure, the abstract role TeamMember is defined. This abstract role is used to simplify the specification by means of inheritance. No agent can directly play an abstract role and it is a “super-role” that all roles inherits characteristics.

A group is composed of a set of related roles. Each role is included in a group with its cardinality (min and max) that represents the number of agents that can play this role. A group can also contain some subgroups, each one with its cardinality. One group is labelled as “well formed” if all its cardinalities are satisfied. In the Agent on Mars example, we defined a main group called Team. This group is composed of two subgroups - Conquest and Defense - and one role called Leader. To be “well formed”, exactly one instance of each subgroup should be created and one agent must play the role Leader. The subgroup Defense is composed of the roles Repairer and Saboteur, and the Conquest subgroup is composed of the roles Sentinel, Inspector, and Explorer. All these roles cardinalities is four, that is, four agents have to play each role. The Leader is a role played by one agent that also plays the role Explorer. In a group, the roles are linked to represent acquaintance, authority, communication or compatibility among them. In the Agents on Mars example, shown in Figure 6, we used the abstract role
**TeamMember** to define, by inheritance, compatibility and acquaintance among all other roles. Further, the role **Leader** has authority on the others and they can communicate with the **Leader**.

The last diagram used to specify the systems organization is the normative diagram. This diagram presents the norms that links roles and missions. Two kinds of norms are used: permission, that states that an agent is allowed to commit to the mission, and obligations, that states that an agent ought to commit to the mission. It is important to note that if an agent is obligated to a mission it is also permitted to this mission. Figure 7 shows the normative diagram for the *Agents on Mars* example. In this Figure, some missions should be committed by more than one role, like the **Survey** mission that is committed to all agents. Likewise, some roles roles are obliged/permissioned to more than one mission, like the role **Explorer**, that is obligated to commit to the missions **Probe** and **Knowing** and is permitted to commit to the mission **Survey**. That implies that the agent playing **Explorer** must achieve the goals of all these missions.
5.2 Environmental Modelling

The environment dimension is specified in the architectural design phase and the main work product used to specify these concepts is the environment overview diagram. It presents the environmental artifacts, that are grouped into workspaces. This diagram shows the possible actions to be performed in each artifact and perceptions generated by them. Artifacts can be defined based on the initial system description, when we identify the need of some shared resource or shared data among agents or the need of an interface between the system and another external entity. For example, in the environment overview for the Agents on Mars example, presented in Figure 8, we use two artifacts in the same workspace called Mars. The artifact Server creates an interface between the system and the game server, and the artifact Map is used to share all agent informations about the scenario map. Furthermore, we can create artifacts to coordinate agents actions or to help the communication among them. In the Figure 8, we also present some of the possible actions performed in each artifact (e.g. the probe action in the Server artifact and the send_data action in the Map artifact) and the perceptions provided by each one (e.g., the position perception from the Server artifact and the new_zone perception from the Map artifact).

![Environment Diagram](image)

Later, in the implementation phase, the artifact descriptor work product is used to refine these artifacts, including some relevant information for the code generation phase. The artifact descriptor is a structured textual work product in which we provide a brief description of the artifact, the operations provided by it, the parameters needed to instantiate it, and all observable properties and events provided by the artifact. The descriptors for the artifacts in the Agents on Mars example are presented in the Table 2.

6 Code Generation

The Prometheus AEOlus method provide some guidelines to allow the assisted translation from the developed work products to code, using as target platform the JaCaMo framework. For the organizational dimension, the code is implemented in Moise, by a XML file. The overall file structure, presented in the Listing 1.1, includes the code for the functional, the structural and the normative system dimensions. The organization is defined within
the `<organizational_specification>` tag. Within it, three tags are included: `<structural_specification>`, where the organizational structure is coded with their roles and groups; `<functional_specification>` tag, where the organizational goals and missions are coded; and `<normative_specification>`, where the organizational norms are placed. Each specification is detailed below.

Listing 1.1: Overall structure to Moise XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<organizational-specification
  id = "id_organizational_specification"
  os-version = "0.8"
  xmlns = 'http://moise.sourceforge.net/os'
  xmlns:xsi = 'http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation = 'http://moise.sourceforge.net/os
                       http://moise.sourceforge.net/xml/os.xsd' >
  <structural-specification>
    <!-- put structural specification here -->
  </structural-specification>
  <functional-specification>
    <!-- put functional specification here -->
  </functional-specification>
  <normative-specification>
    <!-- put normative specification here -->
  </normative-specification>
</organizational-specification>
```

The structural dimension is coded based on the structural diagram. This specification includes all system roles, groups and links. In the Listing 1.2, the code for the structural dimension for Agents on Mars example is presented. In this Listing, translated based on the structural diagram\(^4\) presented in Figure 6, roles and their hierarchy are implemented by the tag `<role-definition>`, in line 1. Within this tag, we use a `<role>` tag to implement each role that composes the organization, like presented in lines 2 to 7. To the child roles, a `<extends>` tag is used to specify the super-role, like presented in lines 3 to 7. Groups are implemented within the `<group-specification>` tag, with the roles that composes

---

\(^4\) Due to limited space, we do not present all the code for each diagram.
this group. Line 8 presents the code for the group Team, that is composed of one role, presented in line 9 within the tag <roles>, and two subgroups, coded in the line 18 within a tag <subgroups>. The links between the roles within a group are presented by the tag <links> in the line 10 and by the tag <formation-constraints> in line 29, in which the compatibility link is implemented.

Listing 1.2: Structural specification to Moise XML file

```
<role-definitions>
  <role id="TeamMember"/>
  <role id="Sentinel" extends role="TeamMember"/>
  <role id="Leader" extends role="Explorer"/>
  <role id="Saboteur" extends role="TeamMember"/>
</role-definitions>
<group-specification id="Team">
  <roles>
    <role id="Leader" min="1" max="1"/>
  </roles>
  <links>
    <link from="TeamMember" to="TeamMember" type="acquaintance" scope="intra-group"/>
    <link from="Leader" to="TeamMember" type="authority" scope="intra-group"/>
    <link from="TeamMember" to="Leader" type="communication" scope="intra-group"/>
  </links>
  <subgroups>
    <group-specification id="Conquest" min="1" max="1">
      <roles>
        <role id="Sentinel" min="4" max="4"/>
        <role id="Explorer" min="4" max="4"/>
      </roles>
    </group-specification>
    <group-specification id="Defense" min="1" max="1">
      <roles>
        <role id="Saboteur" min="4" max="4"/>
      </roles>
    </group-specification>
  </subgroups>
  <formation-constraints>
    <compatibility from="TeamMember" to="TeamMember" type="compatibility" scope="intra-group" bi-dir="false"/>
  </formation-constraints>
</group-specification>
```

The functional dimension is coded within the <scheme> tag. For the Agents on Mars example, we present the code to this dimension in the Listing 1.3. It was translated in two steps. The first one is the implementation to all systems goals and its hierarchy, based on goal overview diagram. For the example, we used the diagram presented in Figure 4. These goals are coded within the tag <goal>, as shown in the line 2. The tag <plan> is used to indicate if the subgoals are achieved in parallel (as in line 3), in sequence (as in line 5) or by a choice (as in line 11). The next step is to implement the missions, based on the missions diagram. For the example, the diagram was presented in Figure 5. A mission is coded within the <mission> tag, where each goal that composes this mission is listed, as shown in lines 17 to 27.

Listing 1.3: Functional specification to Moise XML file

```
<scheme id="scheme">
  <goal id="Maximiz e score">
```

174
<plan operator = "parallel">
  <goal id=" Occupy good zones"/>
  <plan operator = "sequence">
    <goal id=" Figure out the map"/>
    <goal id=" Define good zones"/>
    <goal id=" Place the agents"/>
  </plan>
</goal>
<goal id=" Get Money">
  <plan operator = "choice">
    <goal id=" Probe vertices"/>
    <goal id=" Attack enemies"/>
    <goal id=" Survey edges"/>
  </plan>
</goal>
<goal id=" Occupy">
  <mission id=" Occupy">
    <goal id=" Place the agents"/>
    <goal id=" Survey edges"/>
  </mission>
  <mission id=" Survey">
    <goal id=" Survey edges"/>
  </mission>
  <mission id=" Knowing">
    <goal id=" Figure out the map"/>
  </mission>
  <mission id=" Probe">
    <goal id=" Probe vertices"/>
    <goal id=" Attack">
      <goal id=" Attack enemies"/>
    </goal>
  </mission>
</goal>
</plan>

The normative dimension is coded based on the normative diagram. In the Listing 1.4, it is presented the code for the normative dimension of the example Agent on Mars, based on the diagram presented in the Figure 7. The <norm> tag is used to implement a norm. Each norm is composed of a type, a role and a mission. Two types of norms can be used: obligation, like presented in the line 2, or permission, presented in line 3.

Listing 1.4: Normative specification to Moise XML file

```
<normative-specification>
  <norm id="n1" type = "obligation" role = "Explorer" mission = "Probe"/>
  <norm id="n2" type = "permission" role = "Explorer" mission = "Survey"/>
  <norm id="n3" type = "obligation" role = "Saboteur" mission = "Knowing"/>
  <norm id="n5" type = "obligation" role = "Saboteur" mission = "Attack"/>
  <norm id="n11" type = "obligation" role = "Leader" mission = "Occupy"/>
</normative-specification>
```

The environmental code is implemented in Java programming language using the CArtAgO framework. A CArtAgO artifact is programmed directly by defining a Java class that extends the cartago.Artifact class. To create this class, we use the environment overview diagram and the artifact descriptor, where we have the main information needed. We present, in Listing 1.5, the class for the Server artifact used in the Agents on Mars example. This class was translated based on the environment overview diagram presented in Figure 8 and on the artifact descriptor presented in the Table 2. The class name is defined by the artifact name. A special method called init, presented in line 7, specifies how the artifact is created. In this method, we usually use the primitive defineObsProperty() to define the artifact observable properties, specifying the name and the initial value of each property, like presented in line 8. An operation is defined by
a method annotated by the @OPERATION tag and it has no return value, like presented in the line 9.

Listing 1.5: CArtAgO class to Server artifact

```java
import cartago.Artifact;
import cartago.OPERATION;
import cartago.ObsProperty;

// Server Artifact
public class Server extends Artifact {
    @OPERATION public void init() {
        defineObsProperty("position", 0);
    }
    @OPERATION public void probe() {
        System.out.print("probe() operation");
    }
}
```

For the agent dimension, the code is implemented in Jason and it translated from some Prometheus diagrams that remain in Prometheus AEOlus method. However, as this paper focuses mainly on the environment and organizational dimensions, these diagrams and the code generation for the agent dimension are not presented.

7 Conclusions

The Prometheus AEOlus method aims both the MAS analysis and implementation integrating the agent, organization, and environment dimensions. Each dimension is analyzed and implemented as first-class entity, using specific concepts and abstractions. To minimize the conceptual gap between the analysis and programming phases, we use the same concepts in both phases. This approach’s main advantage is a straightforward translation from the work products used during the analysis phase into code. Nevertheless, the Prometheus AEOlus method is platform-dependent, since it was developed based on the JaCaMo framework, and uses mainly concepts from this framework. Although the proposed method aims at a specific platform, the approach used to achieve this method (metamodels alignment and an existing method extension) could be followed using other platforms and methods.

We also conduct a preliminary evaluation with a group of 30 undergraduate and graduate students. This primary test aimed to evaluate the method and its modelling language, including aspects like understandability, acceptability, expressiveness, and efficiency. The students who had integrated this evaluation have no previous knowledge in the agent-oriented field and they used the method to design and analyze a simple example system. Then, they answered a survey with ten questions about these evaluation aspects. The results allowed us to improve the method, changing some diagrams and notations. However, due to the limited time and the limited users knowledge, this preliminary evaluation did not include all method aspects.

The next step in the Prometheus AEOlus development is to implement a tool that supports all Prometheus AEOlus phases and the automatic code generation.
Also, a formal verification to the final metamodel and a comparison evaluation to other methods is necessary to further improve the method.

Acknowledgements

The authors would like to thank the Brazilian agencies CAPES and CNPq (grant number 140261/2013-3), Prof. Lin Padghan and colleagues for the support of this research.

References

From Multi-Agent Programming to Object Oriented Design Patterns

Mehdi Dastani and Bas Testerink

Intelligent Systems Group
Utrecht University
The Netherlands
{m.m.dastani, b.j.g.testerink@uu.nl}@uu.nl

Abstract. Various agent-based programming languages and frameworks have been proposed to support the development of multi-agent systems. They have contributed to the identification and operationalisation of multi-agent system concepts, features and abstractions by proposing specific programming constructs. Unfortunately, these contributions have not yet been widely adopted by industry. In this paper, we follow the argument that multi-agent programming technology can find its way to industry by introducing design patterns for the existing agent oriented programming constructs. We provide some object-oriented design patterns based on the programming constructs that we have developed in agent-based programming languages.

1 Introduction

Multi-agent systems technology aims at supporting the development of intelligent distributed software systems by providing high-level (social/cognitive) concepts and abstractions to conceptualize, model, analyse, implement, and test intelligent distributed systems. The development of a multi-agent system boils down to the development of a set of individual agents, their organisation, and the environment with which they interact. Individual agents are required to be autonomous in the sense that they are able to make their own decisions to either achieve their objectives (proactive behaviour) or to respond to received events (reactive behaviour). The organisation is supposed to coordinate the agents’ behaviours in order to ensure the overall objectives of the multi-agent system. Finally, the environment encapsulates resources and services that can be used by the agents.

In the past decades, various programming languages and frameworks have been proposed to support the development of multi-agent systems. These programming languages have provided dedicated programming constructs (either in a declarative, imperative, or hybrid style) to support the development of specific features of multi-agent systems. While some programming languages extend standard programming technologies such as Java (e.g. Jade [2] and Jack [4]), other agent-based programming languages are specified from scratch (e.g. 2APL [6],
GOAL [11] and Jason [3]). These programming languages and frameworks focus on specific sets of concepts and abstractions for some of which operational semantics and execution platforms are provided.

Without doubt a merit of these programming languages is the plethora of programming constructs that support the implementation of various features of multi-agent systems. For example, BDI-based agent-oriented programming languages such as 2APL, Goal and Jason can be seen as technologies that demonstrate how autonomous agents can be developed by means of a set of conditional plans and a decision procedure that continuously senses the environment to update its state, reasons about its state to select conditional plans, and executes the selected plans. Other programming proposals focus on the implementation of specific features concerning organisations or environments of multi-agent systems by proposing programming constructs to implement norms and sanctions, mobility, services, resources or artefacts.

Although these programming languages and frameworks have contributed to the identification and operationalisation of multi-agents systems concepts, features and abstractions, they have not been widely adopted as standard technologies to develop large-scale industry applications. This may sound disappointing, in particular because technology transfer has been identified as a main challenge and a milestone for the multi-agent programming community. There are various reasons for why these programming languages and frameworks fell short of expectations [5]. First of all, the adoption of new technologies by the industry is generally assumed to be a slow process as the industry often tends to be conservative, employing known and proven technologies. Moreover, industry adopts new technologies when they can be integrated in their existing technologies, and more importantly, when they reduce their production costs, which is in this case the costs of the software development process. Finally, the industry tends to see the contribution of multi-agent programming community as AI technology. The main problems with such technologies are thought to be their theoretical purpose, scalability, and performance.

The aim of this paper is to stimulate the transfer of multi-agent programming technology to industry. We start by the following three observations. First, object-oriented programming languages and development frameworks have already found their ways to industry. Second, it is common practice to use design patterns for often reoccurring problems in object oriented programs. Third, multi-agent programming technology provides solutions to a variety of reoccurring problems in large-scale distributed applications by means of dedicated programming constructs. Based on these observations and as argued in [20], we believe that multi-agent programming technology may find its way to industry by introducing object oriented design patterns that describe multi-agent programming constructs.

The starting point for our approach is to identify high-level multi-agent concepts and abstractions for which programming constructs have been developed. The identified concepts and abstractions, together with their developed programming proposals, can then be used to introduce corresponding design pat-
terns in the standard object-oriented technology. We first explain the multi-agent concepts and abstractions that form the main concern of existing multi-agent programming languages and for which dedicated programming constructs have been proposed. Subsequently we present object-oriented design patterns that support the implementation of these concepts and abstractions in standard object-oriented technology. Finally, we explain that the idea of agent oriented design patterns is not new and provide an overview of the related work and compare them with our proposal.

2 Autonomous Behaviour and Normative Mechanisms

Multi-agent concepts and abstractions are defined with respect to individual agents, multi-agent organisations and multi-agent environments. For example, individual agents are conceived as having autonomous behaviour in the sense that they have the ability to decide on their own which actions or plans to select and perform. Autonomous behaviour can be either proactive (i.e., agents behave to achieve their objectives) or reactive (i.e., agents behave to respond to their received events). These characteristic behaviours of individual agents, which are introduced to meet reoccurring challenges in the design and development of software agents, can be presented as design patterns in object-oriented technology. This vision suggests having an agent decision module (responsible for the generation of behaviours) that can be fed with various plan libraries (consisting of conditional plans) to achieve objectives or to respond to events.

At the level of multi-agent organisation, the provided concepts and abstractions can be used to introduce design patterns to cope with coordination and regulation concerns involved in distributed software systems. Separating coordination among processes as a concern has been argued in for instance [10], where the case is made for special coordination frameworks such as Linda. Norm-based regulation mechanisms are used to coordinate the behaviour of agents by means of norms being monitored and sanctions being imposed when norms are violated. In such coordination mechanisms a norm is a description of good behaviours and a sanction is a system response to norm violations.

Norms can be state-based, specifying that certain states are obliged or prohibited. Norms can also be conditional and have a deadline. When the norm condition is satisfied, certain states are obliged or prohibited before the given deadline. Norm based regulation mechanisms can be introduced using existing technologies such as aspect-oriented programming. Aspects allow crosscutting concerns to be programmed separately from the system’s core business logic. Norm based regulation mechanisms can be presented as design patterns based on aspects. The key correspondence is to use pointcuts from aspect oriented programming to specify where a norm applies (norm condition), and pointcut advices to check if a norm is violated and how to react to this violation (deadlines and sanctions).

The use of aspects may raise concerns about the open nature of multi-agent systems because programming aspects and weaving them at compile time require
the availability of the source code of the target processes. However, the use of aspects for open multi-agent systems can be realized through organisational interfaces that support the interaction between agents and the environment, or between agents themselves (cf. controllers in [13] and OrgBoxes in [12]). In this way, one can integrate the aspects in the source code of the organisational interfaces which does not require the availability of the agents’ source code. We would like to note that the use of organisational interfaces makes it feasible to develop and maintain norms separately from the business logic of the interfaces.

To illustrate our proposed design patterns we will fall back on the commonly used scenario of an electronic market. In terms of a multi-agent system it has the typical structure of agents, an environment and an organisation. The market is visited by agents that sell and buy items. The environment, which is called the market place, provides services that can be used by agents to register the items they want to sell and obtain the current offers for items. If an agent wants an item, then it can place a bid on it. The item can, for example, go to the highest bidder after a period of time expires. The market also has norms. The organisational part of the system enforces that an agent has to submit its payment details before it places a bid.

The object oriented reflection of the market consists of a market place class to serve as the environment. Agents are trader class instances whose behaviours are made according to our proposed design patterns. They run in their own threads. The norms will be implemented with aspects. At compile time the norms will be weaved in the system code to ensure that they are enforced properly. In the description of the design patterns we shall use sample code from this scenario. After the patterns are described we show how the market system can be developed using these patterns.

In order to describe the design patterns we shall use the common format from [9]. The proposed design patterns are presented with terminology that is common to object oriented programming, rather than agent technology jargon.

3 Pattern: Autonomous Behaviour

The behaviour of agents is generally described by beliefs, goals, events and plans. The beliefs of an agent can be seen as a system view (the agent’s context information) that the agent uses in its deliberation. We will capture this as a separate class that contains all the necessary data which is needed for selecting plans. A plan is coined a strategy, which is in line with the strategy design pattern from [9]. A strategy is selected and executed if it is both relevant and applicable. The relevancy of a strategy depends on the trigger (e.g. a specific goal or event) to which it responds, and whether that trigger occurred. The applicability of a strategy is determined by checking with the context information whether it is possible to execute the strategy.

Name and classification The autonomous behaviour pattern is a concurrency pattern. This design pattern has no other known names.
**Intent** The design pattern’s intent is to provide a solution structure for problems where triggers are processed autonomously. This is required for programs where the processes causing those triggers are not responsible for the processing. Design-wise it separates the core logic of the trigger causing processes, from the trigger handling. Aside from this, the pattern also provides structure to make the trigger processing context sensitive. The basic idea is that the behaviour is triggered by the system either through a notification or a direct method call. Then it tries to apply an execution strategy to process the trigger.

**Motivation & Applicability** The natural scenarios for autonomous behaviours are those where many autonomous processes are already present. For instance in an electronic market negotiating processes send messages (a type of event) which need not be instantaneously handled. A trading process can autonomously decide at its own leisure when and how messages are processed. The structure of typical scenarios is that independent processes work alongside each other. Examples are multi-agent systems, service oriented architectures and actor based systems.

![UML structure of autonomous behaviour.](image)

**Structure** Figure 1 shows the UML representation of the structure of the design pattern.

**Participants**
- **Proxy.** Interface to the behaviour. Either clients call the triggering methods (as in an Active Object) or the proxy is subscribed to triggers in the system.
- **Trigger.** Trigger instantiations are tokens indicating which triggers occurred.
- **ContextInformation.** The behaviour’s interface to the rest of the system. Also contains all the necessary data that is needed for the application of a strategy.
– **Strategy.** Strategies are used to process triggers. But for different circumstances (determined by the context) there can be different strategies for the same triggers. Also, one strategy might be able to handle several triggers.

– **Scheduler.** The scheduler schedules the triggers for processing. It loops through the trigger queue and applies applicable (by context) strategies for relevant triggers.

**Collaborations** The proxy is the interface to the autonomous behaviour. Either client processes can call the trigger methods or the behaviour catches them through a subscribe/notify relation. If a triggering method is called in the proxy, then a trigger instantiation is created and sent to the scheduler. The scheduler schedules the trigger in a queue for processing. It also continuously tries to process triggers by using strategies. A strategy has to be relevant for a trigger, but also applicable given the context of the system. To get information from the rest of the system, the strategy uses the context information instantiation of the behaviour.

**Consequences** The pattern decouples trigger from handling, by separating the triggers from the strategies that process them. The behaviour can be expanded with other capabilities such as dynamically changing the strategies. This enables self-healing and self-optimisation. An important design choice is to make the behaviour proactive or reactive. In a proactive behaviour, triggers stay in the queue until processed. This is similar to the idea of a persistent goal. In a reactive behaviour the trigger is considered only once for processing.

**Implementation** If the triggers stream in faster than their processing, then memory issues can happen. Also, there exists a possibility that a trigger has no relevant and/or applicable strategy for it. The programmer has to decide what should happen in such cases.

**Sample reactive behaviour code** We will use the example of how messages (a special case of events) can be handled by a trading agent. The environment sends a message to an agent if its bid in an auction lost or won, or if it successfully sold an item. The UML for this autonomous message handling behaviour is shown in Figure 2 (the Item class is omitted).

An autonomous behaviour becomes reactive if it only considers a trigger once. If no strategy is currently relevant and applicable for a trigger, then it is dropped. For a reactive behaviour the `processTrigger` method could be implemented like this:

1. Line 2: The trigger is permanently removed from the queue.
2. Line 3: All strategies are tried.
3. Lines 4-7: If a strategy is applicable and relevant, then it is executed.
4. Lines 9-11: If the trigger is still not processed, then an error procedure can take place.
The context of the behaviour contains the trader’s list of items which it can sell (stock). The message class implements the trigger interface. The different strategies check whether they can process a certain message and contain the code for the actual processing. The message proxy creates messages for the behaviour to process and calls the scheduler to enqueue them.

As an example, we will also show the strategy for handling a message that an item is sold.

```java
public void processTrigger() {
    Trigger trigger = triggerqueue.remove();
    for (Strategy strategy : strategies) {
        if (!trigger.processed(context) && strategy.triggeredBy(trigger) && strategy.isApplicable(context)) {
            strategy.execute(context, trigger);
        }
    }
    if (!trigger.processed(context)) {
        // initiate processing error handling
    }
}
```

The context of the behaviour contains the trader’s list of items which it can sell (stock). The message class implements the trigger interface. The different strategies check whether they can process a certain message and contain the code for the actual processing. The message proxy creates messages for the behaviour to process and calls the scheduler to enqueue them.

As an example, we will also show the strategy for handling a message that an item is sold.

---

2 Lines 2-6: The strategy is triggered if the trigger is a message, and that message is about an item that has been sold. The latter is checked by a boolean called sold.

Lines 8-10: The applicability test is in this case for safety. It ensures that the stock reference is not a null pointer.

Lines 12-16: First the trigger is cast to the message type. Then the item that was sold is removed from the stock. Lastly the message’s processed flag is set to true to indicate that it has been successfully processed.
Sample proactive behaviour code  To illustrate proactive behaviour, we will display how the traders pursue their buy and sell goals. The UML for the trigger and strategy interfaces remains the same, as does the UML for the scheduler class. However, we do have different strategies and a different proxy, context, and trigger realization. The UML of the proactive behaviour is shown in Figure 3.

Fig. 3. Example UML structure of a proactive behaviour.

The context for this behaviour has a reference to the trader where it belongs to, and the marketplace in which the trader operates. The processTrigger method is now implemented in such a way that triggers are re-inserted in the queue while they are not processed. Example Java code for the proactive processTrigger method is given below.³

³ Lines 2-3: Per strategy every trigger is checked to see whether the strategy is triggered.
```java
public void processTrigger() {
    for (Strategy strategy : strategies) {
        for (int i = 0; i < triggerqueue.size(); i++) {
            Trigger trigger = triggerqueue.remove();
            if (!trigger.processed(context) &&
                    strategy.triggeredBy(trigger) &&
                    strategy.isApplicable(context))
                strategy.execute(context, trigger);
            if (!trigger.processed(context))
                enqueue(trigger);
        }
    }
}
```

The goals of a trader are the items which it wants to sell or buy. The processed method is used to see whether the goal is achieved. An example of how the goal’s processed method can be implemented is as follows:4

```java
public boolean processed(ContextInformation context) {
    TraderContextInformation c = (TraderContextInformation) context;
    return (c.stock.contains(item) && !wantsToSell) ||
            (!c.stock.contains(item) && wantsToSell);
}
```

The sell strategy is to place an offer in the market place if the trader has not done so already. We assume that if an offer expires (auction deadline has passed) and there are no bidders, then the getOffer method will return null. 5

---

4 Line 3: An item is successfully bought if it is in stock now and the trader does not want to buy it.

5 Lines 2-6: Instances of the goal class trigger this strategy if it is a goal to sell something.

Lines 8-10: The applicability test is in this case for safety. It ensures that the stock reference is not a null pointer.

Lines 12-17: If the item is not already offered by the trader in the market place, then it will do so now.
public class SellStrategy implements Strategy {
  public boolean isTriggeredBy(Trigger trigger) {
    if (trigger instanceof Goal) {
      return ((Goal) trigger).wantsToSell;
    }
    return false;
  }

  public boolean isApplicable(ContextInformation context) {
    return ((TraderContextInformation) context).stock != null;
  }

  public void execute(ContextInformation context, Trigger trigger) {
    Goal g = (Goal) trigger;
    TraderContextInformation c = (TraderContextInformation) context;
    if (c.marketplace.getOffer(c.me, g.item) == null) {
      c.marketplace.offerItem(c.me, g.item, g.price);
    }
  }
}

Known uses

The pattern is visible in Jade where behaviours are used to construct agents. Also actor based programming has similar structures, but with less sophisticated handling of the messages (usually the processing is a big switch/if-else statement). Many web applications use a structure called progressive enhancement. There the content that the viewer gets is dependent on the context of the viewer. In those cases different strategies are applied for the same requests, which depends on the context of the user.

Related patterns

The most related is the active object pattern [9]. It too has this structure where calls are made through a proxy and are processed independently. However, it does not contain strategies, nor is the proactive version described for this pattern. Another important related pattern is the strategy pattern. It contains the solution to problems where a different execution strategy is needed under different circumstances. One could see the autonomous behaviour pattern as an active object, combined with the strategy pattern. Finally, another related pattern is the reactor pattern [17]. In this pattern, applications can register event handlers in an initiation dispatcher. Clients can then send events to the initiation dispatcher which notifies the correct handlers when they can process the events without causing synchronization problems. This is related to our reactive behaviour, due to its similar overall architecture. However, the selected strategies in our proposed pattern do not solely depend on the type of events but also on the system state considered as the context of the strategy. The proactor pattern [17] is a variant of the reactor pattern. The proactor pattern may wrongly suggest a relation with our proposed proactive variant. The main idea of the proactor pattern is to support the handling of the completion of asynchronous events rather than the handling of the initiation of asynchronous event processing, as it is the case with reactor pattern. In contrast, our proactive variant introduces proactiveness by pursuing goals until their achievement.
4 Pattern: Normative Constraint

Norms are related to constraints. But the term constraint already has a set meaning in design patterns (from the constraint pattern). Hence we refer to the counterpart of norms as normative constraints; constraints that can be violated albeit with consequences.

Name and classification The normative constraint pattern is a behaviour pattern. This design pattern has no other known names.

Intent Aspect oriented programming allows to disentangle crosscutting concerns from business logic. Exogenous norm-based regulation mechanisms \[7\] similarly have the separation of concerns between agents’ autonomous behaviour and the norms to which that behaviour must comply. A natural correspondence exists between certain types of norms and aspects.

On the one hand we have a specification of norm violating behaviour, and on the other hand we have the compensation for this violation. The intent of this design pattern is to catch this norm functionality. It allows to exogenously specify the norm from its subjected processes/classes/objects. We achieve this by using aspect oriented programming. In an aspect the pointcuts identify when an obligation starts to hold, when the obligation is fulfilled, and when the deadline has arrived. The advices are used to detach an obligation and to execute the sanction in case of a norm violation.\[6\]

Motivation and Applicability Just like the autonomous behaviour pattern, the normative constraint pattern naturally applies in scenarios where there are many autonomous processes. If multiple processes use the same resource then it is easy to build in constraints in the resource itself (such as in a database). However, sometimes this is not so straightforward.

For instance in the electronic marketplace we want to have norms about which kinds of items may be traded. If we have a market platform where agents can offer and bid on items, then we want to forbid offers or bids on items which are forbidden. Another example of this pattern is for instance a normative constraint for an electronic market that obliges traders after instantiation to submit payment details before they enter an auction.

The kind of scenarios where normative constraints are applicable are those where the constraints are mostly on interaction between components, rather than on the usage of a single resource. Typically the norm can change independently of the rest of the system. Also important is that the constraint is violable, it is not a hard constraint which cannot be transgressed.

Structure In Figure 4 the UML for this pattern is depicted.

\[6\] Prohibition can be implemented in a similar manner using its relation with obligation.
**Participants**

- **NormativeConstraint.** The aspect that contains the norm’s functionality and is responsible for detaching the norm when applicable, checking for violations of detachments, and removing detachments if necessary.
- **Detachment.** A detachment of the normative constraint. It contains relevant data from when the detachment occurred, which can be used to check whether the constraint is violated or not.
- **ContextInformation.** The context is the interface for information and data gathering of the system.

**Collaborations** The normative constraint creates a detachment if the condition holds. It can be the case that there are multiple detachments, but with different data. If the obligation holds then the detachment is removed. If the deadline holds, then the sanction is executed, after which the detachment is also removed.

**Consequences** The main objective is to separate the norm from the subjects of it. This is inherently the case because of the usage of an aspect. The separation between condition, obligation and deadline provides a clear specification of the temporal aspect of a detachable norm. With this pattern a system designer has the possibility to independently design complex violable rule structures for different use cases. The trade off is that the flow of control is harder to grasp because of the use of aspects.

**Implementation** Care has to be taken that the norm is not detached extremely often, because each detachment requires memory. If the detachments can somehow be ordered, then a heap or other sorted datastructure is preferable to an iterable due to run time complexities. Memory issues can occur easily if the deadlines and obligations are met in a slower pace than that the norm is detached.
Sample code The following code is the example where a trader is obliged to submit payment details before it makes a bid. When it does not do so, then the trader gets blacklisted. This means that when an auction ends, its bids will not be considered. Submitting payment details will get a trader of the blacklist (assumed to be implemented in the market place class).\footnote{7}

```
public aspect PaymentDetailsNorm {
    private ArrayList<Trader> detachments = new ArrayList<Trader> ();
    MarketPlace market ;

    pointcut marketInstantiation () : call ( public MarketPlace .new ( . . ) );
    after () returning ( MarketPlace market ) : marketInstantiation () {
        this .market = market ;
    }

    pointcut condition () : call ( public Trader .new ( . . ) );
    pointcut obligation ( Trader t ) :
        call ( public MarketPlace.submitPaymentDetails ( . . ) ) &&
        args ( t , String );
    pointcut deadline ( Trader t ) :
        call ( public MarketPlace.makeBid ( . . ) ) &&
        args ( t , Item , int );

    after () returning ( Trader t ) : condition () {
        detachments .add ( t );
    }

    before ( Trader t ) : obligation ( t ){
        detachments .remove ( t );
    }

    before ( Trader t ) : deadline ( t ){
        if ( detachments .contains ( t ))
            market .blacklist ( t );
        detachments .remove ( t );
    }
}
```

Known uses There is quite a lot of work on norms with a condition, obligation and deadline. In OO programming you typically see some boolean flag in code that signals whether some condition was met before and that is being used to steer execution at a later point.

\footnote{7}{Line 2: If a trader is in the list, then for that trader the obligation still holds. Lines 3-9: The aspect must have the reference to the market place. This is stored after the market place is instantiated. Lines 11 and 21-23: The norm comes into effect for each trader after they are created. Lines 13-15 and 25-27: The obligation is to call the submitPaymentDetails method. If the trader does so, then it is removed from the list of traders that is obliged to commit their payment details. Lines 17-19 and 29-33: The traders have to commit the payment details before they place a bid. If they bid whilst being on the list of detachments (line 30), then they are blacklisted (line 31).}
Related patterns Patterns with contracts among objects are also used to ensure behaviour over time. A related work is for instance Contract4J [18]. In design by contract for programs, contracts consist of preconditions, postconditions and invariants. A client must fulfill the precondition so that a server can perform an operation which fulfills the postcondition. Invariant constraints must hold at all times. If a contract is violated, then the program halts (in contrast to normative constraints).

Another related concept, though no pattern, is the Object Constraint Language (OCL), which is a part of UML. OCL is a design tool that allows a designer to specify very specific constraints such as the range of an integer attribute of an object. However, these constraints are also meant as non-violable constraints.

5 Use case

In the provided sample code of the design patterns we already showed how several components of the electronic market can be implemented. We shall now discuss the complete use case and illustrate how one can go about designing this distributed application.

The electronic market is a natural scenario for the different design patterns. In this use case one can clearly distinguish various concerns such as traders, the environment (market) and the organisation that regulates the agents’ behaviour. The market design will therefore also have the same segmentation. The market place itself, which serves as an environment, is independently maintained from the traders, which are agents. Hence the market has its own package. The busi-
ness logic of the market place is completely focused on the services rendered by the market, cf. registering offers and bids, and providing the auctions.

There are rules on how the market is organised. These are also maintained in their own package and follow the norm design pattern (though we only have one in the example). This has the advantage that the business logic of the market is not convoluted with code that relates to the enforcement of the norms. It is mainly a matter of separation of concerns. The norms can now be developed and maintained independently.

The agents all behave autonomously. Therefore the autonomous behaviour design patterns apply well for them. In our example the only agents are traders, but ideally the software is designed such that it is straightforward to add other agents. For instance, one can imagine a sniffer agent that continuously tries to obtain an overview of the items on offer for a certain type such as comic books. We design the traders for our market as objects that contain two behaviours. A reactive behaviour for handling messages, and a proactive behaviour for pursuing buy and sell goals. These behaviours share the same system view, which can be seen as the trader’s beliefs. The strategies are maintained in a separate subpackage of the trader.

Because we want to be able to introduce new agents we separated the scheduler class, its refinements (reactive scheduler and proactive scheduler) and its required interfaces (the trigger, system view, and strategy interface) in a behaviour package. To introduce for instance the sniffer agent, the developer has to create its behaviour by specifying its proxy, the possible triggers for strategies, and the strategies that the sniffer uses. It can reuse one of the schedulers to obtain autonomous behaviour. The overview of the electronic market is depicted in Figure 5.

6 Related work

The idea of agent-based design patterns has grabbed the attention of many researchers in the field. There have been several proposals focusing on various categories of design patterns. Some of the earliest agent oriented design patterns are proposed by Aridor and Lange [1]. They proposed agent design patterns for mobile agent applications and classified them into traveling patterns, task patterns and interaction patterns. An example of traveling patterns is the itinerary pattern that defines routing schemes for multiple destinations and handles special cases such as non existent destination. The task patterns are concerned with decomposing tasks and their delegation. An example is the master-slave pattern that allows task delegation from master to slave. Finally, the interaction patterns are concerned with agents’ communication and cooperation. For example, the meeting pattern allows agents to dispatch themselves to a specific destination (a meeting place) and engage in local interaction. Our proposed design patterns are complementary as we are not concerned with mobile agent applications, but with the internal design of autonomous agents and how such agents can be controlled and coordinated by means of norms.
Sauvage presents different classes of patterns such as Meta patterns, Methaphoric patterns and Architectural patterns [16]. Examples of meta patterns are organisation and protocols which are defined in terms of roles, their relations, and messages. An example of metaphoric patterns is the marks pattern, which describes an indirect communication model via environment. Examples of architectural patterns are BDI architecture consisting of knowledge bases and horizontal architecture consisting of parallel modules (e.g., deliberation and act modules). Our proposed design patterns for autonomous behaviour and norm-based coordination are related to the BDI architecture pattern and organisation pattern. Although Sauvage provides only a two lines description of BDI architecture pattern, we provide an extensive description and possible refinements of it. Moreover, Sauvage conceives an organisation pattern as being defined in terms roles and their interactions while our organisation is defined in terms of norms being monitored and norm violations being sanctioned.

In order to organise interacting intentional software entities in multi-agent systems, social patterns are introduced in [8]. Two specific categories of patterns introduced here are pair patterns and mediation patterns. The pair patterns describe direct interaction between intentional agents while mediation patterns describe intermediate agents that aim at reaching agreement between other agents. An example of pair patterns is the booking pattern for booking resources from a service provider, and an example of mediation patterns is the monitor pattern that allows receiving notification of changes of state. Our proposed design patterns for autonomous behaviour are complementary to the patterns proposed in [8] and describe the internal design of individual agents.

In [19] a pattern language is presented to capture various patterns in the design of multi-agent systems. The language consists of five interrelated patterns that together capture the different aspects of agent systems. The virtual environment pattern captures the design of an environment in which agents are situated. Those agents are captured with the situated agent pattern. It is very common that agents have a limited view on the system, which is documented as the selective perception pattern. For the coordination of agents the language contains two patterns: protocol-based communication and roles & situated commitments. The patterns are described in a architectural design language whereas we focused on object-oriented programming. That is less general, but easier to adopt.

Probably the closest agent oriented design patterns to ours are those proposed in [14], which aim at supporting the development of BDI agent-based systems. They propose four agent design patterns called dynamic strategy selection pattern, intention decomposition pattern, mutually exclusive intentions pattern, and necessary intention pattern. For example, the dynamic strategy selection pattern describes how an agent’s intention can be achieved by the best strategy from a set of strategies at run time. Our design patterns for autonomous behaviour are similar to dynamic strategy selection pattern. But, in contrast to this pattern, we distinguish two different refinements for both reactive and proactive behaviours. In our view this distinction is crucial as they generate two
important types of behaviour, i.e., reactive behaviour generates only one single response to an event while proactive behaviour maintains a response until the goal is achieved. Moreover, our norm based design pattern are complementary to the patterns introduced in [14].

Finally, Oluyomi et al. [15] presents a two dimensional classification in order to analyse, classify, and describe some existing agent-oriented patterns. The vertical dimension is based on the stages of agent-oriented software engineering and distinguishes seven stages from requirement analysis to implementation and testing phases. The horizontal dimension is based on tasks and activities that are relevant at each stage of software development. For example, at the multi-agent system architectural level, the tasks are to design the system, the involved agents, and their interaction. The vertical and horizontal dimensions identify categories of agent oriented design patterns. For example, the category defined by the multi-agent system architectural level (vertical dimension) and system design activity (horizontal dimension) is identified as a structural patterns which describe the structure of agent organisations in terms of architectural components including knowledge component and environment. An example of an agent oriented pattern that belongs to this category is the embassy pattern. This pattern introduces an agent responsible for the interaction between a multi-agent system and other heterogeneous domains. Our design patterns for autonomous behaviour can be seen as a member of the category Agent Internal Architecture - Interaction patterns and our design pattern for norms as a member of the category Agent Oriented Analysis - Organizational patterns.

7 Conclusion

The adoption of multi-agent programming tools and technologies by the industry is a major challenge that still needs to be met by the multi-agent programming community. One possible way to meet this challenge is by transferring multi-agent programming technologies to the standard software technologies. An idea is to start with the high-level concepts and abstractions for which the multi-agent programming research field has provided computational models and programming constructs, and propose either corresponding language level supports in the standard programming languages (e.g., C++ or Java), or alternatively propose corresponding design patterns, i.e., general reusable solutions to problems such as proactivity, reactivity, adaptivity, monitoring and control. The language level support can either be realized by standard programming approaches such as meta-programming or aspect-oriented programming, where concepts such as deliberation and control can be considered as different concerns that can be programmed either by meta-programs or aspects. Although these suggestions are not mature and need to be worked out both in details and in practice, attempts along these lines can bring the multi-agent community closer to the industry.
References


196
CaFé: A Group Process to Rationalize Technologies in Hybrid AAMAS Systems

H. Van Dyke Parunak, Marcus Huber, Randolph Jones, Michael Quist, Jack Zaientz

Soar Technology, Inc.
3600 Green Court, Suite 600
Ann Arbor, MI 48105
{van.parunak, marc.huber, rjones, quist, jzaientz}@soartech.com

Abstract. Most agent research seeks insights about a single technology, and problems are chosen to allow this focus. In contrast, many real-world applications do not lend themselves to a single technology, but require multiple tools. In an applied AI company, each tool often has its own advocate, whose specialized knowledge may lead her to overestimate her tool’s contribution and diminish that of other tools. To form an effective team, the various members must have a shared understanding of how their tools complement one another. This paper describes CaFé (“Cases-Features”), a group process that we have prototyped for building a consensus mapping between tools and real-world problems. The five AI technologies encompassed in our prototype are cognitive architectures, intelligent user interfaces, classic multi-agent system paradigms, statistics and machine learning, and swarming. Structured group discussion identifies the dimensions of a feature space in which the technologies are distinct. The scheme that emerged from our exercise does not pretend to be an exhaustive characterization of the techniques, but it is a jointly owned map of our technology capabilities that has proven useful in design of new use cases.

1 INTRODUCTION

A recurring topic at AAMAS is how to move the results of research into real-world applications. Our company, Soar Technology (SoarTech), provides applied AI solutions to a range of customers. We find that real applications often do not align well with disciplinary boundaries that guide basic research.

Research progress requires focusing the researcher’s attention on a particular approach, tool, or technology, so that it can be characterized theoretically, implemented elegantly, and examined with a thorough experimental design. In this setting, it is appropriate to choose problems that are tailored to the features of the being studied.

Customers in the real world usually do not start with a particular method they wish to exercise. Their pressing problems do not respect the convenient categories according to which we structure research and train students. As a result, organizations that address real-world needs often assemble a toolbox of capabilities. In our case, we

1 For our purposes, we use the terms “approach,” “tool,” and “technology” interchangeably.
started with a single flagship technology (the Soar cognitive architecture [6]), but over the years have recruited a staff with capabilities quite different from our original focus. In the process, we have encountered a challenge.

Our researchers understand their own approaches very well, and tend to view every problem through a perspective that is appropriate to their own tools. Companies like SoarTech often dissolve into disjoint “centers of excellence,” each focused on a single technology, and each marketing to customer problems that align more or less with a center’s capabilities. Such a structure under-serves customers in two ways.

First, it may not fully address the needs of the problems to which it does respond. It is not uncommon for a multi-disciplinary company to end up competing with itself on some opportunities, when different technologists want to bring different tools to bear. In such cases, the different facets of the problem might be more thoroughly and robustly addressed if multiple tools could be applied in tandem.

Second, some large and gnarly problems are too complex for a single technical perspective, even for the most optimistic advocate of a single technology. Such problems are typically left to large “system integrators” who may not bring the depth of technical understanding offered by expert researchers. In overcoming the narrowness of academic researchers, system integrators often fall victim to technical shallowness.

As a company, we seek to avoid both the narrow stove-piping of the academy and the shallow technical depth of a large integrator. We want our technical experts to share an understanding of our set of technologies that will enable them to deploy the full strength of their capabilities in synergy with one another. This paper reports on the form and initial results of a group process that we have implemented for this purpose. We expect it to be of value to the AAMAS community in two ways.

First, as a contribution to the software engineering of agent-based systems, it offers a process to enable multi-disciplinary teams to address complex problems that require the hybridization of multiple agent technologies.

Second, though preliminary, the joint feature space that we derived in our initial deployment of the CaFé method may be of interest in its own right.

Section 2 outlines the CaFé process, which draws its name from two information artifacts contributed by each technical advocate: a Case study of a problem that is particularly appropriate for her technology, and a list of Features of problems for which her technology is appropriate. Section 3 summarizes the specific Cases and Features in our prototype exercise of the methodology. Section 4 reports on the case discussions that form the heart of the process. Section 5 describes the feature space that results from our process. Section 6 demonstrates the use of this feature space in a series of new design patterns. Section 7 offers a concluding discussion.

2 THE CaFé PROCESS

CaFé is a structured group process among advocates for different technologies that encourages them to compare their approaches in the context of several example applications, and helps them to generalize these comparisons as a set of features that make a problem (or part of a problem) appropriate for one or another tool (Fig. 1). Each
technology or tool is represented by an advocate who is expert in its use. Each advocate produces two artifacts representing her technology: a feature list describing the characteristics of a problem that would recommend the use of her technology, and a use case or example problem that she would consider an ideal candidate for deploying her technology.

The process of preparing these artifacts before the group begins interaction encourages each advocate to recognize that her technology is better suited to some problems than to others, and to articulate what those problems might look like.

The entire group of advocates then discusses each use case. The discussion includes proposals by each advocate of how each technology might contribute to the case, and fitting the different technologies into an overall pattern based on the case.

Finally, after discussing all of the individual use cases, the advocates review the features from the individual cases and seek an overall synthesis that discriminates among the individual approaches.

The features that result from this process are not as detailed as those initially proposed by the advocates. They do not characterize each technology by itself, but situate it with respect to the other technologies. Most important, they are jointly owned by the advocates as a group, and so can guide collaborative design on new projects.

3 THE ARTIFACTS

We considered five technologies, all familiar to the AAMAS community, in our initial foray with CaFé. Each has a strong advocate on SoarTech’s current technical staff, some of whose publications in each area are referenced below.

- Cognitive Architectures (CA) are reasoning frameworks, such as Soar [6,13] or ACT-R [1], that are derived from high-level cognitive models of human reasoning and problem solving, and are intended to produce realistic human-like results.
- Intelligent User Interfaces (IUI) are technologies intended to mediate between human users and machine reasoners (e.g., [12,14]).
• Multi-Agent Systems (MAS) is a collection of conventional MAS techniques that focus on inter-agent coordination, including BDI models, joint intention theory, and agent communication languages (e.g., [4,5]).
• Statistics and Machine Learning (SML) uses formal statistical methods to characterize data and detect patterns [8,10].
• Swarming harnesses self-organizing methods inspired by natural systems, with many simple agents interacting locally in a shared environment (“stigmergy”) [7].

For each of these approaches, we summarize the features and the case study proposed by its advocate. The purpose of these summaries is not to attempt a definitive statement of each approach, but to illustrate the flavor and level of detail involved in these artifacts. While these descriptions are abbreviations of the documents prepared by our advocates, each of those documents is still only one or two pages long.

3.1 Cognitive Architectures (CA)

Feature list: Cognitive architectures fit problems with these characteristics:
• Multiple simultaneous, interleaving tasks that frustrate the development of linear procedural code, but can be managed by pattern recognition
• Ability to handle and categorize special cases with pattern-driven processing
• Need to execute in real time (not much slower, but also not much faster), using least commitment to support rapid computation of an acceptable answer that can be refined if time is available
• Need for rapid reactivity to changed circumstances
• Need to support explanation of behavior to human stakeholders
• Real-time learning as the agent executes in the domain

They are a poor choice for problems that involve
• Rapid processing of large amounts of data (more than 10k items per second)
• Sequential batch processing
• Number crunching
• Execution much faster than real time (as in constructive forecasting)
• Offline learning

Case study: CA would be a good choice for a chef’s decision-support assistant. Recipes are declarative representations of “how to cook” something. But having a great cookbook doesn’t make someone a great chef. A great chef has extensive procedural knowledge and the ability to substitute, adapt, and handle interruptions and opportunities. Recipes are inherently serial, but cooking a meal requires opportunistic parallelism. A complete system would require situation interpretation and human-system interaction. The chef domain reflects the need for learning in a number of ways.
• Recipes are forms of declarative knowledge.
• Recipes can be taught/demonstrated.
• There is also “book knowledge” about ingredients, cooking techniques, etc.
• Recipes can be generalized and decomposed in goal-based fashion.
• Chefs acquire expertise by practicing cooking.
• Chefs learn about substitutions, short cuts, and handling unexpected events.
• Cooking knowledge can be “recomposed” to create new recipes and techniques.
• Chefs need to communicate with fellow chefs, servers, and suppliers.

3.2 Intelligent User Interfaces (IUI)

Feature list: Systems for which development of an IUI is appropriate tend to have one or more of the following features:

• Human-centric: Humans need to control, understand, and trust the system and its outputs.
• Incorporate human knowledge: The operator (or operators) have knowledge, including long term domain knowledge and short term situation awareness, that can improve system performance and/or outputs.
• Incorporate human decision-making: The operator(s) can make detections or decisions beyond the system’s capability or authorization.
• Adaptive / Mixed Initiative: The system needs to adjust its operating characteristics to take into account changing operator (or actor) beliefs, desires, and intentions, both between and within system execution cycles; alternatively, the system needs to prompt the operator (or actor) to adjust their behavior.
• Representation boundaries: The system needs to mediate between two or more frames (typically, a user representation such as a doctrinal air traffic control grammar and a software engineered representation such as an AI planner structure).
• Naturalistic (multi-modal) usage environment: The system needs to interpret multiple streams of user input (mouse, voice, text, pointing) and/or coordinate multiple streams of output (video, audio, haptic).
• Supporting human constraints: The system needs to act for the user in a domain that exceeds human scale (either long time intervals, large data sets, fast reaction time) or that exceeds the specific operator's ability to act effectively (e.g. expert support for novice users, problems of high complexity or very high cost of error).
• Personalization: The system should be tuned to the specific preferences of a particular user or user group (or actor/actor group).

Case study: It quickly became apparent that any realistic system we discussed would need to interact with human stakeholders, and in the end we did not consider a separate case for IUI, since we were comfortable that the cases proposed by other advocates would serve well to explore its complementarity with the other technologies.

3.3 Multi-Agent Systems (MAS)

AAMAS is accustomed to a broad use of the acronym “MAS” as including any system (including, for example, a swarming system) with many interacting agents. For our purposes, we focused on coarse-grained MAS techniques that rely on symbolic
representations. The advocate for this area is expert in agent communication languages, joint intention theory, and related high-level coordination techniques.

**Feature list:** Problems that are suggest the need for multi-agent systems exhibit some of the following features.

- **Teaming:** More than one agent is required to solve a problem.
- **Distributed:** Computational solution needs to be divided (e.g., complexity, location, incomplete information, role, function, computational space/power).
- **Synergistic:** Using multiple agents gives a better solution that using a single one.
- **Robustness:** Reduces/removes single point of failure.
- **Decentralized:** Advantageous for distinct agents to make independent local decisions, processing (e.g. parallelism), or actions.
- **Asynchronous:** Computation and interaction aren’t tightly coupled.
- **Organization:** Structure (interaction, control) between agents important and/or advantageous (e.g., societal, problem structure, communications requirement).
- **Heterogeneous:** Distinct agents with differing capabilities.
- **Dynamic teaming:** Components (agents) motivated but not required to coordinate.
- **Competitive:** Agents can work against each other.
- **Flexibility:** Independent contributors to portions of distributed solution.
- **Complexity/Scalability:** Multiple agents with localized modeling and reasoning can address larger problems.
- **Semantic:** Disparate localized representations and meanings.
- **Perspective:** Modeling and interpreting other components behavior/state.
- **Opacity/Compartmentalized:** Certain aspects of solution need to be hidden.

**Case study:** An MAS approach would be ideal for a mixed team of soldiers and heterogeneous robots. The robots could include ground, air, surface, and subsurface vehicles, each with potentially different types of sensors, effectors, communication modes, and levels of local computation. Special attention needs to be paid to the changing roles of each entity in the team. Communications are dynamic, because of adversarial jamming, complex terrain that limits propagation, and the need for tight coordination. Relations among the units change constantly as the mission unfolds.

### 3.4 Statistics and Machine Learning (SML)

**Feature list:** Problems that are suitable for SML exhibit some of these features:

- The availability of large amounts of sensor data (video/audio capture, etc.) to yield useful levels of significance;
- Difficult to reduce data down to a manageable amount of symbolic information, whether because
  - the correct feature set is not known and must be discovered,
  - the data is intrinsically complex (e.g., speech data), or
  - different symbolic reductions are appropriate in different contexts;
- The availability of clear metrics for correctness of data handling to guide learning;
- Training and testing data available or easy to generate at will;
Black-box with correct output is sufficient; no requirement to explain the interpretation of the raw data;

Need to handle uncertain inputs, or to produce multiple results with varying levels of numerical confidence

**Case study:** Consider the problem of commanding one or many autonomous (or partially autonomous) assets using multiple modalities in a naturalistic way. Such a system would need to integrate speech recognition, gesture recognition (whether visual or by smartphone or smartwatch with gyro and accelerometer), and sketching, as well as traditional computer or mobile device UIs. For user acceptance, the system would need to match existing protocols. For example, in a military context, gestures should be those already used to command infantry, and structured speech forms such as the SALUTE report [3] or the nine-line brief [2] should be followed, so that a mix of human and robotic assets can be commanded simultaneously.

### 3.5 Swarming (SW)

**Feature list:** The advocate for swarming characterized appropriate problems as

- consisting of discrete parts, such as robotic platforms, people, units of information, or events; if the natural decomposition of a problem is functional or assertional, rather than in terms of a set of entities, another technology may be preferred;

- consisting of diverse entities, performing diverse functions, and dealing with diverse information sources (since individual agents can preserve distinctions that would be lost in the mean-field approach of many equation-based formalisms);

- favoring distribution of computation across multiple platforms, whether because of communication limits that hinder centralizing data, or because of the need to parallelize computation in combinatorially large problems.

- allowing decentralized decision making by individual members of the swarm, within bounds established by the operator;

- subject to deprivation of computational resources, since swarming coordination through shared scalar fields is less demanding than symbolic manipulations;

- subject to rapid dynamic change that requires constant self-reorganization.

**Case study:** SoarTech has several projects in autonomous systems, such as ground robots and UAVs, and our sponsors are interested in assessing the trustworthiness of their autonomy software. Conventional assessments of the trustworthiness of an engineered system are based on statistical analysis of a fault tree describing the structure of the system [11]. Once we endow a system with autonomy, we must also consider different trajectories through mission space and the demands they put on various platform subsystems. We have developed a representation of an extended fault tree that combines a conventional fault tree of the platform with a hierarchical task network representing mission space, but the resulting structure is too complex to explore exhaustively. We propose using swarming agents to compute a probability distribution over alternative mission instantiations, and thus compute the probability of mission failure, analogous to the Top Undesirable Event in a conventional fault tree analysis.
These feature lists and case study nominations were prepared by the advocates independently of one another. Not surprisingly, they are difficult to compare directly. Some of the features do not distinguish between technologies (for example, the ability to respond to dynamic changes in the world). Others have no counterparts across approaches that would allow direct comparison.

This incommensurability of features is not surprising. In fact, it reflects the challenge of designing a hybrid AAMAS system, starting just with a set of technologies. The trade-offs among them emerge only when we consider them in the context of specific problems, motivating the series of case study discussions that we conducted.

CASE DISCUSSIONS

After advocates have circulated feature lists, we discuss each proposed case study. As suggested in Section 3, each discussion has two phases (though in our experience the thread of conversation often switches multiple times between the phases). In the proposal phase, advocates suggest how their technologies could be applied to the case, or to extensions of it that might realistically be required. In the fitting phase, the group seeks to fit the various technologies into the specific use case, exploring how to rationalize the role of each technology. This rationalization frequently draws from the feature lists originally prepared by the advocates, but instead of being unilaterally proposed by the advocates, it is the result of a group consensus. Each of these phases yields important insights about the relations among the technologies.

Each case was suggested by an advocate as ideally suited to one specific technology, but the proposal phase of each discussion never lacked for contributions from other advocates. As different advocates envisioned how their tools could be applied to a case, the problem tended to expand in scope. Sometimes different tools addressed the same facet of the problem from a different perspective, but more often the viewpoint prompted by a given tool encouraged us to consider a richer, more complex version of the use case, one that looked less like a toy laboratory problem and more like a real-world system. This experience reflects the insight about real-world problems that motivated CaFé in the first place. We hypothesized that such problems would benefit by synergy among multiple approaches, and in fact the more approaches we considered alongside a problem, the more realistic the problem itself became.

In the fitting phase of the discussions, we tried to rationalize the complementary contributions of each technology to the (sometimes expanded) case. This rationalization usually took the form of identifying some feature that distinguished alternative technologies in the context of the case under discussion. Sometimes these features were already articulated in the feature lists submitted by the advocates in advance, but often they became clear only through discussion of a concrete case.

A central insight resulting from our work on CaFé is the difficulty of comparing technologies directly with one another, and the relative ease of comparing them in the context of a specific problem. The individual features lists often claim the same problem characteristics for different technologies, but discussion of a concrete example
serves as a catalyst to highlight the differences that matter among the various approaches. Of course, different cases may yield different points of comparison among technologies, but in practice, after we had gone through three cases, we began to see recurring problem features that repeatedly distinguished between tools. We summarized these features in the final feature synthesis discussion (right-hand side of Fig. 1) to define the joint feature space discussed in the next section.

5 THE JOINT FEATURE SPACE

Two dimensions distinguish four of our technologies: CA, MAS, SML, and SW. These dimensions are a) high and low data integration, and b) high and low decomposability. We were unable to localize IUI in this space in a way that would distinguish it from the other four. Recall that one motive for CaFé is to understand what portions of a complex problem we should address with which technology. To achieve this objective, we seek a joint feature space that distinguishes all of our technologies. To meet this criterion for IUI, we propose a third dimension, c) high vs. low human involvement. Fig. 2 shows the resulting overall feature space. This joint feature space is not a definitive characterization of any of our methods, but instead focuses on features that distinguish them from each other.

5.1 Data Integration

The Data Integration dimension reflects the degree of linkage among the data items that the problem presents. High data linkage corresponds to a knowledge-rich domain, in which information includes a representation of the semantic relations among data items. In a domain with low data linkage, the relationships among data items are yet to be discovered. Often, problems with low data linkages present a larger amount of data (“data rich” problems), while the knowledge captured in spaces at the high end of the dimension allows the system to work with smaller amounts of data. From a systems perspective, the low integration, data-rich end of the dimension is associated with sensors that access the world directly, while the high integration, knowledge-rich end deals with analysis of data that has been subject to a fair amount of preprocessing. Some aspects of this dimension correspond to the JDL Data Fusion hierarchy [9], in which Level 0 deals with raw signal data, Level 1 identifies objects, Level 2 detects situations among multiple objects, and Level 3 identifies threats.
MAS and CA rely on symbolic knowledge representations, and so are most naturally applied to knowledge-rich problems. SW and SML can use data without such a knowledge overlay and suggest relations among data items that might later be represented explicitly. They can use a knowledge structure as a template against which to compare raw data (for example, using SML with a symbolic grammar), but they do not require this knowledge to be embedded in the data at the outset.

On review, several of the features suggested by the advocates for individual approaches align with this dimension.

- CA identified the need to explain its reasoning to humans, which requires high semantic content in its representations.
- SML recognized that it is most appropriate when the problem needs a “black box” solver that cannot explain itself.
- SW’s use of scalar fields to support deprived applications reflects its focus on data with low semantic integration.

However, by themselves these independent features are not nearly as useful in de-conflicting the technologies as is the data integration dimension that emerged as we discussed the application of these tools to common problems.

5.2 Decomposability

The decomposability dimension reflects the degree to which the problem invites solution by multiple interacting components. At the high end, it is natural to distribute the solution process across multiple platforms. At the low end, the most natural processing approach presumes that all information is available on a single platform.

Where the data integration dimension grouped MAS and CA against SML and SW, the decomposability dimension groups MAS and SW against CA and SML. Both MAS and SW use multiple computational entities, but differ in how they coordinate these entities: the stigmergic coordination common with SW agents is subsymbolic, relying on the amplitude of numerical fields over the environment, while MAS agents exchange symbolic information. But in both cases, the information available to individual agents is limited, and differs from agent to agent. CA and SML assume low decomposability. Most examples of CA assume a monolithic reasoner (like the human whose cognition these architectures are intended to imitate). While some clever methods for distributing SML computations have been explored, the fundamental model on which SML rests is the development of a single joint distribution over the variables of interest, which can then be marginalized as required, a computation that is most readily done with all the data in one place.

Again, this dimension reflects some features identified initially by tool advocates:

- SW is applicable to distributed, decentralized problems.
- MAS similarly recognized Teaming, Decentralized, and Distributed as problem characteristics that favor its application.
The case discussion, unlike the individual feature lists, showed the need for low decomposability for most effective application of SML and CA.

These two dimensions effectively distinguish four of our approaches (Table 1). However, IUI did not fit neatly into this taxonomy, leading to a third dimension.

5.3 Human Involvement

By definition, IUI technologies facilitate interaction of a human user with an automated system. One can envision a system drawing on our other approaches that does not interact with a human, but when the system as a whole requires human involvement, a user interface is required, and increasingly these interfaces use some degree of AI to facilitate the interaction. So we distinguish IUI from the other four technologies along a “Human Involvement” dimension on which the others are low and IUI is high.

Though IUI is applicable across the entire space spanned by the two dimensions of Table 1, it takes different forms in different areas of this space, depending on the other processes with which it interacts, as shown in Table 2.

<table>
<thead>
<tr>
<th>Data Integration</th>
<th>Low (Data-Rich)</th>
<th>High (Knowledge-Rich)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swarming</td>
<td>Low (single agents)</td>
<td>High (multiple agents)</td>
</tr>
<tr>
<td>Multi-Agent Systems</td>
<td>Statistics &amp; Machine Learning</td>
<td>Cognitive Architectures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Integration</th>
<th>Low (Data-Rich)</th>
<th>High (Knowledge-Rich)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Visualizer</td>
<td>Low (single agents)</td>
<td>High (multiple agents)</td>
</tr>
<tr>
<td>Peer Decision-Maker</td>
<td>Cognitive State Inspector</td>
<td></td>
</tr>
</tbody>
</table>

- In data-rich domains, IUI predominantly supports data retrieval and visualization. They allow humans to guide automated reasoners (whether SW or SML) (for example, by identifying information requirements, or presenting knowledge templates to which data should be fit), and they present to the user the structures discovered by underlying SW or SML processing. They naturally support an interactive approach to data exploration.
- In knowledge-rich, highly decomposable domains, IUI naturally allow humans to function as peers alongside computational agents. IUI presents the user with information that is sent to her from other agents, and translates human input into messages that are exchanged with other agents.
- In knowledge-rich domains with low decomposability, IUI enables the user to interact with a single CA agent (e.g., to inspect or modify the agent’s state).

The Human Interaction dimension directly reflects the multiple references to people in the original IUI feature list, including “Human centric,” “Incorporate human knowledge,” and “Incorporate human decision-making.”
6  SOME NEW DESIGN SCHEMATA

One of our motives in developing CaFé was facilitating the design of systems to address large, complex problems that require synergy among multiple AI approaches. In this section, we sketch a series of design patterns that illustrate the value of the feature space that we have developed. We could simply present hybrid designs for the case studies that we discussed, but to demonstrate the extensibility of our results, we instead describe a series of concepts distinct from the original case studies, but drawing on the same joint feature space.

6.1  Data Fusion and Shared Situational Assessment

A common problem in many domains, both military and industrial, is gathering data from many sensors monitoring the physical world, discovering patterns to develop a knowledge-rich characterization of the current situation, and then assuring that all decision-makers share a common view of that situation. Fig. 3 shows how our technologies might interact in such a system.

1. Both SW and SML deal with the raw data and detect regularities and patterns, which they expose to a human through a data visualizer IUI. The human in turn can guide the SW and SML agents to refine her view of the world, and refine and enhance the structures that are discovered.
2. Enriched with explicit knowledge through the actions of the human operating the data visualizer, the data can now be consumed by a CA agent that reasons over it in the light of other knowledge (including previous states of the world, mission plans and objectives, and hypotheses). The CA agent can also identify linkages that the human should further explore through the data visualizer IUI.
3. A cognitive state inspector IUI allows the human to monitor the reasoning of the CA agent and perhaps adjust it.
4. The CA agent shares its conclusions with other agents via MAS interfaces, achieving shared situational assessment across the team.
5. Some of these agents may be humans, who participate in the team via a peer decision-maker IUI.

We intentionally leave the links between components in this and the following schemata undirected. In general, we believe that information will flow in both directions; a more refined design would distinguish the nature of the flows in each direction.

Fig. 3. Schema for Data Fusion and Shared SA
6.2 Complex Pattern Detection in Data

Modern data analytics faces a tension between data that are too atomic to be diagnostic and knowledge that is too complex to guide search. For example, a single negative Tweet about US policy might be an isolated comment, part of an emerging viral propaganda campaign, or motivation for an invitation to a public demonstration. These alternatives require different responses, and detecting them depends on patterns involving multiple Tweets. Yet traditional methods of matching an overall pattern against high-volume, high-velocity data do not scale with the complexity of the pattern, particularly if the pattern encompasses several alternative possibilities, only one of which may match. Such patterns are too complex for efficient single-item queries, but the processing to match complete patterns is combinatorially infeasible.

We are developing an approach to such problems that fits the schema in Fig. 4.

1. A major challenge in knowledge-based systems is authoring the knowledge that drives the system. Currently, complex queries are assembled manually, but our schema anticipates the role of a CA agent in helping a human develop these patterns, perhaps on the basis of learning from past experience (not shown in the figure). A cognitive state inspector IUI facilitates this interaction.
2. This link indicates interaction between two different human roles: the pattern author (via a cognitive state inspector IUI) and the person using the pattern to interact with the data (via a data visualizer IUI). These may be the same person, or different specialists.
3. To avoid the combinatorial complexity of matching the entire pattern at once to massive data, we use swarming to evaluate the probability that different portions of the pattern are supported by the data, then estimate the value of alternative atomic queries in sharpening these distributions, and execute those queries, all under the supervision of a human via a data visualizer IUI.

6.3 Multi-Unit Combat Simulator

A major application area for MAS is in constructive combat simulations. Fig. 5 shows a schema that supports the development of a simulator for a multi-component force.

1. The simulator’s core is a set of CA agents, interacting through MAS interfaces.
2. The MAS organization allows humans to participate in the simulation via a peer decision-maker IUI, realizing the increasingly popular LVC (Live-Virtual-Constructive) mode of simulation.
3. One important feature of cognitive reasoning is anticipating future events. CA agents include some mechanisms for anticipation, but in anticipating geospatial motions, swarming has proven to be a powerful tool.

4. Human players can also benefit from the anticipatory view provided by swarming, via a data visualizer IUI.

5. The data visualizer and peer decision-maker IUIs in this case may be integrated to support a single human player.

6.4 Model Fitting

A recent project gathered opinions from humans via a (non-intelligent) interface to fit weights to knowledge models that let us estimate the similarity behind the human judgments informing the elicited opinions. Fig. 6 shows an expanded version of this system.

1. A CA agent, directed by a human via a cognitive state inspector IUI, develops the knowledge model that is to be fitted to the elicited opinions.
2. Swarming over the model develops the weights on individual edges in the model.
3. The differences between the spectra of weights from different informants are evaluated statistically.
4. The resulting measures of informant similarity then enable a CA agent (which may or may not be the same one involved in the original model authoring) to make more intelligent use of the opinions elicited from the different informants.

7 DISCUSSION AND CONCLUSION

The method described in this paper enabled experts in different AI specialties to develop a shared feature space showing how their tools complement each other. In turn, this feature space was effective in initial design of new systems beyond the case studies that drove the CaFé process itself.

Our exercise was a prototype of CaFé. Consider its extensibility and alignment.
By *extensibility*, we mean the behavior of the feature space as new technologies are added to the collection, and as we consider new problems.

We begin with extensibility to new technologies. The five we considered in this exercise do not by any means exhaust the repertoire that we have currently in house, not to mention others that we may acquire. One can imagine game theory in its many variations, distributed constraint optimization, and logic programming, to name only a few. Will adding others require redoing the whole process, yielding a feature space that is radically different from what we discovered for our initial five approaches?

Our experience with IUI is evidence that we can expand the feature space incrementally rather than having to redo it each time we add new technologies with new advocates. IUI did not fit cleanly into the two-dimensional space that the other four approaches suggested. However, adding the Human Involvement dimension allows us to disambiguate it from the other approaches, and careful attention to the nature of the original two-dimensional space allows us to tease apart different techniques within IUI that do exploit the insights of the two-dimensional space.

A related aspect of extensibility concerns the robustness of the joint feature space as we consider new problems. We developed the design schemata in Section 6 to test whether the feature space could be applied to problems other than those that stimulated its definition in our case discussions, and the results encourage us that the space is in fact robust across a wide range of problems.

By *alignment*, we call attention to the fairly minimal overlap between the original feature lists submitted by the advocates, and the dimensions of the resulting feature space. Because the Human Interaction dimension was introduced to distinguish IUI from the other approaches, it is not surprising that this dimension corresponds very closely to the features enumerated by the IUI advocate. However, other individual feature lists include a great deal of information and insight about individual approaches that is not captured explicitly in the dimensions of the joint space.

Some details of the original feature lists do align with the dimensions of the joint space. In addition, this observation about alignment reminds us again of the distinctive purpose of the joint space. Unlike the original feature lists, it is not intended to define each technology, but rather to show how they complement each other. Unused features in the original lists are a reservoir on which we may draw as we consider new technologies and new problems, to refine our understanding, not of technologies in isolation, but of the joint technical space that we are positioned to exploit.

Perhaps the most powerful insight from the CaFé experience is the ability of concrete problems to facilitate comparison of different technologies. The usefulness of a third object for clarifying mappings between two other objects suggests that a category theoretic model might be a useful way to formalize the CaFé process and lead to automated tools to support it, a direction that we hope to pursue in future work.

REFERENCES


Twenty Years of Engineering MAS
Methods, Tools, and Technologies

Koen V. Hindriks
Delft University of Technology, EEMCS, The Netherlands

The past twenty years we have seen an enormous development of new techniques and technologies for developing multi-agent systems (MAS) as well as an enormous growth in the number of methods and tools that support the engineering of MAS. Whereas the 1990ties perhaps is best characterized as the period that laid the foundations and started out with exploring the more theoretical underpinnings of the MAS field, besides a continuation of this foundational work, since 2000 the field has also demonstrated to have great potential for applying MAS technology in a very diverse range of application domains.

In a recent survey of applications of MAS technology [2], mature applications are reported in such diverse areas as Logistics and Manufacturing, Telecommunication, Aerospace, E-commerce, and Defense. The authors conclude that “dedicated agent platforms actually can make a difference regarding business success”. They also write that “more recent platforms [...] may take some more time to mature”. It was found, for example, that quite a few mature applications were built using one of the older and well-known agent platforms JADE [1]. In order to continue these successes, a key challenge is to identify what is needed to advance our technologies for engineering MAS to a level that they can be used to develop mature applications.

In this paper, we will focus in particular on cognitive agents, as it seems fair to say that most work reported in the international workshops ProMAS, AOSE, and DALT that recently merged into the EMAS workshop has taken its inspiration of what have been usually called Belief-Desire-Intention (BDI) agents. Arguably, the step to mature applications for technologies that support the engineering of cognitive agents and MAS is bigger than that of more general purpose frameworks for developing agents such as JADE. Interestingly, however, where initially most agent platforms were building on top of JADE, more recently we also have seen a move away from platforms that only recently still were running on top of JADE. One reason, moreover, why a broader uptake and the application of cognitive agent technologies has been somewhat slow perhaps may be that this work originally has had a strong conceptual focus, aiming, for example, to relate agent frameworks to formal theories of rational agents. We suggest that it is time to start paying more attention to what kind of support a developer needs to facilitate him or her when engineering future MAS applications.

We believe that to move forward it is important to learn from past successes and failures and to take stock of where we are today. To this end, we will explore and provide an overview of some of the more successful and well-known agent platforms and engineering methods. It may, however, be just as important to identify how we can make sure that a developer is provided with the right tools
for engineering MAS. We argue that to do so it is important to put more emphasis on practical aspects that not only relate to the design of languages, tools, and methods but also focus more on ease of use, scalability and performance, and testing.

References

Efficient Verification of MASs with Projections

Davide Ancona, Daniela Briola, Amal El Fallah Seghrouchni, Viviana Mascardi, and Patrick Taillibert

1DIBRIS, University of Genova, Italy
{Davide.Ancona,Daniela.Briola,Viviana.Mascardi}@unige.it
2LIP6, University Pierre and Marie Curie, Paris, France
{Amal.Elfallah,Patrick.Taillibert}@lip6.fr

Abstract. Constrained global types are a powerful means to represent agent interaction protocols. In our recent research we demonstrated that they can be used to represent complex protocols in a very compact way, and we exploited them to dynamically verify correct implementation of a protocol in a real MAS framework, Jason. The main drawback of our previous approach is the full centralization of the monitoring activity which is delegated to a unique monitor agent. This approach works well for MASs with few agents, but could become unsuitable in communication-intensive and highly-distributed MASs where hundreds of agents should be monitored.

In this paper we define an algorithm for projecting a constrained global type onto a set of agents $Ags$, by restricting it to the interactions involving agents in $Ags$, so that the outcome of the algorithm is another constrained global type that can be safely used for verifying the compliance of the sub-system $Ags$ to the protocol specified by the original constrained global type. The projection mechanism is the first step towards distributing the monitoring activity, making it safer and more efficient: the compliance of a MAS to a protocol could be dynamically verified by suitably partitioning the agents of the MAS into small sets of agents, and by assigning to each partition $Ags$ a local monitor agent which checks all interactions involving $Ags$ against the projected constrained global type. We leave for further investigation the problem of finding suitable partitions of agents in a MAS, to guarantee that verification through projected types and distributed agents is equivalent to verification performed by a single centralized monitor with a unique global type.

Keywords: Constrained Global Type, Projection, Dynamic Verification, Agent Interaction Protocol

1 Introduction and Motivation

Distributed monitoring of agent interaction protocols is interesting for various reasons. First, the distribution of monitoring reduces the bottleneck issue due to the potentially high number of communications between the central monitor and the agents of the system. Consequently, the communications are localized according to the distribution topology (how many local monitors are available
and where they are localized in the system), improving the efficiency of the monitoring. As usual, distribution increases the robustness of the whole system and prevents for a breakdown, crash or failure of the system. In particular, in the context of distributed environments, having a robust monitoring system requires to distribute the monitoring on several agents which ensure their prompt reaction to events.

In addition, the distributed approach is more suitable than the centralized one for asynchronous and/or distributed contexts.

Hence, we can mention at least three classes of applications where the distribution of monitoring is relevant.

1. MASs dealing with huge number of agents, for example applications in the context of supervising networks (e.g. [14]). The distribution becomes mandatory to deal with the complexity of the system and to guarantee its scalability.

2. Distributed MASs dealing with distributed agents because of the intrinsic geographical distribution of the system. This often happens in the context of industrial projects.

3. Pervasive MASs: in ambient intelligent systems for instance, agents are mobile (they move from one locality to another one) and their communication depends on their location. In such open environments, agents enter and leave the system and this requires a distributed monitoring of communication (e.g. local registration, etc.).

Usually, in systems related to the above three classes of applications, an overlay of agents is deployed above the real system. Agents are distributed over the system according to the topology distribution which has to satisfy several criteria (logical, physical or temporal, etc.) of communication in order to meet the target application requirements. The induced topology leads the agents to communicate with their local monitor or with their neighboring agents in order to exchange information.

In order to distribute the monitoring activity, the first step to face is to distribute the specification of the global interaction protocol in such a way that a subset of agents can monitor a subset of the interactions, still respecting the constraints stated by the global protocol.

In this paper, we address this first step by defining and implementing an algorithm for projecting the protocol representation onto subsets of agents, and then allowing interactions taking place within these subsets to be monitored by local monitors. Automatically identifying these subsets of agents in order to guarantee that the distributed monitoring behaves like the centralized one goes beyond the aims of this paper, but is matter of our current research activity; we have started studying sufficient conditions for distributing the monitoring of a protocol “at design time”.

Another interesting issue concerns dynamic redistribution of monitoring agents; even if not explored in this work, projected types could be recomputed dynamically to balance the load among local monitors depending on the currently available resources, and according to some “meta-protocol”.
The formalism that we exploit for representing and dynamically verifying agent interaction protocols, is constrained global types [2]. In our recent research we demonstrated that they can be used to represent complex protocols in a very compact way, and we exploited them to detect deviations from the protocol in a real MAS framework, Jason [1]. Extensions of the original formalism with attributes have been described [8] and exploited to model a complex, real protocol in the railway domain [9]. This paper shows how a constrained global type can be projected onto a set of agents $Ags$, obtaining another constrained global type which contains only interactions involving agents in $Ags$. Although the projection is always possible, this does not mean that it is always useful: we will show in the paper a protocol which can be projected onto any individual agent in the MAS, but that needs to be monitored in a centralized way to verify all its constraints.

The paper is organized in the following way: the sequel of this section describes one motivating scenario for our research; Section 2 briefly reviews the state of the art in runtime monitoring of distributed systems; Section 3 gives the technical background needed for presenting the projection algorithm in Section 4, Section 5 describes the implementation of the algorithm in SWI Prolog, Section 6 describes the algorithm at work, and Section 7 concludes.

**Motivating scenario.** In order to better understand the impact of distributed monitoring of complex and open systems, let us consider the following scenario: a humanitarian convoy in charge of food transportation is traversing a potentially hostile country. In order to ensure the convoy safety, a set of autonomous unmanned aerial vehicles (UAV) is deployed. The goals assigned to the UAVs are as diverse as: 1. maintaining the convoy within sight of a distant control center thanks to an embedded camera and data transmission; 2. transmitting images of the situation ahead of the convoy (to the convoy itself and to the control center); 3. ensuring data transmission from the convoy to the external world and conversely; 4. detecting potential hazards and informing the convoy and the control center; 5. localizing suspicious vehicles; 6. identifying a designated mobile entity, etc.

Several UAVs are required to achieve some of these goals since they require being at different locations at the same time (goals 1, 2). On the contrary, some goals can be assigned to the same UAV, providing the UAV traveling from one specific location to another one (goals 4, 5, 6). Moreover, some goals can be shared between UAVs (goal 3). When some UAV becomes unavailable, its goals must be allocated to another one or a new UAV must take-off depending on the resources availability. It is the case when communication failures occur, which might be temporary or permanent. It is also the case of instrument failures on-board UAVs, of meteorological events, etc. Due to situation-related hazards, the convoy might (autonomously or by a decision coming from the control center) decide to change its route. This change has to be taken into account by all the UAVs, which implies at the same time a re-planning of UAVs trajectories but also re-planning of the tasks they have been allocated to since their feasibility is not anymore ensured (fuel resources, communication network, etc.). It is of a
major importance that the protocols used inside the system is monitored for two reasons: 1. possible errors in protocols might generate confusion among agents and generate bad decisions whose consequences might be dramatic; 2. malevolent actors might try to penetrate the system since humanitarian operations almost often occur in a tense political context.

Unfortunately, a centralized monitoring is difficult to carry out in such a system since it forces every agent to communicate with a unique control agent, which is not always possible due to the physical dispersion of the agents. For example, a low altitude UAV can only communicate with a distant control center in gaining altitude, which is incompatible with a permanent monitoring of its communications since most of the UAV mission takes place close to the ground. Hence, in an application such as the humanitarian convoy the distribution of protocol monitoring and the ability of any agent to monitor part of the protocol, if needed, is a problem that must be addressed. It is not a surprise since the functions of the application themselves have to be implemented as autonomous goal-directed agents to be able to tackle the complexity inherent to this kind of systems. Adding a centralized monitoring is then hopeless.

2 State of the art

Many frameworks and formalisms for monitoring the runtime execution of a distributed system have been proposed in the last years.

One of the most recent and relevant works in this area is SPY (Session Python) [12], a tool chain for runtime verification of distributed Python programs against Scribble (http://www.scribble.org) protocol specifications. Given a Scribble specification of a global protocol, the tool chain validates consistency properties, such as race-free branch paths, and generates Scribble (i.e. syntactic) local protocol specifications for each participant (role) defined in the protocol. At runtime, an independent monitor (internal or external) is assigned to each Python endpoint and verifies the local trace of communication actions executed during the session. This work shares the same motivations and approach with our work, and like our work concentrates on the projection of the global type to the local one rather than on the criteria for identifying in an automatic way how to distribute the monitoring activity. The main differences lie in the expressive power of the two languages, which is higher for constrained global types due to the constrained shuffle operator which is missing in Scribble, and in the availability of tools for statically verifying properties of Scribble specifications, which are not available for constrained global types.

Many other approaches for runtime monitoring of distributed systems and MASs exist like those mentioned in the sequel, but with no emphasis on the projection from global to local monitors. This represents the main difference between those proposals and ours.

In [7], aspect-oriented development techniques are used to enhance existing code of runtime monitors, checking the interaction behavior of applications against their specifications. Message Sequence Charts (MSCs) are exploited to
specify the interaction behavior of distributed systems and as a basis for automatic runtime monitor generation. An explanation of the monitor generation procedure and tool set is presented using a case study from the embedded automotive systems domain. Addressing the need for formal specification and runtime verification of system-level requirements of distributed reactive systems, [5] presents a formalism for specifying global system behaviors in terms of MSCs assertions, with a technique for the evaluation of the likelihood of success of a distributed protocol under non-trivial communication conditions via discrete event simulation and runtime execution monitoring.

Moving to the MAS field, a great attention has been recently devoted to monitoring norms and commitments: formalizing the entities participating to a protocol and the rules regulating their interaction is in fact an inherent aspect of normative systems. In [11] a generic architecture for observing agent behaviors and recognizing those which comply to or violate the predefined norms is described. The architecture deploys monitors that receive inputs from observers and process these inputs together with transition network representations of individual norms. In this way, monitors determine the fulfillment or violation status of norms. As far as commitments are concerned, one of the first contributions were Commitment Machines [15], a formalism modeling communication protocols supplying a content to protocol states and actions in terms of the social commitments of the participants. The content can be reasoned about by the agents thereby enabling flexible execution of the given protocol. In [13] Distributed Commitment Machines are defined and the properties of Commitment Machines, both Distributed and centralized, are explored. A recent work on relationship between agents and commitment-based protocols is [4], where the authors specify agents in terms of goal models and protocols in terms of commitments among agents. The semantic relationship between agents and protocols is formalized exploiting the relationship between goals and commitments. Given an agent specification and a protocol, it is possible to verify whether the protocol allows to achieve particular agent goals, and whether the agent’s specification supports the satisfaction of particular commitments. In [3] commitments are exploited again in normative MASs: the authors focus on one of the best-known agent platforms, Jade (http://jade.tilab.com/), and show that it is possible to account for interactions by exploiting commitment-based protocols, by modifying the Jade Methodology so as to include the new features in a seamless way, and by relying on the notion of artifact.

In [6] a framework for automatic processing of interactions generated using FIPA-ACL (http://www.fipa.org/specs/fipa00061/SC00061G.html) is presented. This framework includes three elements: i) an agent interaction architecture to systematize interaction processing tasks, ii) interaction models to build re-usable validated code used to check different phases of interaction processing associated with message semantics, and iii) components and control structures implementing interaction architecture for a particular agent platform. The paper describes the implementation details of the proposed approach developed within the CAPNET agent platform.
Finally, [10] describes an architecture allowing to verify properties of a multi-
agent system during its execution. Considering that a correct system is a system
verifying the properties specified by the designer, the authors focus on the “prop-
erty” notion. The architecture, a MAS itself, is based on a set of agents whose
goals are to check at runtime the whole system’s properties.

3 Background

This section briefly recaps on constrained global types, omitting their extension
with attributes [8] because the projection algorithm discussed in Section 4 is
currently defined on “plain” constrained global types only.

Constrained global types (also named “types” in the sequel, when no ambig-
uity arises) are defined starting from the following elements:

Interactions\(^1\). An interaction \(a\) is a communicative event taking place between
two agents. For example, \(msg(right\_robot, right\_monitor, tell, put\_sock)\)
is an interaction involving the sender \(right\_robot\) and the receiver \(right\_monitor\),
with performative \(tell\) and content \(put\_sock\).

Interaction types. Interaction types model the message pattern expected at a
certain point of the conversation. An interaction type \(\alpha\) is a predicate on inter-
actions. For example, \(msg(right\_robot, right\_monitor, tell, put\_sock) \in put\_right\_sock\)
means that interaction \(msg(right\_robot, right\_monitor, tell, put\_sock)\) has type \(put\_right\_sock\).

Producers and consumers. In order to model constraints across different
branches of a constrained fork, we introduce two different kinds of interaction
types, called producers and consumers, respectively. Each occurrence of a pro-
ducer interaction type must correspond to the occurrence of a new interaction;
in contrast, consumer interaction types correspond to the same interaction speci-
fied by a certain producer interaction type. The purpose of consumer interaction
types is to impose constraints on interaction traces, without introducing new
events. A consumer is an interaction type, whereas a producer is an interaction
type \(\alpha\) equipped with a natural superscript \(n\) specifying the exact number of
consumer interactions which are expected to coincide with it.

Constrained global types. A constrained global type \(\tau\) represents a set of
possibly infinite traces of interactions, and is a possibly cyclic term defined on
top of the following type constructors:

\begin{itemize}
\item \(\lambda\) (empty trace), representing the singleton set \(\{\epsilon\}\) containing the empty
trace \(\epsilon\).
\item \(\alpha^n: \tau\) (seq-prod), representing the set of all traces whose first element is an
interaction \(a\) matching type \(\alpha\) \((a \in \alpha)\), and the remaining part is a trace
in the set represented by \(\tau\). The superscript\(^2\) \(n\) specifies the number \(n\) of
\end{itemize}

\(^1\) “Interactions” were named “sending actions” in our previous work. We changed
terminology to be consistent with the one used in the choreography community.

\(^2\) In the examples throughout the paper we use the concrete syntax of Prolog where
producer interaction types are represented by pairs \((\alpha, n)\).
corresponding consumers that coincide with the same interaction type α; hence, \( n \) is the least required number of times \( a \in α \) has to be “consumed” to allow a transition labeled by \( a \).

- \( α:τ \) (seq-cons), representing a consumer of interaction \( a \) matching type \( α \) \((a \in α)\).
- \( τ_1 + τ_2 \) (choice), representing the union of the traces of \( τ_1 \) and \( τ_2 \).
- \( τ_1|τ_2 \) (fork), representing the set obtained by shuffling the traces in \( τ_1 \) with the traces in \( τ_2 \).
- \( τ_1 \cdot τ_2 \) (concat), representing the set of traces obtained by concatenating the traces of \( τ_1 \) with those of \( τ_2 \).

Let us consider the following simple example where there are two robots (right and left), two monitors (right and left) associated with each robot, and a plan monitor which supervises them (Figure 1). The goal of the MAS is to help mothers in speeding up dressing their kids by putting their shoes on: robots must put a sock and a shoe on the right (resp. left) foot of the kid they help. As robots are autonomous, they could perform the two actions in the wrong order, making the life of the mothers even more crazy... Monitors are there to ensure that wrong actions are immediately rolled back. Robots communicate their actions to their corresponding monitors, which, in turn, notify the plan monitor when the robots accomplish their goal. Each robot can start by putting the sock, which is the correct action to do, or by putting the shoe, which requires a recovery by the (right or left, resp.) robot monitor.

As we will see, the left and right monitors play two different roles: they interact with robots to detect wrong actions and recover them, and they also verify part of the protocol, notifying the user of protocol violations. In this MAS, monitors are part of the protocol itself. In the MASs described in our previous papers, monitors performed a runtime verification of all the other agents but themselves, and their sole goal was to detect and signal violations. Extending monitors with other capabilities (or, taking another perspective, extending “normal” agents with the capability to monitor part of the protocol) does not represent an extension of the language or framework. The possibility of having agents that can

![Fig. 1. The “socks and shoes” MAS](image-url)
monitor, can be monitored, and can perform whatever other action, was already there, but we did not exploit it before.

The interactions involved in the protocol and their types are as follows:

\[
\begin{align*}
\text{msg(right\_robot, right\_monitor, tell, put\_sock)} & \in \text{put\_right\_sock} \\
\text{msg(right\_robot, right\_monitor, tell, put\_shoe)} & \in \text{put\_right\_shoe} \\
\text{msg(right\_robot, right\_monitor, tell, removed\_shoe)} & \in \text{rem\_right\_shoe} \\
\text{msg(right\_monitor, right\_robot, tell, obl\_remove\_shoe)} & \in \text{obl\_rem\_right\_shoe} \\
\text{msg(right\_monitor, plan\_monitor, tell, ok)} & \in \text{ok\_right} \\
\text{msg(left\_robot, left\_monitor, tell, put\_sock)} & \in \text{put\_left\_sock} \\
\text{msg(left\_robot, left\_monitor, tell, put\_shoe)} & \in \text{put\_left\_shoe} \\
\text{msg(left\_robot, left\_monitor, tell, removed\_shoe)} & \in \text{rem\_left\_shoe} \\
\text{msg(left\_monitor, left\_robot, tell, obl\_remove\_shoe)} & \in \text{obl\_rem\_left\_shoe} \\
\text{msg(left\_monitor, plan\_monitor, tell, ok)} & \in \text{ok\_left}
\end{align*}
\]

The protocol can be specified by the following types, where SOCKS corresponds to the whole protocol.

\[
\begin{align*}
\text{RIGHT} &= ((\text{put\_right\_sock},0):(\text{put\_right\_shoe},0):(\text{ok\_right},0):\lambda) + \\
& ((\text{put\_right\_shoe},0):(\text{obl\_rem\_right\_shoe},0):(\text{rem\_right\_shoe},0):\text{RIGHT}), \\
\text{LEFT} &= ((\text{put\_left\_sock},0):(\text{put\_left\_shoe},0):(\text{ok\_left},0):\lambda) + \\
& ((\text{put\_left\_shoe},0):(\text{obl\_rem\_left\_shoe},0):(\text{rem\_left\_shoe},0):\text{LEFT}), \\
\text{SOCKS} &= (\text{RIGHT} \mid \text{LEFT})
\end{align*}
\]

The type SOCKS specifies the shuffle (symbol “|”) of two sets of traces of interactions, corresponding to RIGHT and LEFT, respectively. The shuffle expresses the fact that interactions in RIGHT are independent (no causality) from interactions in LEFT, and hence traces can be mixed in any order.

Types RIGHT and LEFT are defined recursively, that is, they correspond to cyclic terms. RIGHT consists of a choice (symbol “+”) between the finite trace (the constructor for trace is “:) of interaction types (put\_right\_sock,0), (put\_right\_shoe,0), (ok\_right,0) corresponding to the correct actions of the right robot, and the trace of interaction types (put\_right\_shoe,0), (obl\_rem\_right\_shoe,0), (rem\_right\_shoe,0) corresponding to the wrong initial action of the robot, followed by an attempt to perform the RIGHT branch again. Basically, either the right robot tells the right monitor that it put the sock on first, and then it can go on by putting the shoe, or it tells that it started its execution by putting the shoe on. In this case, the right monitor forces the robot to remove the shoe, the robot acknowledges that it removed the shoe, and then starts again. The LEFT branch is the same as the RIGHT one, but involves the left robot and the left node monitor.

An example where sets of traces could be expressed with a fork, but are not completely independent, is given by the Alternating Bit Protocol ABP. We consider the instance of ABP where six different sending actions may occur (Figure 2): Bob sends msg1 to Alice (interaction type m1), Alice sends ack1 to Bob (sending action type a1), Bob sends msg2 to Carol (interaction type m2), Carol sends ack2 to Bob (sending action type a2), Bob sends msg3 to Dave (interaction type m3), Dave sends ack3 to Bob (interaction type a3) The ABP
is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the sending actions:

- The \( n \)-th occurrence of an interaction of type \( m_1 \) must precede the \( n \)-th occurrence of an interaction of type \( m_2 \) which in turn must precede the \( n \)-th occurrence of an interaction of type \( m_3 \).

- For \( k \in \{1, 2, 3\} \), the \( n \)-th occurrence of \( \text{msg}_k \) must precede the \( n \)-th occurrence of the acknowledge \( \text{ack}_k \), which, in turn, must precede the \( (n + 1) \)-th occurrence of \( \text{msg}_k \).

The ABP cannot be specified with forks of independent interactions, hence a possible solution requires to take all the combinations of interactions into account in an explicit way. However with this solution the size of the type grows exponentially with the number of the different interaction types involved in the protocol.

With producer and consumer interaction types it is possible to express the shuffle of non independent interactions which have to verify certain constraints. In this way the ABP can be specified in a very compact and readable way. The whole protocol is specified by the following constrained global type \( \text{ABP}_3 \):

\[
\text{ABP}_3 = (\text{M}_1 \text{M}_2 \text{M}_3 | \text{M}_1 \text{A}_1) | (\text{M}_2 \text{A}_2 | \text{M}_3 \text{A}_3)
\]

Fork is associative and the way we put brackets in \( \text{ABP}_3 = (\text{M}_1 \text{M}_2 \text{M}_3 | \text{M}_1 \text{A}_1) | (\text{M}_2 \text{A}_2 | \text{M}_3 \text{A}_3) \) does not matter.

## 4 Projection Algorithm

In the “socks and shoes” example the monitors, besides checking that the robots accomplish their goal, verify also the compliance of the system to the specification of the protocol, given by the type \( \text{SOCKS} \). If we assume that the right robot and the right monitor reside on the same node, then it is reasonable that the right monitor verifies only the interactions which are local to its node; to do
that, we must project the type $SOCKS$ onto the agents of the node, that is, the right robot and the right monitor.

What we would like to obtain is the type $\text{RIGHT}_P = ((\text{put}\,\text{right}\,\text{sock},0):(\text{put}\,\text{right}\,\text{shoe},0):(\text{ok}\,\text{right},0)\lambda) + (\text{put}\,\text{right}\,\text{shoe},0):(\text{obl}\,\text{rem}\,\text{right}\,\text{shoe},0):(\text{rem}\,\text{right}\,\text{shoe},0)\text{RIGHT}_P)$,

$\text{SOCKS}_P = (\text{RIGHT}_P|\lambda)$

which only contains interactions where the right robot and the right monitor are involved, either as sender or as receiver.

We can project any protocol onto any set of agents (although it is not necessarily meaningful or useful). For example, projecting the $ABP3$ on Dave should result into

$$\text{ABP3}_P\text{compact} = (m3,0):(a3,0)\text{ABP3}_P\text{compact}$$

which just states that Dave must ensure to respect the order between messages and acknowledges that involve it (Dave cannot be aware of the order among messages coming from other agents). That projected type can be represented in an equivalent way, even if less compact, as

$$\text{M1M2M3}_P = m3:\text{M1M2M3}_P,$$

$$\text{M3A3}_P = (m3,1):(a3,0):\text{M3A3}_P,$$

$$\text{ABP3}_P =((\text{M1M2M3}_P|\lambda)|(\lambda|\text{M3A3}_P))$$

Projecting the $ABP3$ on Bob, instead, should result into the $ABP3$ itself as Bob is involved in all communications and hence no interaction will be removed from the projection.

In order to allow agents to verify only a sub-protocol of the global interaction protocol, we designed a projection algorithm that takes a constrained global type and a set of agents $\text{Ags}$ as input, and returns a constrained global type which contains only interactions involving agents in $\text{Ags}$. The intuition besides the algorithm is that interactions that do not involve agents in $\text{Ags}$ are removed from the projected constrained global type. Given the finite set $\text{AGS}$ of all the agents that could play a role in the MAS and an interaction type $\alpha$, $\text{senders}(\alpha)$ is the set of all the agents in $\text{AGS}$ that could play the role of sender in actual interactions having type $\alpha$ and $\text{receivers}(\alpha)$ is the set of all the agents in $\text{AGS}$ that could play the role of receiver in interactions of type $\alpha$. The $\text{involves}$ predicate holds on one interaction type $\alpha$ and one set of agents $\text{Ags}$, $\text{involves}(\alpha, \text{Ags})$, if $\text{senders}(\alpha) \subseteq \text{Ags}$ and $\text{receivers}(\alpha) \subseteq \text{Ags}$.

Projection can be described as a function $\Pi : \mathcal{CT} \times \mathcal{P}^{\text{AGS}} \rightarrow \mathcal{CT}$ where $\mathcal{CT}$ is the set of constrained global types. $\Pi$ is driven by the syntax of the type to project; as a first attempt, the function could be coinductively defined as follows:

(i) $\Pi(\lambda, \text{Ags}) = \lambda$

(ii) $\Pi(\alpha : \tau, \text{Ags}) = \alpha : \Pi(\tau, \text{Ags})$ if $\text{involves}(\alpha, \text{Ags})$

(iii) $\Pi(\alpha : \tau, \text{Ags}) = \Pi(\tau, \text{Ags})$ if $\neg\text{involves}(\alpha, \text{Ags})$

(iv) $\Pi(\tau' \text{ op } \tau'', \text{Ags}) = \Pi(\tau', \text{Ags}) \text{ op } \Pi(\tau'', \text{Ags})$, where $\text{op} \in \{+, |, \cdot\}$. 
We have to consider the greatest fixed point (coinductive interpretation) of the recursive definition above, since the least fixed point (inductive interpretation) would only include non cyclic types (that is, non recursive types).

Let us consider a simple non recursive term $T$ defined by $T = a : b : \lambda$. We want to project $T$ on $\text{Ags}$. Suppose for that $\text{involves}(a, \text{Ags})$ holds, whereas $\text{involves}(b, \text{Ags})$ does not, meaning that interaction type $a$ must be kept in the projection and $b$ must be removed. From (ii) we get $\Pi(a : b : \lambda, \text{Ags}) = a : \Pi(b : \lambda, \text{Ags})$ ($a$ is kept in the projection), from (iii) we have $\Pi(b : \lambda, \text{Ags}) = \Pi(\lambda)$ ($b$ is discarded from the projection), and finally, from (i) we know that $\Pi(\lambda) = \lambda$, therefore $\Pi(T, \text{Ags}) = a : \lambda$.

Let us now consider the recursive type $T$ s.t. $T = a : T'$ and $T' = b : T$. Again, the projection is driven by the syntax of $T$; from the definition above we have $\Pi(a : T', \text{Ags}) = a : \Pi(T', \text{Ags}) = a : \Pi(b : T, \text{Ags}) = a : \Pi(T) = a : \Pi(a : T', \text{Ags})$; while in the previous case we can conclude by applying the base case corresponding to the $\lambda$ type, in this case we do not have any basis, but we can conclude by coinduction that $\Pi(a : T', \text{Ags})$ has to return the unique recursive type $T''$ s.t. $T'' = a : T''$ (see lhs picture in Figure 3).

The definition above however needs to be refined because it does not always specify a unique result for $\Pi$; to see that, let us consider the recursive type $T$ s.t. $T = a : T'$ and $T' = b : T'$. Now from the definitions above we get $\Pi(a : T', \text{Ags}) = a : \Pi(T', \text{Ags})$, $\Pi(T', \text{Ags}) = \Pi(b : T', \text{Ags}) = \Pi(T', \text{Ags})$; since $\Pi(T', \text{Ags}) = \Pi(T', \text{Ags})$ is an identity, $\Pi$ is allowed to return any type when applied to $T'$, while the expected correct type should be $\lambda$, so that $\Pi(a : T', \text{Ags}) = a : \lambda$ (see rhs picture in Figure 3).

Finally, let us consider the recursive type $T$ s.t. $T = (a : T) + (b : T)$; by (iv) $\Pi(T, \text{Ags}) = \Pi(a : T, \text{Ags}) + \Pi(b : T, \text{Ags})$, by (ii) $\Pi(a : T, \text{Ags}) = a : \Pi(T, \text{Ags})$, and by (iii) $\Pi(b : T, \text{Ags}) = \Pi(T, \text{Ags})$, therefore by coinduction the returned type is $T''$ s.t. $T'' = (a : T') + T'$; although in this case there exists a unique type that can returned by $\Pi$, such a type is not contractive. A type is contractive if all possible cycles in it contain an occurrence of the sequence constructor “+”; Figure 4 shows that type $T''$ s.t. $T'' = (a : T') + T'$ is not contractive, since the rhs cycle contains only the “+” operator.

The notion of contractive type is crucial for implementing efficient runtime verification.

To ensure that the projection function always returns a contractive type and that the correct coinductive definition is implemented, we need to keep track of
all types visited along a path; each type is associated with its depth, and with a fresh variable which will be unified with the corresponding computed projection. During the visit the depth $\text{DeepestSeq}$ of the deepest visited sequence operator is kept. If a type $\tau$ has been already visited, then a cycle is detected: if its depth is less then $\text{DeepestSeq}$ then the cycle contains an occurrence of the sequence constructor, therefore the projected type associated with $\tau$ is contractive and, hence, is returned; otherwise, the projection would not be contractive, therefore $\lambda$ is returned.

Let us consider again the type $T = (a : T) + (b : T)$; when computing its projection, the depth of $T$ is 0, and initially $\text{DeepestSeq}$ contains the value -1. When visiting the lhs path starting from the “+” operator, the type $a : T$ is visited at depth 1, and $\text{DeepestSeq}$ is set to 1, since the root of $a : T$ is the sequence constructor. Then $T$ is revisited, and since its depth 0 is less then $\text{DeepestSeq}$, the projection of the lhs is $T' = a : T'$. When visiting the rhs path starting from the “+” operator, $\text{DeepestSeq}$ contains again the value -1, and the type $b : T$ is visited at depth 1, but because $\text{involves}(b, \text{Ags})$ does not hold, $b$ is discarded with the corresponding sequence constructor, hence $\text{DeepestSeq}$ is not updated. Then $T$ is revisited, and since its depth 0 is not less then $\text{DeepestSeq}$, the projection of the rhs is $\lambda$.

5 Implementation

The projection algorithm has been implemented in SWI Prolog, http://www.swi-prolog.org/, which manages infinite (cyclic, recursive) terms in an efficient way. Since we need to record the association between any type and its projection in order to correctly detect and manage cycles, we exploited the SWI Prolog library assoc for association lists, http://www.swi-prolog.org/pldoc/man?section=assoc. Elements of an association list have 2 components: a (unique) key and a value. Keys should be ground, values need not be. An association list can be used to fetch elements via their keys and to enumerate its elements in ascending order of their keys. The library(assoc) module uses AVL trees to implement association lists which makes inserting, changing and fetching a single element an $O(\log(N))$ operation. The three predicates of the library assoc that we use for our implementation are

- empty_assoc(-Assoc): Assoc is unified with an empty association list.
- **get_assoc**(Key, Assoc, Value): Value is the value associated with Key in the association list Assoc.

- **put_assoc**(Key, Assoc, Value, NewAssoc): NewAssoc is an association list identical to Assoc except that Key is associated with Value. This can be used to insert and change associations.

The projection is implemented by a predicate **project**(T, ProjAgs, ProjT) where T is the constrained global type to be projected, ProjT is the result, and ProjAgs is the set of agents onto which the projection is performed. The algorithm exploits the predicate **involves**(IntType, ProjAgs) succeeding if IntType may involve one agent, as a sender or a receiver, in ProjAgs.

Currently **involves** looks for actual interactions **ActInt** whose type is IntType and assumes that senders and receivers in **ActInt** are ground terms, but it could be extended to take agents’ roles into account or in other more complex ways. It uses the “or” Prolog operator ; and the **member** predicate offered by the library **lists**. It exploits the predicate **has_type**(ActInt, IntType) implementing the definition of the type IntType of an actual interaction **ActInt**.

\[
\text{involves(IntType, List)} :\neg \\
\text{has_type(msg(Sender, Receiver, _, _), IntType),} \\
\text{(member(Sender, List);member(Receiver, List)).}
\]

For the implementation of **project/3** we use an auxiliary predicate **project** with six arguments, which are the same as those of the main predicate plus

- an initially empty association A to keep track of cycles;
- the current depth of the constrained global type under projection, initially set to 0;
- the depth of the deepest sequence operator belonging to the projected type, initially set to -1.

```
project(T, ProjAgs, ProjT) :-
empty_assoc(A), project(A, 0, -1, T, ProjAgs, ProjT).
```

The predicate is defined by cases.

1. **lambda** is projected into **lambda**.

\[
\text{project(Assoc, _Depth, _DeepestSeq, lambda, _ProjAgs, lambda):- !.}
\]

2. If Type has been already met while projecting the global type (**get_assoc**(Type, Assoc, (AssocProjType, LoopDepth)) succeeds), then its projection ProjT is AssocProjType if LoopDepth =< DeepestSeq and is **lambda** otherwise. The “if-then-else” construct is implemented in Prolog as **Condition** -> **ThenBranch** ; **ElseBranch**.

\[
\text{project(Assoc, _Depth, DeepestSeq, Type, _ProjAgs, ProjT) :-} \\
\text{get_assoc(Type, Assoc, (AssocProjType,LoopDepth))},!, \\
\text{(LoopDepth =< DeepestSeq -> ProjT=AssocProjType; ProjT=lambda).}
\]

227
3. \( T = (\text{IntType}:T_1) \). \text{IntType} is a consumer as it has no integer number associated with it. Project\( T \) is recorded in the association \( A \) along with the current depth \( \text{Depth} \). (put\_assoc\((\text{IntType}:T_1), A, (\text{ProjT}, \text{Depth}), \text{NewAssoc}\)). If \text{IntType} involves \text{ProjAgs}, \text{ProjT}=\( (\text{IntType}:\text{ProjT}_1) \) where \text{ProjT}_1 is obtained by projecting \( T_1 \) onto \text{ProjAgs}, with association \text{NewAssoc}, depth of the type under projection increased by one, and depth of the deepest sequence operator equal to \text{Depth}. If \text{IntType} does not involve \text{ProjAgs}, then the projection on \( T \) is the same of \( T_1 \) with association \text{NewAssoc}, depth of the type under projection equal to \text{Depth}, and depth of the deepest sequence operator equal to \text{DeepestSeq}.

\[
\text{project}(A, \text{Depth}, \text{DeepestSeq}, (\text{IntType}:T_1), \text{ProjAgs}, \text{ProjT}) :- !, \phantom{1}\text{put\_assoc}\((\text{IntType}:T_1), A, (\text{ProjT}, \text{Depth}), \text{NewAssoc}\), \phantom{1}(\text{involves}(AMsg, \text{ProjAgs}) \rightarrow \text{IncDepth} \leftarrow \text{Depth}+1, \phantom{1}\text{project}(\text{NewAssoc}, \text{IncDepth}, \text{Depth}, T_1, \text{ProjAgs}, \text{ProjT}_1), \phantom{1}\text{ProjT}=\( (\text{IntType}:\text{ProjT}_1) \); \phantom{1}\text{project}(\text{NewAssoc}, \text{Depth}, \text{DeepestSeq}, T_1, \text{ProjAgs}, \text{ProjT}) \).
\]

4. \( T = ((\text{IntType},N):T_1) \). \text{IntType}, \( N \) is a producer as it has an integer number \( N \) associated with it. The projection is identical to the previous case, apart from the fact that \text{ProjT}=\( ((\text{IntType},N):\text{ProjT}_1) \) in the first branch of the condition in the clause’s body.

5. \( T = T_1 \text{ op } T_2 \), where \( \text{op} \in \{+, |, \ast\} \): the association between \( T_1 \text{ op } T_2 \) and the projected type \text{ProjT} is recorded in the association \( A \) along with the current depth \( \text{Depth} \). \( T_1 \) and \( T_2 \) are projected into \text{ProjT}_1 and \text{ProjT}_2 respectively, with association equal to \text{NewAssoc}, depth of the type under projection increased by one and depth of the deepest sequence operator equal to \text{DeepestSeq}. The result of the projection is \text{ProjT}=\( (\text{ProjT}_1 \text{ op } \text{ProjT}_2) \). For example, if \( \text{op} \) is +, the Prolog clause is:

\[
\text{project}(A, \text{Depth}, \text{DeepestSeq}, (T_1+T_2), \text{ProjAgs}, \text{ProjT}) :- !, \phantom{1}\text{put\_assoc}\((T_1+T_2), A, (\text{ProjT}, \text{Depth}), \text{NewAssoc}\), \phantom{1}\text{IncDepth} \leftarrow \text{Depth}+1, \phantom{1}\text{project}(\text{NewAssoc}, \text{IncDepth}, \text{Depth}, T_1, \text{ProjAgs}, \text{ProjT}_1), \phantom{1}\text{ProjT}=\( (\text{ProjT}_1+\text{ProjT}_2) \).
\]

Types \text{SOCKS}\_P and \text{AP3}\_P shown at the beginning of Section 4 have been obtained by applying the projection algorithm to types \text{SOCKS} and \text{ABP3} respectively. The reason why they are not as compact as possible, which is mainly evident in \text{AP3}\_P, is that the projection algorithm does not implement a further normalization step and hence some types which have been projected into \text{lambda} and might be removed, are instead kept.

The result of the projection may be a type equivalent to \text{lambda}. For example, if we project \text{ABP} to the set \{\text{eric}\}, no interaction involves it and the result is \text{lambda}|\text{lambda}|\text{lambda}|\text{lambda}. On the other hand, we have already observed that the projection may be the same as the projected type. This happens for example if we project \text{ABP} to the set \{\text{bob}\}, which interacts with all the agents in the MAS.
6 Projection at Work

6.1 Design Time Experiments with SWI Prolog

In SWI Prolog we have implemented a mechanism for generating all the different traces (sequences of interactions) with length N, where N can be set by the user, that respect a given protocol. This mechanism is necessary during the design of the protocol and allows the protocol designer to make an empirical assessment of the conversations that will be recognized as valid ones during the runtime verification. We used this mechanism for validating both the complete protocols and the projected ones; also with projected types, the generated traces are correct w.r.t. the protocol specification.

<table>
<thead>
<tr>
<th>SOCKS protocol</th>
<th>SOCKS protocol projected onto {right_robot, right_monitor}</th>
</tr>
</thead>
<tbody>
<tr>
<td>m(right_r, right_m, put_sock)</td>
<td>m(right_r, right_m, put_shoe)</td>
</tr>
<tr>
<td>m(left_r, left_m, put_shoe)</td>
<td>m(right_m, right_r, oblige_remove_shoe)</td>
</tr>
<tr>
<td>m(left_m, left_r, oblige_remove_shoe)</td>
<td>m(right_r, right_m, removed_shoe)</td>
</tr>
<tr>
<td>m(left_robot, left_m, removed_shoe)</td>
<td>m(right_r, right_m, put_shoe)</td>
</tr>
<tr>
<td>m(right_r, right_m, put_shoe)</td>
<td>m(right_r, right_m, removed_shoe)</td>
</tr>
<tr>
<td>m(right_m, plan_monitor, ok)</td>
<td>m(right_r, right_m, removed_shoe)</td>
</tr>
<tr>
<td>m(left_robot, left_m, put_shoe)</td>
<td>m(right_r, right_m, put_shoe)</td>
</tr>
<tr>
<td>m(left_m, left_r, oblige_remove_shoe)</td>
<td>m(right_r, right_m, oblige_remove_shoe)</td>
</tr>
<tr>
<td>m(left_r, left_m, removed_shoe)</td>
<td>m(right_r, right_m, removed_shoe)</td>
</tr>
<tr>
<td>m(left_r, left_m, put_sock)</td>
<td>m(left_m, plan_monitor, ok)</td>
</tr>
<tr>
<td>m(left_r, left_m, put_shoe)</td>
<td>m(right_r, right_m, put_shoe)</td>
</tr>
<tr>
<td>m(left_m, plan_monitor, ok)</td>
<td>m(right_r, right_m, put_shoe)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ABP3 protocol</th>
<th>ABP3 protocol projected onto {dave}</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg(bob, alice, tell, m1)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(bob, carol, tell, m2)</td>
<td>msg(bob, dave, tell, a3)</td>
</tr>
<tr>
<td>msg(carol, bob, tell, a2)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(alice, bob, tell, a1)</td>
<td>msg(dave, bob, tell, a3)</td>
</tr>
<tr>
<td>msg(bob, dave, tell, m3)</td>
<td>msg(bob, dave, tell, a3)</td>
</tr>
<tr>
<td>msg(bob, alice, tell, m1)</td>
<td>msg(dave, bob, tell, a3)</td>
</tr>
<tr>
<td>msg(alice, bob, tell, a1)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(bob, dave, tell, m3)</td>
<td>msg(bob, dave, tell, a3)</td>
</tr>
<tr>
<td>msg(bob, alice, tell, m1)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(carol, bob, tell, a2)</td>
<td>msg(dave, bob, tell, a3)</td>
</tr>
<tr>
<td>msg(dave, bob, tell, a3)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(dave, bob, tell, m3)</td>
<td>msg(dave, bob, tell, a3)</td>
</tr>
<tr>
<td>msg(alice, bob, tell, a1)</td>
<td>msg(dave, bob, tell, m3)</td>
</tr>
<tr>
<td>msg(bob, carol, tell, m2)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
<tr>
<td>msg(alice, bob, tell, a1)</td>
<td>msg(bob, dave, tell, m3)</td>
</tr>
</tbody>
</table>

Table 1. Examples of traces compliant with complete and projected protocols.
For example, Table 1 (top left) shows one of the 16380 different traces with length 12 of the SOCKS protocol and Table 1 (top right) shows one of the 2 different traces with length 12 of the SOCKS protocol projected onto \{right_robot, right_monitor\} (for sake of presentation, we abbreviate right_robot in right_r, right_monitor in right_m, left_robot in left_r, left_monitor in left_m, msg in m, and we drop the tell performative from interactions). Both traces correspond to an execution where the protocol reached a final state and no other interactions could be accepted after the last one. In the output produced by the SWI Prolog algorithm, this information is given by means of an asterisk after the last interaction. Traces that are prefixes of longer (maybe infinite) ones have no asterisk at their end.

Table 1 (bottom left) shows one of the 30713 different traces with length 16 of the ABP3 protocol and Table 1 (bottom right) shows the only trace with length 16 of the ABP3 protocol projected onto \{dave\}. Since the ABP3 is an infinite protocol, both traces are prefixes of infinite ones.

By generating traces of different length and inspecting some of them, the protocol designer can get a clear picture of whether the protocol he/she designed behaves in the expected way. Of course this manual inspection gives no guarantees of correctness, but in our experience it was enough to early detect flaws in the protocol specification.

### 6.2 Runtime Experiments with Jason

We have implemented the “socks and shoes” MAS in Jason. The MAS is represented in Figure 1. We projected the SOCKS constrained global type shown in Section 3 onto the three sets of agents \{left_monitor\}, \{right_monitor\} and \{plan_monitor\}. The three resulting constrained global types are used by agents left_monitor, right_monitor and plan_monitor respectively.

Each of these agents monitors all the messages that it either receives or sends, using the “message sniffing” mechanism described in [1].

We run different experiments by changing the actual messages sent by the agents in the MAS, in order to obtain both correct and wrong executions. As an example, Figure 5 shows the output of an interaction where the right_monitor sends a message with content very_good to the plan_monitor, instead of the ok content foreseen by the protocol. Figure 6 shows an interaction where left_robot sends a put_boot message instead of put_shoe, which is correctly identified by the left_monitor as a violation. The conversation between the other agents goes on.

### 6.3 Methodological issues

In the case of the SOCKS protocol, deciding which were the subsets of agents onto which projecting the global protocol in order to distribute the monitoring activity was easy: interactions induce a graph connecting pairs of agents that interact at some point, and in this case the graph is a tree as shown in Figure 1. By projecting onto \{left_monitor\} and allowing left_monitor to
monitor its own interactions, we make a complete check of the left branch of the tree. In the same way, by projecting onto \{right_monitor\} and allowing right_monitor to monitor its own interactions, we make a complete check of the right branch. Projecting onto \{plan_monitor\} in this case would be useless, as interactions with this agent are already checked by the left and right monitors and the plan_monitor does not perform further checks - in particular, it does not check that messages from the left and right monitor arrive in some specific order. However, projecting onto \{plan_monitor\} would make sense if the MAS were a “sub-MAS” of a larger system, where more couples of robots exist. In that case, we might expect that each plan monitor would report the outcome of activities of its couple of robots to an agent higher in the hierarchy. Interactions with this top-level agent should be monitored by the plan monitor (or vice-versa) and should be transparent to the agents monitoring the robots.

In the MAS implementing the ABP3 protocol shown in Figure 2 things are different due to the constraints in the fork. Although interactions induce a tree
Fig. 6. The **left_robot** violates the protocol.

like in the SOCKS case, projecting onto Alice, Carol and Dave and allowing these three agents to check their own interactions would not be enough to verify all the protocol’s constraints. In fact, these agents could verify that messages and respective acknowledges are exchanged in the correct order, but none of them alone can verify that interaction \( m_1 \) takes place before \( m_2 \) and that \( m_2 \) takes place before \( m_3 \). The only projection which ensures that all the protocol constraints are verified is the one onto Bob, namely the entire protocol. The ABP3 cannot be distributed, hence we need a centralized monitor (which might be an external monitor or Bob himself, as it is involved in all the interactions) that “sniffs” the interactions among all the agents and verifies their compliance to the ABP3. None prevents us from projecting ABP3 also onto Alice, Carol and Dave and asking them to monitor the part of the protocol where they are involved, but this would be a useless redundancy, as Bob (or the external monitor) would already verify their part.

Distributing the monitoring activity in order to guarantee that it is equivalent to the centralized one requires building the “interaction graph” and identifying connected sub-graphs such that all and only interactions in that sub-graph can
be captured by projecting the global protocol onto one subset of agents, and allowing one agent to monitor interactions involving agents in that subset. Many partitions of this kind can be found, but not all of them ensure that the global protocol is verified, and in some cases like the ABF3 the protocol cannot be distributed at all. Identifying the criteria for stating whether a protocol can be distributed or not, and how it can be distributed if possible, is part of our close future work.

7 Conclusions and Future Work

In this paper we have defined an algorithm for projecting a constrained global type onto a set of agents $Ags$, to allow distributed dynamic verification of the compliance of a MAS to a protocol. This is important in communication-intensive and highly-distributed large MASs, where a centralized approach with a unique monitoring agent would be unfeasible.

Besides describing the algorithm and its SWI Prolog implementation, we have shown some preliminary experiments in Jason with the running example “socks and shoes” where two local monitors with projected types are sufficient for verifying the whole system.

For what concerns future work, we are investigating on the possible ways to partition the set of agents for projecting types, to minimize the number of monitors, while ensuring safety of dynamic verification.

We are also planning to extend the projection algorithm in order to be able to properly deal with a more general form of type: attribute global types.

Finally, in the examples considered in this paper, types are projected statically (that is, before the system is started) because we have assumed that agents cannot move between nodes, but monitoring would be also possible in the presence of agent mobility, as described in the scenario outlined in the introduction. However, in this case the implementation of a self-monitoring MAS is more challenging, because monitor agents have to dynamically project the global type in reaction to any change involving the set of monitored agents. Tackling scenarios of this kind is the final goal of our research.

References


Infinite States Verification in Game-Theoretic Logics: Case Studies and Implementation

Slawomir Kmiec and Yves Lespérance
Dept. of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada
skmiec@cse.yorku.ca lesperan@cse.yorku.ca

Abstract. Many practical problems where the environment is not in the system’s control can be modelled in game-theoretic logics (e.g., ATL). But most work on verification methods for such logics is restricted to finite state cases. De Giacomo, Lespérance, and Pearce have proposed a situation calculus-based logical framework for representing such infinite state game-type problems together with a verification method based on fixpoint approximates and regression. Here, we extend this line of work. Firstly, we describe some case studies to evaluate the method. We specify some example domains and show that the method does allow us to verify various properties. We also find some examples where the method must be extended to exploit information about the initial state and state constraints in order to work. Secondly, we describe an evaluation-based Prolog implementation of a version of the method for complete initial state theories with the closed world assumption. It generates successive approximates and checks if they hold in the situation of interest. We describe some preliminary experiments with this tool and discuss its limitations.

1 Introduction

Many practical problems where the environment is not completely under the system’s control, such as service orchestration, contingent planning, and multi-agent planning, can be modeled as games and specified in game-theoretic logics. There has been much work to define such logics (e.g., Alternating-Time Temporal Logic (ATL)) and develop verification methods for them, mainly model checking techniques [1]. However, most such work is restricted to finite state settings. De Giacomo, Lespérance, and Pearce [2] (hereafter DLP) have developed an expressive logical framework for specifying such problems within the situation calculus [3]. In their approach, a game-like problem/setting is represented as a situation calculus game structure, a special kind of action theory that specifies who the players are, what the legal moves are, etc. They also define a logic that combines the $\mu$-calculus, game-theoretic path quantifiers (as in ATL), and first-order quantification, for specifying properties about such game settings. Additionally, they propose a procedural language for defining game settings, GameGolog, which is based on ConGolog [4]. Finally, they propose a method for verifying temporal properties over infinite state game structures that is based on fixpoint approximates and regression.

While DLP give examples to illustrate the expressiveness and convenience of their formalism, they recognize that their work is essentially theoretical and call for experimental studies to understand whether these techniques actually work in practice. This is
what we begin to address in this paper. We develop several example problems involving
infinite state domains and represent them as situation calculus game structures. We then
examine whether the DLP fixpoint approximates verification method works to verify
common temporal properties. In many cases, it does indeed work. So to some extent,
our work validates the DLP proposal.

We do however find other examples where the DLP method does not converge in a
finite number of steps. We note that the method uses only the simplest part of the ac-
tion theory, the unique name and domain closure axioms, to try to show that successive
approximates are equivalent (after performing regression). Clearly, using the whole ac-
tion theory is problematic as it includes a second-order axiom to specify the domain of
situations. We show that in some cases, adding a few key facts that are entailed by the
entire theory (from simple axioms about the initial state to state constraints proven by
induction) is sufficient to get convergence in a finite number of steps. This means that
the method can be used successfully in a wider range of problems if we can rely on
the modeler to identify such facts. Thus, our case studies show that the kind of method
introduced in [2] often does work for infinite domains, where very few verification
methods are available, and allow reasoning about a range of game problems. Note that
in our case studies, the fixpoint approximation method was performed manually. We
also describe an evaluation-based Prolog implementation of a version of the method for
complete initial state theories with the closed world assumption. It generates succes-
sive approximates and checks if they hold in the situation of interest. We describe some
experiments with this tool and discuss its limitations.

2 Situation Calculus Game Structures

The situation calculus (SitCalc) is a many-sorted predicate logic language for represent-
ing dynamically changing worlds in which all changes are the result of named actions
[3, 5]. Actions are terms in the language, e.g., pickup(R, X) could represent an action
where a robot R picks up an object X. Action terms are denoted by α possibly with
subscripts to differentiate different action terms. Action variables are denoted by lower
case letters a possibly with subscripts. Action types, i.e., actions functions, which may
require parameters, are denoted by upper case letters A possibly with subscripts. Situa-
tions represent possible world histories and are terms in the language. The distinguished
constant S0 denotes the initial situation where no action has yet been performed. The
distinguished function symbol do is used to build sequences of actions such that do(a, s)
denotes the successor situation that results from performing action a in situation s. Flu-
ents are predicates or functions whose values may vary from situation to situation. They
are denoted by symbols that take a situation term as their last argument.

Given this language, one can specify action theories that describe how the world
changes as the result of the available actions. We focus on basic action theories as
proposed in [5]. We assume that there is a finite number of action types in the domains
we consider. Thus, a basic action theory D is the union of the following disjoint sets: the
foundational, domain independent axioms of the situation calculus (Σ); precondition
axioms stating when actions are executable (Dposs); successor state axioms describing
how fluents change between situations (Dssa); unique name axioms for actions and
domain closure on action types \( \mathcal{D}_{\text{ca}} \); and axioms describing the initial configuration of the world \( \mathcal{D}_{\mathcal{S}_0} \). Successor state axioms specify the value of fluents in situation \( \text{do}(a, s) \) in terms of the action \( a \) and the value of fluents in situation \( s \); they encode the causal laws of the world and provide a solution to the frame problem.

Situation calculus game structures, proposed by DLP, are a specialization of basic action theories that allow multi-agent game-like settings to be modeled. In SitCalc game structures, every action \( a \) has an agent parameter and the distinguished function \( \text{agent}(a) \) returns the agent of the action. Axioms for the \( \text{agent} \) function are specified for every action type and by convention the agent parameter is the first argument of any action type. It is assumed that there is a finite set \( \text{Agents} \) of agents who are denoted by unique names. Actions are divided into two groups: choice actions and standard actions. Choice actions model the decisions of agents and they are assumed to have no effect on any fluent other than \( \text{Poss}, \text{Legal}, \) and \( \text{Control} \). Standard actions are the other non-choice actions. \( \text{Poss}(a, s) \) specifies that an action \( a \) is physically possible (i.e. executable) in situation \( s \). Choice actions are always physically possible. There is also a distinguished predicate \( \text{Legal}(s) \) that is a stronger version of possibility/legality and models the game structure of interest. It specifies what actions an agent may execute and what choices can be made according to the rules of the game. The axioms provided for \( \text{Legal} \) specify the game of interest. It is required that the axioms for \( \text{Legal} \) entail 3 properties: 1) \( \text{Legal} \) implies physically possible \( (\text{Poss}) \), 2) legal situations are the result of an action performed in legal situations, and 3) only one agent can act in a legal situation. \( \text{Control}(agt, s) \) holds if agent \( agt \) is the one that is in control and can act in a legal situation \( s \); it is defined as follows: 

\[
\text{Control}(agt, s) \equiv \exists a. \text{Legal}(\text{do}(a, s)) \land \text{agent}(a) = agt.
\]

As a result of the above constraints on \( \text{Legal} \), it follows that the predicate \( \text{Control} \) holds for only one agent in any given legal situation. As explained in DLP, games where several agents act simultaneously can be modeled using a round-robin of choice actions. If the result of such simultaneous choices is non-deterministic, a “game master” agent that makes the decision can be introduced. It is worth noting that the state of the game in situation \( s \) is captured by the fluents. Finally, DLP define a SitCalc game structure to be an action theory \( \mathcal{D}_{\text{GS}} = \Sigma \cup \mathcal{D}_{\text{poss}} \cup \mathcal{D}_{\text{ssa}} \cup \mathcal{D}_{\text{ca}} \cup \mathcal{D}_{\mathcal{S}_0} \cup \mathcal{D}_{\text{legal}} \), where \( \mathcal{D}_{\text{legal}} \) contains the axioms for \( \text{Legal} \) and \( \text{Control} \) and for the function \( \text{agent}() \), and the other components are as for standard basic action theories. Note that here a game structure is a type of situation calculus theory and not a single game model as is often the case.

DLP introduces a logical language \( L \) for expressing temporal properties of game structures. It is inspired by ATL [1] and based on the \( \mu \)-calculus [6], as used over game structures as in [7]. The key element of the \( L \)-logic is the \( \langle \langle G \rangle \rangle \circ \varphi \) operator defined as follows:

\[
\langle \langle G \rangle \rangle \circ \varphi \equiv \\
(\exists agt \in G. \text{Control}(agt, \text{now}) \land \\
\exists a. \text{agent}(a) = agt \land \text{Legal}(\text{do}(a, \text{now})) \land \varphi[do(a, \text{now})]) \lor \\
(\exists agt \notin G. \text{Control}(agt, \text{now}) \land \\
\forall a. \text{agent}(a) = agt \land \text{Legal}(\text{do}(a, \text{now})) \supset \varphi[do(a, \text{now})])
\]

This operator, in essence, specifies that a coalition \( G \) of agents can ensure that \( \varphi \) holds next, i.e., after one more action, as follows. If an agent from the coalition \( G \) is in control
in the current situation, then all we need is that there be some legal action that this agent can perform to make the formula \( \varphi \) hold. If the agent in control is not in coalition \( G \), then what we need is that regardless of the action taken by the in-control agent (for all) the formula \( \varphi \) holds after the action. The whole logic \( \mathcal{L} \) is defined as follows:

\[
\Psi ::= \varphi \mid Z(x) \mid \Psi_1 \land \Psi_2 \mid \Psi_1 \lor \Psi_2 \mid \exists x. \Psi \mid \forall x. \Psi \mid (\langle G \rangle \circ \Psi) \mid [\lceil G \rceil] \circ \Psi \mid \mu Z(x). \Psi(Z(x)) \mid \nu Z(x). \Psi(Z(x)).
\]

In the above, \( \varphi \) is an arbitrary, possibly open, situation-suppressed situation calculus uniform formula. \( Z \) is a predicate variable of a given arity. \( \langle G \rangle \circ \Psi \) is as defined above, \([\lceil G \rceil] \circ \Psi \) is the dual of \( \langle G \rangle \circ \Psi \) (i.e., \([\lceil G \rceil] \circ \Psi \equiv \neg \langle G \rangle \circ \neg \Psi \)), and \( \mu \) (resp. \( \nu \)) is the least (resp. greatest) fixpoint operator from the \( \mu \)-calculus, where the argument is written as \( \Psi(Z(x)) \) to emphasize that \( Z(x) \) may occur free, i.e., not quantified by \( \mu \) or \( \nu \), in \( \Psi \).

The language \( \mathcal{L} \) allows one to express arbitrary temporal/dynamic properties. For example, the property that group \( G \) can ensure that eventually \( \varphi(x) \) (or has a strategy to achieve \( \varphi(x) \)), where \( \varphi(x) \) is a situation suppressed formula with free variables \( x \), may be expressed by the following least fixpoint construction:

\[
\langle \langle G \rangle \rangle \circ \varphi(x) = \mu Z(x) \cdot \varphi(x) \lor \langle \langle G \rangle \rangle \circ Z(x)
\]

Similarly, group \( G \)'s ability to maintain a property \( \varphi(x) \) can be expressed by the following greatest fixpoint construction:

\[
\langle \langle G \rangle \rangle \Box \varphi(x) = \nu Z(x) \cdot \varphi(x) \land \langle \langle G \rangle \rangle \circ Z(x)
\]

We say that there is a path where \( \varphi(x) \) holds next if the set of all agents can ensure that \( \varphi(x) \) holds next: \( \exists \circ \varphi(x) \equiv \langle \langle \text{Agents} \rangle \rangle \circ \varphi(x) \). Similarly there is a path where \( \varphi(x) \) eventually holds if the set of all agents has a strategy to achieve \( \varphi(x) \): \( \exists \Box \varphi(x) \equiv \langle \langle \text{Agents} \rangle \rangle \Box \varphi(x) \).

DLP propose a procedure based on regression and fixpoint approximation to verify formulas of logic \( \mathcal{L} \) given a SitCalc game structure theory. This recursive procedure \( \tau(\Psi) \) tries to compute a first-order formula uniform in current situation now that is equivalent to \( \Psi \):

\[
\begin{align*}
\tau(\varphi) &= \varphi & \tau(Z) &= Z \\
\tau(\Psi_1 \land \Psi_2) &= \tau(\Psi_1) \land \tau(\Psi_2) \\
\tau(\Psi_1 \lor \Psi_2) &= \tau(\Psi_1) \lor \tau(\Psi_2) \\
\tau(\exists x. \Psi) &= \exists x. \tau(\Psi) & \tau(\forall x. \Psi) &= \forall x. \tau(\Psi) \\
\tau(\langle G \rangle \circ \Psi) &= \mathcal{R}(\langle \lceil G \rceil \rangle \circ \tau(\Psi)) \\
\tau([\lceil G \rceil] \circ \Psi) &= \neg \mathcal{R}(\langle \lceil G \rceil \rangle \circ \tau(\neg \Psi)) \\
\tau(\mu Z. \Psi) &= \text{lfp} Z. \tau(\Psi) & \tau(\nu Z. \Psi) &= \text{gfp} Z. \tau(\Psi)
\end{align*}
\]

In the above, \( \mathcal{R} \) represents the regression operator and \( \langle \lceil G \rceil \rangle \circ \Psi \) is regresable if \( \Psi \) is regresable, \( \text{NNF}(\neg \Psi) \) denotes the negation normal form of \( \neg \Psi \), and \( \text{lfp} Z. \Psi \) and \( \text{gfp} Z. \Psi \).

---

1. Although \( \langle \langle G \rangle \rangle \circ \neg \Psi \) is not in \( \mathcal{L} \) according to the syntax, the equivalent formula in negation normal form is.
gfp \cdot \Psi are formulas resulting from the following least and greatest fixpoint procedures:

\[ \text{lfp} Z. \Psi = \]
\[ R := \text{False}; R_{\text{new}} := \Psi(\text{False}); \]
\[ \text{while} (D_{\text{ca}} \not\models R \equiv R_{\text{new}}) \{
\]
\[ R := R_{\text{new}}; R_{\text{new}} := \Psi(R) \}
\[ \}
\]
\[ \text{gfp} Z. \Psi = \]
\[ R := \text{True}; R_{\text{new}} := \Psi(\text{True}); \]
\[ \text{while} (D_{\text{ca}} \not\models R \equiv R_{\text{new}}) \{
\]
\[ R := R_{\text{new}}; R_{\text{new}} := \Psi(R) \}
\[ \}
\]

The fixpoint procedures test if \( R \equiv R_{\text{new}} \) is entailed given only the unique name and domain closure for actions axioms \( D_{\text{ca}} \). In general, there is no guarantee that such procedures will ever terminate i.e., that for some \( i \) \( D_{\text{ca}} \models R_i \equiv R_{i+1} \). But if the lfp procedure does terminate, then \( D_{GS} \models R_i[S] \equiv \mu Z. \Psi[Z][S] \) and \( R_i \) is first-order and uniform in \( S \) (and similarly \( gfp \)). In such cases, the task of verifying a fixpoint formula in the situation calculus is reduced to that of verifying a first-order formula. W e have the following result:

**Theorem 1.** [2] Let \( D_{GS} \) be a situation calculus game structure and let \( \Psi \) be an \( L \)-formula. If the algorithm above terminates, then \( D_{GS} \models \Psi[S_0] \) iff \( D_{S_0} \cup D_{\text{ca}} \models \tau(\Psi)[S_0] \).

### 3 Case Studies

#### 3.1 Light World (LW)

Our first example domain is the Light World (LW), a simple game we designed that involves an infinite row of lights, one for each integer. A light can be on or off. Each light has a switch that can be flipped, which will turn the light on (resp., off) if it was off (resp., on). There are 2 players, \( X \) and \( O \). Players take turns and initially it is \( X \)’s turn. The goal of player \( X \) is to have lights 1 and 2 on in which case player \( X \) wins the game. Initially, lights 1 and 2 are known to be off and light 5 is known to be on. Note that this is clearly an infinite state domain as the set of lights that can be turned on or off is infinite. Note also that the game may go on forever without the goal being reached (e.g., if player \( O \) keeps turning light 1 or 2 off whenever \( X \) turns them on).

We will show that the DLP method can be used to verify some interesting properties in this domain. We apply the method with one small modification: when checking whether the two successive approximates are equivalent, we use an axiomatization of the integers \( D_Z \) in addition to the unique names and domain closure axioms for actions \( D_{\text{ca}}^{LW} \), as our game domain involves one light for every integer.\(^2\) The game structure axiomatization for this domain is:

\[ D_{GS}^{LW} = \Sigma \cup D_{\text{poss}}^{LW} \cup D_{\text{ssa}}^{LW} \cup D_{\text{ca}}^{LW} \cup D_{S_0}^{LW} \cup D_{\text{Legal}}^{LW} \cup D_Z. \]

\(^2\) Our axioms and the properties we attempt to verify only use a very simple part of integer arithmetic. It should be possible to generate the proofs using the decidable theory of Presburger arithmetic [8] after encoding integers as pairs of natural numbers in the standard way [9]. Most theorem proving systems include sophisticated solvers for dealing with formulas involving integer constraints and it should be possible to use these to perform the reasoning about integers that we require.
We have only one action \( \text{flip}(p, t) \), meaning that player \( p \) flips light \( t \), with the precondition axiom (in \( D_{\text{pos}}^{LW} \)): \( \text{Poss}(\text{flip}(p, t), s) \equiv \text{Agent}(p) \). We have the fluents \( \text{On}(t, s) \), meaning that light \( t \) is on in situation \( s \), and \( \text{turn}(s) \), a function that denotes the agent whose turn it is in \( s \). The successor state axioms (in \( D_{\text{ss}}^{LW} \)) are as follows:

\[
\text{On}(t, \text{do}(a, s)) \equiv \exists p \ a = \text{flip}(p, t) \land \neg \text{On}(t, s) \lor \text{On}(t, s) \land \forall p. a \neq \text{flip}(p, t)
\]

\[
\text{turn}(\text{do}(a, s)) = p \equiv p = O \land \text{turn}(s) = X \lor p = X \land \text{turn}(s) = O
\]

The rules of the game are specified using the \( \text{Legal} \) predicate. We have the following axioms in \( D_{\text{legal}}^{LW} \):

\[
\text{Legal}(\text{do}(a, s)) \equiv \text{Legal}(s) \land \exists p, t. \text{Agent}(p) \land \text{turn}(s) = p \land a = \text{flip}(p, t)
\]

Control \( \equiv \exists a. \text{Legal}(\text{do}(a, s)) \land \text{Agent}(a) = p \land \text{Agent}(\text{flip}(p, t)) = p, \forall p. \{ \text{Agent}(p) \equiv (p = X \lor p = O) \}, X \neq O
\]

Thus legal moves involve the player whose turn it is flipping any switch. We have the following unique name and domain closure axioms for actions in \( D_{\text{ca}}^{LW} \):

\[\forall a. \{ \exists p, t. a = \text{flip}(p, t) \}\]

\[\forall p, p', t, t'. \{ \text{flip}(p, t) = \text{flip}(p', t') \equiv p = p' \land t = t' \} \]

Finally, the initial state axioms in \( D_{\text{io}}^{LW} \) are: \( \text{turn}(S_0) = X, \neg \text{On}(1, S_0), \neg \text{On}(2, S_0), \neg \text{On}(5, S_0), \) and \( \text{Legal}(S_0) \).

For our first verification example, we consider the property that it is possible for \( X \) to eventually win assuming \( O \) cooperates, which can be represented by the following formula:

\[\exists \Diamond \text{Wins}(X) \equiv \mu Z. \text{Wins}(X) \lor \exists \bigcirc Z,\]

where \( \text{Wins}(X, s) \equiv \text{Legal}(s) \land \text{On}(1, s) \land \text{On}(2, s) \). We apply the DLP method to this example. We can show that the regressed approximations simplify as follows (see [10] for a more detailed version of all proofs in this paper):

\[D_{\text{ca}}^{LW} \models \text{R}_0(s) \equiv \text{Wins}(X, s) \lor R(\exists \bigcirc \text{False}) \equiv \text{Legal}(s) \land \text{On}(1, s) \land \text{On}(2, s)\]

This approximation evaluates to true if \( s \) is such that \( X \) is winning in \( s \) already (in no steps), i.e., if light 1 and light 2 are on in \( s \).

\[D_{\text{ca}}^{LW} \cup D_Z \models \text{R}_1(s) \equiv \text{Wins}(X, s) \lor R(\exists \bigcirc \text{R}_0) \equiv \text{Legal}(s) \land \text{On}(1, s) \land \text{On}(2, s) \lor \text{Legal}(s) \land \text{turn}(s) = X \lor \text{turn}(s) = O \land \text{On}(1, s) \lor \text{Legal}(s) \land \text{turn}(s) = X \lor \text{turn}(s) = O \land \text{On}(2, s)\]

This approximation evaluates to true if \( s \) is such that \( X \) can win in at most 1 step; these are legal situations where player \( X \) is already winning or where one of lights 1 or 2 is on, as \( X \) or \( O \) can turn the other light on at the next step.

\[D_{\text{ca}}^{LW} \cup D_Z \models \text{R}_2(s) \equiv \text{Wins}(X, s) \lor R(\exists \bigcirc \text{R}_1) \equiv \text{Legal}(s) \land \text{On}(1, s) \land \text{On}(2, s) \lor \text{Legal}(s) \land \text{turn}(s) = X \lor \text{turn}(s) = O\]

240
This approximation evaluates to true if $s$ is such that $X$ can win in at most 2 steps; this is the case if $X$ is winning already or if $s$ is any legal situation where it is $X$ or $O$’s turn, as the controlling player can turn light 1 on at the next step and the other player can and light 2 on at the second step).

$D_{ca}^{LW} \cup D_{Z} \models R_3(s) \equiv \text{Wins}(X,s) \lor R(\exists \circ R_2) \equiv L_{\text{legal}}(s) \land \text{On}(1,s) \land \text{On}(2,s) \lor L_{\text{legal}}(s) \land (\text{turn}(s) = X \lor \text{turn}(s) = O)$

The fixpoint iteration procedure converges at the 4th step as we have: $D_{ca}^{LW} \cup D_{Z} \models R_2(s) \equiv R_3(s)$. Note that it can be shown using the entire theory (by induction on situations) that $D_{ca}^{LW} \models R_2(s) \equiv \text{Legal}(s)$, as it is always either $X$’s or $O$’s turn. Thus, it is possible for $X$ to eventually win in any legal situation. It then follows by Theorem 1 of DLP that: $D_{GS}^{LW} \models \exists \text{Wins}(X)[S_0]$ if $D_{GS}^{LW} \models \text{Legal}(S_0) \land \{\text{On}(1,S_0) \land \text{On}(2,S_0) \lor \text{turn}(S_0) = X \lor \text{turn}(S_0) = O\}$. By the initial state axioms, the latter holds so $D_{GS}^{LW} \models \exists \text{Wins}(X)[S_0]$, i.e., player $X$ can eventually win in the initial situation.

For our second example, we look at the property that $X$ can ensure that he/she eventually wins no matter what $O$ does, i.e., the existence of a strategy that ensures $\text{Wins}(X)$. This can be represented by the following formula:

$$\langle\langle\{X\}\rangle\rangle \text{Wins}(X) = \mu Z. \text{Wins}(X) \lor \langle\langle\{X\}\rangle\rangle \circ Z$$

We apply the DLP method to try to verify this property. We can show that the regressed approximations simplify as follows:

$D_{ca}^{LW} \cup D_{Z} \models R_0(s) \equiv \text{Wins}(X,s) \lor R(\langle\langle\{X\}\rangle\rangle \circ \text{false})$

$\equiv L_{\text{legal}}(s) \land \text{On}(1,s) \land \text{On}(2,s)$

This approximation evaluates to true if $s$ is such that $X$ is already winning in $s$ (in no steps); these are situations where lights 1 and 2 are already on.

$D_{ca}^{LW} \cup D_{Z} \models R_1(s) \equiv \text{Wins}(X,s) \lor R(\langle\langle\{X\}\rangle\rangle \circ R_0)$

$\equiv L_{\text{legal}}(s) \land \text{On}(1,s) \land \text{On}(2,s) \lor L_{\text{legal}}(s) \land \text{turn}(s) = X \lor \text{turn}(s) = \text{On}(1,s) \lor L_{\text{legal}}(s) \land \text{turn}(s) = X \land \text{On}(2,s)$

This approximation evaluates to true if $s$ is such that $X$ can ensure it wins in at most 1 step. This holds if lights 1 and 2 are already on or if either light 1 or 2 is on and it is $X$’s turn, as $X$ can then turn the other light on at the next step.

The next approximate $R_2$ simplifies to the same formula as $R_1$ and $D_{ca}^{LW} \cup D_{Z} \models R_1(s) \equiv R_2(s)$, so the fixpoint iteration procedure converges in the 3rd step. Therefore by Theorem 1 of DLP: $D_{GS}^{LW} \models \langle\langle\{X\}\rangle\rangle \text{Wins}(X)[S_0] \equiv R_1(S_0)$. Since both lights 1 and 2 are off initially, it follows by the initial state axioms that $D_{GS}^{LW} \models \neg \langle\langle\{X\}\rangle\rangle \text{Wins}(X)[S_0]$, i.e., there is no winning strategy for $X$ in $S_0$. However, we also have that $D_{GS}^{LW} \models \langle\langle\{X\}\rangle\rangle \text{Wins}(X)[S_1]$, where $S_1 = \text{do}(\text{flip}(O,3), \text{do}(\text{flip}(X,1), S_0))$, i.e., $X$ has a winning strategy in the situation $S_1$ where $X$ first turned light 1 on and then $O$ flipped light 3, as $X$ can turn on light 2 next.

Note that when the fixpoint approximation method is able to show that a coalition can ensure that a property holds eventually, the theory is complete, and we have domain closure, we can always extract a strategy that the coalition can follow to achieve the
property: a strategy works if it always selects actions for the coalition that get it from one approximate to a lower approximate \((R_i \to R_{i-1})\).

### 3.2 Oil Lamp World (OLW)

The DLP method tries to detect convergence by checking if the \(i\)-th approximate is equivalent to the \((i+1)\)-th approximate using only the unique name and domain closure axioms for actions \(D_{ca}\) (to which we have added the axiomatization of the integers). We now give an example where this method does not converge in a finite number of steps. However, we also show that if we use some additional facts that are entailed by the entire theory \(D_{GS}^{OLW}\), including the initial state axioms, when checking if successive approximates are equivalent, then we do get convergence in a finite number of steps.

Consider the Oil Lamp World (OLW), a variant of the Light World (LW) domain discussed earlier. It also involves an infinite row of lamps one for each integer, which can be on or off. A lamp has an igniter that can be flipped. When this happens, the lamp will turn it on if lamp \(_t+1\) is already on. There is only one agent, \(X\). The goal of \(X\) is to have lamp 1 on, in which case \(X\) wins. Observe that the game may go on indefinitely without the goal being reached, e.g., if \(X\) keeps flipping a lamp other than lamp 1 repeatedly.

The game structure axiomatization for this domain is: \(D_{GS}^{OLW} = \Sigma \cup D_{pos}^{OLW} \cup D_{ssu}^{OLW} \cup D_{ca}^{OLW} \cup D_{ssu}^{OLW} \cup D_{Legal}^{OLW} \cup D_{Z}\). As in the previous example, we have only one action, \(flip(p, t)\), meaning that \(p\) flips the igniter on light \(t\), with the following precondition axiom (in \(D_{pos}^{OLW}\)): \(Poss(flip(p, t), s) \equiv Agent(p)\). But there is no turn taking in this game as there is only one agent \(X\). We have the successor state axiom (in \(D_{ssu}^{OLW}\)): \(On(t, do(a, s)) \equiv \exists p \ a = flip(p, t) \land On(t + 1, s) \lor On(t, s)\). Note that once a lamp is turned on it remains on. The axioms in \(D_{Legal}^{OLW}\) specifying the rules of the game are similar to the ones given earlier for the Light World domain, and include: \(Legal(do(a, s)) \equiv Legal(s) \land \exists p, t. \ Agent(p) \land a = flip(p, t)\). Thus legal moves involve \(X\) flipping any igniter. The unique name and domain closure axioms for actions and the initial state axioms are exactly as in the Light World example.

We are interested in verifying the property that it is possible for \(X\) to eventually win \(\exists W ins(X)\), where \(W ins(X, s) \equiv Legal(s) \land On(1, s)\). We begin by applying the DLP method and try to show that successive approximates are equivalent using only the unique name and domain closure axioms for actions \(D_{ca}^{OLW}\) and the axiomatization of the integers \(D_{Z}\). We can show that the regressed approximations simplify as follows:

\[
D_{ca}^{OLW} \cup D_{Z} \models R_0(s) \equiv W ins(X, s) \lor R(\exists R_0) \equiv Legal(s) \land On(1, s)
\]

This approximation evaluates to true if \(s\) is such that \(X\) is already winning (in no steps); these are legal situations where lamp 1 is on.

\[
D_{ca}^{OLW} \cup D_{Z} \models R_1(s) \equiv W ins(X, s) \lor R(\exists R_0) \equiv Legal(s) \land (On(1, s) \lor On(2, s))
\]

This approximation evaluates to true if \(s\) is such that \(X\) can win in at most 1 step; these are legal situations where either lamp 1 is on or where lamp 2 is on, and then \(X\) can turn lamp 1 on at the next step.
$D^{OLW}_{ca} \cup D_Z \models R_2(s) \equiv \text{Wins}(X, s) \lor R(\exists \cap R_1) \equiv \text{Legal}(s) \land (\text{On}(1, s) \lor \text{On}(2, s) \lor \text{On}(3, s))$

This approximation evaluates to true if $s$ is such that $X$ can win in at most 2 steps; these are legal situations where either lamp 1 is on, or where lamp 2 is on (and then $X$ can turn lamp 1 on at the next step), or where lamp 3 is on (and then $X$ can turn on lamps 2 and 1 at the next steps).

We can generalize and show that for all natural numbers $i$, $D^{OLW}_{ca} \cup D_Z \models R_i \equiv \text{Legal}(s) \land \bigvee_{1 \leq j \leq i+1} \text{On}(j, s)$. That is, $X$ can win in at most $i$ steps if some lamp between 1 and $i + 1$ is on. It follows that for all $i$, $D^{OLW}_{ca} \cup D_Z \not\models R_i \equiv R_{i+1}$, since one can always construct a model of $D^{OLW}_{ca} \cup D_Z$ where every light except $i + 2$ is off. Thus, the plain DLP method fails to converge in a finite number of steps.

Nonetheless, there is a way to strengthen the DLP method to get convergence in a finite number of steps. The idea is to use some facts that are entailed by the entire theory in addition to the unique name and domain closure axioms for actions $D^{OLW}_{ca}$ and the integer axioms $D_Z$. First, we can show by induction on situations that any lamp that is on in the initial situation will remain on forever, i.e., $D^{OLW}_{GS} \models \phi_{op}$, where $\phi_{op} \equiv \forall k(\text{On}(k, S_0) \lor \forall s \text{On}(k, s))$. Then, it follows that for any natural numbers $i, j, i \leq j$, $D^{OLW}_{ca} \cup D_Z \cup \{\text{On}(i + 1, S_0), \phi_{op}\} \models R_j \equiv \text{Legal}(s)$. In essence, $X$ can eventually win in any legal situation where some lamp $n$ is known to be on. It follows that $D^{OLW}_{ca} \cup D_Z \cup \{\text{On}(i + 1, S_0), \phi_{op}\} \models R_i \equiv R_{i+1}$. Thus, the method converges in a finite number of steps if we use the facts that some lamp $n$ is known to be on initially and that a lamp that is on initially remains on forever. Moreover, our initial state axioms include $\text{On}(5, S_0)$. Thus, $D^{OLW}_{GS} \models \exists \phi \text{Wins}(X)[S_0]$, i.e., $X$ can eventually win in the initial situation, as it is legal and lamp 5 is on.

We can also show by induction on situations that if all lights are off initially, they will remain so forever: $D^{OLW}_{GS} - D^{OLW}_{S_0} \models (\forall k \neg \text{On}(k, S_0)) \lor (\forall s \forall k \neg \text{On}(k, s))$. Then, we can show by a similar argument as above that the fixpoint approximation method converges in a finite number of steps if we use the facts that all lamp are off initially and that if all lamps are off initially, they remain off forever.

### 3.3 In-Line Tic-Tac-Toe (TTT1D)

Our final example domain is more like a traditional game. It involves a one-dimensional version of the well-known Tic-Tac-Toe game that is played on an infinite vector of cells, one for each integer. We show that the DLP method does work to verify both the possibility to win and the existence of a winning strategy in this game, although in the former case the proof is long and tedious. There are two players, $X$ and $O$, that take turns, with $X$ playing first. All cells are initially blank, i.e., marked $B$. Players can only put their mark at the left or right edge of the already marked area. The functional fluent $\text{curn}$ denotes the marking position on the left (negative) side of the marked area and similarly $\text{curp}$ denotes the marking position on the right (positive) side of the marked area. Initially, $\text{curn}$ refers to cell 0 and $\text{curp}$ to cell 1. Player $p$ can put its mark in the cell on the left (negative) side of the marked area, i.e., the cell referred to by $\text{curn}$, by doing the action $\text{markn}(p)$. This also decreases the value $\text{curn}$ by 1 so that afterwards, it points to the next cell on the left. There is an analogous action $\text{markp}(p)$ for marking the the cell
The rules of the game are specified (in DLP method to this property (using only the unique name and domain closure on the right (positive) side of the marked area denoted by curp. A player wins if it succeeds in putting its mark in 3 consecutive cells. E.g., if initially we have the following sequence of moves: [markp(X), markn(O), markp(X), markn(O), markp(X)], then in the resulting situation the board is as follows:

$$\ldots, B_{-3}, B_{-2}, O_{-1}, O_0, X_1, X_2, X_3, B_4, B_5, \ldots$$

(with the subscript indicating the cell number) and X wins. Note that the game may go on indefinitely without either player winning, for instance if player \(O\) always mimics the last move of player \(X\).

The game structure axiomatization for this domain is: \(D_{GS}^{31D} = \Sigma \cup D_{poss}^{31D} \cup D_{ssa}^{31D} \cup D_{ca}^{31D} \cup D_{Legal}^{31D} \cup D_Z\). The precondition axioms (in \(D_{poss}^{31D}\)) state that the actions \(markn(p)\) and \(markp(p)\) are always possible if \(p\) is an agent. The successor state axioms (in \(D_{ssa}^{31D}\)) are as follows:

\[
curn(do(a,s)) = k \equiv \\
\exists p. \{a = markn(p)\} \land curn(s) = k + 1 \lor curn(s) = k \land \forall p. \{a \neq markn(p)\}
\]

\[
curlp(do(a,s)) = k \equiv \\
\exists p. \{a = markp(p)\} \land curlp(s) = k - 1 \lor curlp(s) = k \land \forall p. \{a \neq markp(p)\}
\]

\[
cell(k, do(a,s)) = p \equiv \\
a = markp(p) \land curlp(s) = k \lor a = markn(p) \land curn(s) = k \lor \\
\land \neg \exists p'. \{a = markn(p') \land curlp(s) = k\}
\]

\[
turn(do(a,s)) = p \equiv \text{agent}(a) = X \land p = O \land turn(s) = X
\]

\[
\land \text{agent}(a) = O \land p = X \land turn(s) = O
\]

The rules of the game are specified (in \(D_{Legal}^{31D}\)) as follows:

\[
Legal(do(a,s)) \equiv Legal(s) \land \\
\exists p. \{\text{turn}(s) = p \land \text{agent}(a) = p \land (a = markn(p) \lor a = markp(p))\}
\]

\[
Control(p, s) \equiv \exists a. Legal(do(a,s)) \land \text{agent}(a) = p \\
\land \forall p. \{Agent(p) \equiv (p = X \lor p = O), \ X \neq O\}
\]

The unique name and domain closure axioms for actions are specified in the usual way. Finally, we have the following initial state axioms in \(D_{S_0}^{31D}\): \(curn(S_0) = 0, curlp(S_0) = 1, turn(S_0) = X\), and \(Legal(S_0)\).

We first consider whether it is possible for \(X\) to eventually win \(\exists X \land Wins(X)\), where

\[
Wins(p, s) \equiv \exists k(\text{Legal}(s) \land \\
\{ (curn(s) = k - 2 \land cell(k - 1, s) = p \land cell(k, s) = p \land cell(k + 1, s) = p) \lor \\
(curlp(s) = k + 2 \land cell(k + 1, s) = p \land cell(k, s) = p \land cell(k - 1, s) = p))\}
\]

(Note that this simple definition allows both players to win.) If we apply the original DLP method to this property (using only the unique name and domain closure
axioms for actions $D^{T_{31}D}_{ca}$ and the axiomatization of the integers $D_Z$ to show that successive approximates are equivalent), the fixpoint approximation procedure does eventually converge, but only after 11 steps. The proof is very long and tedious and there are numerous cases to deal with. The reason for this is that we cannot use the fact that $\text{curn}$ is always less than $\text{curp}$ and that the cells that are between them are non-blank and that the other cells are blank, which are consequences of the initial state axioms. So our proof has to deal with numerous cases where there are non-blank cells to the left of $\text{curn}$ or to the right of $\text{curp}$ (if we can rule these cases out, the proof becomes much simpler). We omit the detailed proof (see [10]). But we have that:

$$\mathcal{D}_{ca}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{10}(s) \equiv \text{Wins}(X, s) \lor R(\exists \circ \ R_{0}) \equiv \text{Legal}(s)$$

Thus, it is possible for $X$ to win in at most 10 steps in all legal situations. Moreover we have that $\mathcal{D}_{ca}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{11}(s) \equiv R_{11}(s)$, and thus the fixpoint approximation procedure converges in the 11th step. There are situations where it does take at least 10 steps/moves for $X$ to win, for instance if we have $\text{curp} < \text{curn}$, i.e., $\uparrow_p < \uparrow_n$, with two blank cells in between, i.e., $\uparrow_p \text{BB} \uparrow_n$, and it is $O$’s turn. The fact that $\uparrow_p < \uparrow_n$ means that the initial marks that are made will later be overwritten. It is straightforward to check that it takes at least 10 moves for $X$ to have 3 X’s in a row and win ($O$ wins as well), for instance if $O$ keeps playing markn and $X$ keeps playing markp. It follows from our convergence result by Theorem 1 of DLP that: $\mathcal{D}_{GS}^{T_{31}D} \models \exists \text{Wins}(X)[S_0] \equiv R_{10}(S_0) \equiv \text{Legal}(S_0)$. Since we have $\text{Legal}(S_0)$ in the initial state axioms, it follows that $\mathcal{D}_{GS}^{T_{31}D} \models \exists \text{Wins}(X)[S_0]$, i.e., it is possible for $X$ to win in the initial situation.

Finally, we consider the property that $X$ can ensure that it eventually wins $\langle\langle\{\text{X}\}\rangle\rangle \odot \text{Wins}(X)$. We can apply the original DLP method to this property (using only the unique name and domain closure axioms for actions $D^{T_{31}D}_{ca}$ and the axiomatization of the integers $D_Z$ to show that successive approximates are equivalent). We can show that the regressed approximations simplify as follows:

$$\mathcal{D}_{ca}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{0}(s) \equiv \text{Wins}(X, s) \lor R(\exists \circ \ R_{0}) \equiv \text{Wins}(X, s)$$

$$\mathcal{D}_{ca}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{1}(s) \equiv \text{Wins}(X, s) \lor R(\exists \circ \ R_{0}) \equiv R_{0}(s) \lor X \text{CanPlayToWinNext}(s) \equiv R_{0}(s) \lor X \text{CanPlayToWinNext}(s)$$

This approximation evaluates to true if $s$ is such that $X$ can ensure to win in at most 1 step. These are legal situations where there are 3 X marks in a row on either side, or where it is $X$’s turn and there are 2 X marks already and $X$ can fill in the missing cell to get 3 in a row next (we omit details).

$$\mathcal{D}_{ca}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{2}(s) \equiv \text{Wins}(X, s) \lor R(\exists \circ \ R_{1}) \equiv R_{1}(s) \lor \text{Legal}(s) \land \text{turn}(s) = O \land \exists m. (\text{curn}(s) < m - 2 \land \text{cell}(m - 2, s) = X \land \text{cell}(m - 1, s) = X \land \text{curp}(s) = m) \land \exists n. (\text{curn}(s) = n - 1 \land \text{cell}(n, s) = X \land \text{cell}(n + 1, s) = X \land n + 1 < \text{curp}(s))$$

This approximation evaluates to true if $s$ is such that $X$ can ensure to win in at most 2 steps. These are legal situations where $X$ can ensure to win in at most 1 step as above, or where it is O’s turn and we have both $X_{k+1}X \uparrow_p$ with $\uparrow_n < k$ and $\uparrow_n \ X_{k+1}$ with $\uparrow_p > k$; then if $O$ plays markn then $X$ can play markp to win afterwards, and if $O$ plays markp then $X$ can play markn to win afterwards. The next approximation $R_{3}(s)$ simplifies to exactly the same formula as $R_{2}(s)$. Thus the procedure converges in the 4th step as we have: $\mathcal{D}_{GS}^{T_{31}D} \cup \mathcal{D}_{Z} \models R_{2}(s) \equiv R_{3}(s)$. Therefore by
Theorem 1 of DLP: $D_{GS}^{31D} \models \langle\{X\}\rangle \diamond Wins(X)[S_0] \equiv R_2(S_0)$. It follows by the initial state axioms that $D_{GS}^{31D} \models \neg\langle\{X\}\rangle \diamond Wins(X)[S_0]$ i.e., there is no winning strategy for $X$ in $S_0$. But $D_{GS}^{31D} \models \langle\{X\}\rangle \diamond Wins(X)[S_1]$, where $S_1 = do([markp(X), markn(O), markp(X), markn(O)], S_0)$, i.e., there is a winning strategy for $X$ in a situation where $X$ has marked twice on the right and $O$ has marked twice on the left. We have also developed two other examples of games played on an infinite infinite vector of cells to evaluate the DLP method; see [10] for details.

4 An Evaluation-Based Verification Tool

To further examine the feasibility of automating the DLP method, we have developed an evaluation-based Prolog implementation of a version of the method for complete initial state theories with the closed world assumption. The algorithm can correctly verify many properties in infinite state game structures. The method is completely automated, unlike most theorem proving-based approaches. One major limitation is that it does not actually check for convergence of the fixpoint approximation, and thus may not terminate when the property to verify is false, as we discuss later.

Our verifier builds on the logic programming evaluator for Situation Calculus projection queries developed by Reiter [5] for complete initial state theories with the closed world assumption. The approach uses a Prolog encoding of the domain’s basic action theory as defined in [5]. E.g., for the $T^{31D}$ domain, we have:

% Precondition Axioms
poss(markn(P),S) :- agent(P).
poss(markp(P),S) :- agent(P).

% Successor State Axioms
curn(K,do(A,S)) :- A=markn(_), curn(KX,S), K is KX - 1;
        not(A=markn(_)), curn(K,S).
curp(K,do(A,S)) :- A=markp(_), curp(KX,S), K is KX + 1;
        not(A=markp(_)), curp(K,S).
cell(K,M,do(A,S)) :- A=markp(M), curp(K,S); A=markn(M), curn(K,S);
        (not(A=markn(M)); not(curn(K,S))),(not(A=markp(M)); not(curp(K,S)));
        cell(K,M,S).
turn(P,do(A,S)) :- turn(x,S), P = o; turn(o,S), P = x.
legal(do(A,s0)) :- turn(x,S), P = o; turn(o,S), P = x.

% Initial State Axioms
cell(_,b,s0). % all cells are initially blank
curn(0,s0). curp(1,s0). turn(x,s0). legal(s0).

One can evaluate projection queries using such a program, e.g., check whether $cell(2,b,do(markp(x,s0)))$, i.e., that cell 2 is still blank after agent $X$ marks right in the initial situation. The program works essentially by regressing the query to the initial situation and evaluating it against the initial state axioms.

Reiter [5] shows how to define an evaluator for a rich set of first order queries on top of such an encoding of the basic action theory. Here is some of the evaluator code:

holds(P & Q,S) :-!, holds(P,S), holds(Q,S). % conjunction
holds(P v Q,S) :-!, (holds(P,S); holds(Q,S)). % disjunction
holds(some(V,P),S) :-!, subst(V,_,P,P1), holds(P1,S). %existential
% handled by replacing the variable by a fresh Prolog variable
holds(all(V,P),S) :-!, holds(-some(V,-P),S). % universal
...

% handling negation
holds(-P,S) :- 1l_atom(P), !, not(holds(P,S)).
holds(-(-P),S) :- !, holds(P,S).
holds(-(P & Q),S) :- !, holds(-P v -Q,S).
holds(-(P v Q),S) :- !, holds(-P & -Q,S).
...
holds(-all(V,P),S) :- !, holds(some(V,-P),S).
holds(-P,S) :- not(holds(P,S)).%

% handling atoms
holds(Pred,S) :- restoreSitArg(Pred,S,PredEx), !, PredEx.

The evaluator recursively evaluates the arguments of conjunctions and disjunctions. Existential quantification is left for Prolog to handle. Universal quantification is rewritten using negation and existential quantification. Negation is distributed over conjunction and disjunction. Finally, atomic fluents are evaluated using the Prolog encoding of the basic action theory.

In our verifier, we handle the key temporal operator $\langle\langle G \rangle\rangle ⃝ \Psi[S]$ essentially by translating it into its situation calculus definition and evaluating the resulting query. The algorithm is implemented in Prolog. Here “evaluation-based” refers to the use of evaluation instead of entailment to check state properties under the condition of complete information (i.e., single model) and the closed-world assumption. In general, the verifier checks if a given temporal property expressed in the $\mathcal{L}$-Logic holds for a given situation. The verifier is domain-independent.

In our verifier, temporal formulas such as $\langle\langle G \rangle\rangle ⃝ \Psi[S]$ are translated according to their definition into situation calculus projection queries and checked in the usual way using a combination of regression and evaluation:

holds(canEnsureNext(G,F),S) :- !, (incontrol(G,S), holds(exists_successor(G,F),S);
 incontrol(-G,S), holds(forall_successors2(-G,F),S)).
holds(exists_successor(G,F),S) :- !, member(P,G),
 agent_action(P, A), S1=do(A,S), legal(S1), holds(F,S1), !.
holds(forall_successors2(-G,F),S) :- !,
 not(holds(exists_successor2(-G,-F),S)).
holds(exists_successor2(-G,F),S) :- !, agent(P), not(member(P,G)),
 agent_action(P, A), S1=do(A,S), legal(S1), holds(F,S1), !.

The $\langle\langle G \rangle\rangle ⃝ \Psi[S]$ case is handled as $\neg\langle\langle G \rangle\rangle ⃝ \neg\Psi[S]$.

The $\mu$ operator is handled by generating successive fixpoint approximates $R_i$ as in the DLP method, except that we bound the number of approximates generated and we do not check for convergence, we simply check if the successive approximates hold in the situation of interest $S$:

mu_approx(Z,F,Int,N,S) :- binding_diameter(Max), N>Max, !,
write('binding diameter '), write(N),
write(' reached - stop'), nl, !, fail.
mu_approx(Z,F,Int,N,S) :- subst(Z,Int,F,Fx), holds(Fx,S), !,
output1(N,Fx).
mu_approx(Z,F,Int,N,S) :- M is N+1, subst(Z,Int,F,Int2), !,
mu_approx(Z,F,Int2,M,S).

By not checking for convergence, i.e. whether \( D \models R_{i+1} \equiv R_i \), we avoid the need for complex logical reasoning. The downside is that the verifier will never terminate on \( \mu Z. \Psi \) queries that are false even if the fixpoint approximation converges, as it does not detect this. To ensure termination, the user may impose a bound on the number of approximates that are generated and evaluated. The idea is similar to the binding diameter concept in bounded model checking [11]. In some cases, the bound can be a number of moves that is reasonable in the game modeled. The formula \( \langle\langle G\rangle\rangle \diamond \Psi \) is defined in terms of the \( \mu \) operator as \( \mu Z. \Psi \lor \langle\langle G\rangle\rangle \circ Z \). For this, our verifier generates fixpoint approximates and evaluates them in the given situation \( S \), stopping as soon as one of the approximates evaluates to true:

let \( R_0 := \Psi \lor \langle\langle G\rangle\rangle \circ \text{False} \) and evaluate \( R_0[S] \); if it succeeds, return success;
else let \( R_1 := \Psi \lor \langle\langle G\rangle\rangle \circ R_0 \) and evaluate \( R_1[S] \); if it succeeds, return success;
... else let \( R_{\text{limit}} := \Psi \lor \langle\langle G\rangle\rangle \circ R_{\text{limit}-1} \) and evaluate \( R_{\text{limit}}[S] \); if it succeeds, return success;
else return failure.

We have tested our verifier on some of our infinite state game structure examples. On the \( T^31D \) domain, the verifier can confirm that both agents can cooperate to ensure that \( X \) wins (in 5 steps) in the initial situation, i.e., the following query succeeds after generating and evaluating 6 approximates:

?- holds(canEnsureEventually([x,o],wins(x)),s0).

trying ##### approximation 1 ---> wins(x) v next([x, o], false)

[...]

trying ##### approximation 6 ---> wins(x) v
next([x, o], wins(x) v
next([x, o], wins(x) v
next([x, o], wins(x) v
next([x, o], wins(x) v
next([x, o], wins(x) v
next([x, o], false)))

[...]
> successor EXISTS for G --->
next([x, o], wins(x) v next([x, o], false)) ---> for
do(markn(o), do(markp(x), do(markn(o), do(markn(x), s0))))

[...]
> ##### approximation 6 holds --->

[...]
wins(x) v
next([x, o], wins(x) v
next([x, o], wins(x) v
As part of doing the verification, the system finds a sequence of actions by the 2 cooperating agents that allows \( X \) to win.

The verifier can also confirm that agent \( X \) can win (in 1 step) in the situation
\[
\text{do}(\text{markn}(o), \text{do}(\text{markp}(x), \text{do}(\text{markn}(o), \text{do}(\text{markp}(x), s0))))
\]
where \( X \) has already put 2 marks on the right and \( O \) had already put 2 marks on the left. However, if we try to check if \( X \) can ensure that it wins in the situation
\[
\text{do}(\text{markp}(x), \text{do}(\text{markn}(o), \text{do}(\text{markp}(x), s0)))
\]
where \( X \) has already put 2 marks on the right and \( O \) had already put 1 mark on the left, the verifier cannot confirm that the query is in fact false; it keeps generating successive approximates and eventually gives up after reaching the binding diameter. The problem is that \( O \) can always prevent \( X \) from winning at the next step and the verifier is not checking whether it has reached a fixpoint in the approximation.

We have also tested our verifier on the LW domain. This is a bit more challenging because there are infinitely many legal actions at every state, as any switch can be flipped. We represent legal actions so that agents try flipping switches in increasing order. For this domain, the verifier succeeds in confirming that the two agents can cooperate to ensure that \( X \) eventually wins by getting lights 1 and 2 on in \( S_0 \) (in 2 steps). It can also confirm that \( X \) can ensure that it wins in the situation where it has already flipped light 2 on and \( O \) has flipped light 4 on (as \( X \) can flip light 1 on at the next step). But the verifier cannot show that \( X \) cannot ensure that it eventually wins in the situation where it has already flipped light 2 on (as \( O \) can flip it off next and continue undoing any progress that \( X \) makes towards the goal). The verifier succeeds in showing that \( O \) can prevent \( X \) from winning at the next step (\( O \) can flip any switch except 1). It then generates approximate 3 and tries to show that \( X \) can win in one step after every action that \( O \) makes next. If we limit the set of actions that are considered (e.g., only flipping the first 10 switches) the verifier can confirm that \( X \) cannot win in 2 steps as \( O \) can flip light 2 off next. The verifier keeps generating and evaluating successive approximates and eventually gives up after reaching the binding diameter.

We have also tested our verifier on a formalization of the standard 2D Tic-Tac-Toe game (used as an example in DLP), a finite state domain. In this case the verifier can do a complete search and correctly answers queries about the existence of a winning strategy. For example it can confirm that \( X \) cannot ensure that it eventually wins in the initial situation with a blank board; it can also confirm that \( X \) can ensure that it eventually wins in a situation where \( X \) has marked the center square and \( O \) has then marked a non-corner square.

To summarize, in finite state domains the verifier correctly answers queries as it can do a complete search. In infinite state domains, our verifier can often show that least fixpoint queries are true but cannot show that least fixpoint queries are false (and greatest fixpoint queries are true), because it does not check whether successive approximates are equivalent. We hope to address this in future work.
In many cases, we would like to verify properties assuming that agents are following certain strategies, or have certain strategic preferences. E.g., in Tic-Tac-Toe, one might know that a player always tries to mark corners first. This would allow modeling more realistic types of agents. It can also cut down significantly on the number of alternative actions that must be considered and speed up verification. Knowing that the opponent follows certain strategic preferences may provide the player with a way to ensure it eventually wins when it could not otherwise.

We have extended the DLP logic to support this. There are many ways to model strategic preferences. A simple approach is to assume that the modeler defines a predicate $\text{Preferred}(p, a, s)$ that holds iff action $a$ is a preferred action for player $p$ in situation $s$. Note that there may be several alternative preferred actions in a situation. Other representations can be mapped to this form.

It is straightforward to modify the logic to only consider paths where all players select actions according to their preferences. We change the semantics of the $\langle\langle G \rangle\rangle$ operator as follows. If a player in $G$ is in control in the current situation, $\Psi$ must hold after some preferred action for him if there is one; if there is no preferred action, $\Psi$ must hold after some legal action. If a player not in $G$ is in control, $\Psi$ must hold after all preferred actions for him if there is some preferred action, and after all legal actions if there is none. This means that $\text{Preferred}(p, a, s)$ represents soft-constraints. If there are no preferred actions in a situation, we revert to considering all legal actions. Our implementation supports this type of specification of player action preferences and we have tested it on some Tic-Tac-Toe examples.

Our verifier also supports the use of the GameGolog language proposed in DLP to specify the game structure procedurally. See [10] for more details. The current prototype implemented in SWI Prolog with examples is available at www.cse.yorku.ca/~skmiec/SCGSverifier/. We believe that our verifier implementation is sound (assuming a "proper" Prolog interpreter is used, i.e., one that flounders on negative queries with free variables). It is not complete, in part for the same reasons that Prolog is not a complete reasoner for first order logic. We leave the proof of soundness for future work.

5 Discussion

In this paper, we described the results of some case studies to evaluate whether the DLP verification method actually works. We developed various infinite state game-type domains and applied the method to them. Our example domains are rather simple, but have features present in practical examples (e.g., the $T^31D$ domain is 1D version of Tic-Tac-Toe on an infinite board). Our experiments do confirm that the method does work on several non-trivial verification problems with infinite state space. We also identify some examples where the method, which only uses the simplest part of the domain theory, the unique names and domain closure for action axioms, fails to converge in a finite number of steps. We show that in some of these cases, extending the method to use some selected facts about the initial situation and some state constraints does allow us to get convergence in a finite number of steps. Our example domains and properties should be useful for evaluating other approaches to infinite state verification and synthesis.
We also described an evaluation-based Prolog implementation of a version of the DLP method for complete initial state theories with the closed world assumption. It generates successive approximates and checks if they hold in the situation of interest, but does not check if the sequence of approximates converges. Our verifier is fully automatic, unlike most theorem proving-based tools. We have also extended the framework to allow agents’ strategic preferences to be represented and used in verification. See [10] for more details about our verification experiments, proofs, and implemented verifier.

Among related work that deals with verification in infinite-states domains, let us mention [12, 13], which also uses methods based on fixpoint approximation. There, characteristic graphs are introduced to finitely represent the possible configurations that a Golog program representing a multi-agent interaction may visit. Their specification language is rich modal variant of the situation calculus with first and second order quantifiers, temporal operators and path quantifiers as in CTL*, and dynamic logic operators labeled with Golog programs. However, the language does not include fixpoint operators or alternating-time quantifiers, and is not a game structure logic. In their verification procedure, like DLP, they check for convergence using only the unique name axioms for actions part of the action theory. Also closely related is [14], which uses a fixpoint approximation method to compose a target process expressed as a ConGolog program out of a library of available ConGolog programs. Earlier, [15] proposed a fixpoint approximation method to verify a class of temporal properties in the situation calculus, called property persistence formulas. [16] show how a theorem proving tool can be used to verify properties of multi-agent systems specified in ConGolog and an extended situation calculus with mental states. A leading example of a symbolic model checker for multi-agent systems is MCMAS [17]. [18] show that model checking of an expressive temporal language on infinite state systems is decidable if the active domain in all states remains bounded. As well, [19] show that verification of temporal properties in bounded situation calculus theories where there is a bound on the number of fluent atoms that are true in any situation is decidable. [20] identifies an interesting class of Golog programs and action theories for which verification is decidable.

In future work, we would like to further develop our evaluation-based verifier. We plan to extend it to perform limited symbolic reasoning to detect if successive approximates are equivalent. We will also do more experimental evaluation. We would also like to implement an open-world symbolic version of the DLP method, perhaps by writing proof tactics in a theorem proving environment. It would also be desirable to develop techniques for identifying initial state properties and state constraints that can be used to show finite convergence in cases where these are needed. More generally, we need a better characterization of when this kind of method can be used successfully. Note that the DLP framework assumes that every agent has access to all the information specified in the theory. The framework should be generalized to deal with private knowledge and partial observability. Finally, the approach should be evaluated on real practical problems.

References


251
Side effects of agents are not just random

Bruno Mermet and Gaëlle Simon

Greyc – CNRS UMR 6072
Bruno.Mermet@univ-lehavre.fr

Abstract. Side effects are an important characteristic of MAS, and proving them is an interesting issue. They often can be expressed as liveness properties. But there is no system dedicated to this kind of proof. The GDT4MAS framework allows to specify and prove the correctness of multiagent systems. This framework is mainly dedicated to prove safety properties about the system and to prove that agents achieve their goal(s). However, there is no proof principle to prove that agents satisfy liveness properties that are not part of their goal(s). In this article, we propose a proof mechanism that addresses this kind of problem: we show how we can add to GDT4MAS a proof mechanism adapted to prove leads-to properties, a subclass of liveness properties.

1 Introduction

During the execution of a MAS, unexpected system properties are often observed. These properties can either be useful (they can be for example called emergent properties) or harmful. In both cases, it may be interesting to prove that such properties will eventually happen in order to understand how they happen. However, to our knowledge, there is no system suitable to prove such properties.

Proving the correctness of multiagent systems is a hard problem that has been tackled for several years. Most of the works on the subject have established that a new formal specification and proof system, dedicated to multiagent systems, should be developed. Among them, GDT4MAS [1] proposes interesting characteristics. Especially, the proof obligation generation process is fully automatisable and it can be applied to very large systems, essentially because it relies on theorem proving and first-order logic rather than on model-checking and propositional logic.

However, if this system is well-suited to prove invariant properties (also called safety properties) and to guarantee that agents satisfy their main goal, it does not propose any proof system to guarantee that an agent establishes liveness properties that are not part of its main goal, which is necessary when considering side effects.

So, in this article, we propose to add a new proof system to GDT4MAS dedicated to the verification of a well-known kind of liveness properties: leads-to properties.
Of course, there are techniques to verify leads-to properties in distributed systems [2–4], but these techniques are dedicated to systems where all the processes are globally taken into account for each proof that must be performed.

Most of these techniques are dedicated to systems where processes work on independant variables, and synchronize occasionnaly to exchange information (This for instance the case of the π-calculus [4]).

On the contrary, in a method such as TLA+ [3], shared variables can be specified, but the proof process requires to consider, for each step of the system trace, all the actions that may be considered. This is not suitable for multi-agents systems, because the number of possible actions is very large and thus, the property to prove would be to complicated to be performed by an automatic (or human) prover.

To perform efficient proofs on multi-agents systems, a compositional proof system is required. This is the case of the GDT4MAS proof system, and this is a property we require for a proof system dedicated to leads-to properties in multi-agents systems.

In the next section, we briefly introduce the notion of liveness property. In section 3, we recall the main concepts of the GDT4MAS framework. The new proof system we propose is described in section 4, and its application is examplified in section 5. Finally, a comparison with other works is proposed in section 6.

2 Invariant and liveness properties

When dealing with formal verification of software, many kinds of properties may be considered. In this section, we present two kinds of them: invariant properties and liveness properties.

2.1 Invariant properties

Invariant properties specify a set of states the system must satisfy at every moment. They are often presented as properties specifying that “nothing bad happens”. Theses properties are mainly safety properties. Indeed, when specifying safety critical systems (as, for instance, a train control system), a first critical step is to verify that the system does not reach an unsafe state. In a formal verification system such as the B method, used by corporates developing safety critical systems, this kind of property is the only one that is formally proven.

Using temporal logic, an invariant property IP is specified as:

$$\square(IP)$$

However, a system doing nothing may trivially verify such invariant properties. Indeed, these properties do not specify anything about the task of the system.
2.2 Liveness properties

Contrary to invariant properties, liveness properties specify how the system should modify its state. They are often presented as properties specifying that “something good will happen”. There are many kinds of liveness properties. Here are a small presentation of some of them. More details can be found for instance in [2, 3].

– **One-day**: a given property OD will eventually become true:

\[
\lhd (OD)
\]

– **Leads-to**: a given property LT will eventually become true every time another property P is true:

\[
\Box (P \rightarrow \lhd LT)
\]

– **Until**: a given property UP remains true until another property becomes true (and P will eventually become true):

\[
\Box (UP \rightarrow (\Box (UP \lor P) \land \lhd P))
\]

In the rest of this article, we will only consider leads-to properties. Indeed, a one-day property is a subtype of a leads-to property (with \(P \equiv \text{true}\) and \(LT \equiv OD\)), and an until property is also a special kind of leads-to property.

3 The GDT4MAS framework and the GDT model

3.1 Main concepts

In the GDT4MAS framework, the MAS is described by an environment, mainly specified by variables, an invariant (denoted \(iE\) in the sequel) and a population of agents evolving in this environment. Each agent is described as an instance of an agent type. As a consequence, in the following, after a short description of the notations we use, the notions of agent type and of agent behaviour are described.

3.2 Notation

**Notation 1 (primed and unprimed variable)**

When the value of a variable \(v\) in two execution states is considered, the value of \(v\) in the first state, called the current state, is written \(v\), and its value in the second state is written \(v'\). For instance, the action consisting in increasing the value of \(v\) by 1 is specified by the postcondition \(v' = v + 1\).
3.3 Agent Type Specification

**Simplified Definition 1 (Agent Type)** An agent type $t$ is mainly described by a name ($\text{name}_t$), a set of variables ($\text{Var}_t$), an invariant ($i_A$), and a behaviour ($b_t$) defined by a GDT.

**Definition 1 (Goal Decomposition Tree (GDT)).** A Goal Decomposition Tree describes the behaviour of the agents of a given type. Each node of this tree is a GDT goal. The tree structure is defined thanks to the decomposition of each GDT goal into subgoals using decomposition operators. A predicate called Triggering Context (TC) is associated to each GDT: an agent begins the execution of its behaviour every time its TC is true.

**Simplified Definition 2 (GDT goal)** A GDT goal $g$ is described by a name ($\text{name}_g$), a satisfaction condition ($\text{sc}_g$), a gpf ($gpf_g$), a decomposition or an action and an ns flag ($\text{ns}_g$). The satisfaction condition is a predicate specifying the property the goal must establish when it succeeds, whereas the gpf (Guaranteed Property in case of Failure) is a predicate specifying the property the goal must establish when its execution fails. The ns flag specifies whether the goal always succeeds (Necessarily Satisfiable or NS) or not (NNS).

Please notice that when the execution of a node fails, the invariant must still remain true. The failure of a node represents the fact that, in a real world, an agent is not always guaranteed to succeed in realizing a task dealing with the environment. For instance, a robot that must move its arm may be blocked by an object, or its arm may be rusty, reducing the amplitude of its move.

**Simplified Definition 3 (Action)** An action $\alpha$ is specified by a name ($\text{name}_\alpha$), a precondition ($\text{pre}_\alpha$), a postcondition ($\text{post}_\alpha$), an ns flag ($\text{ns}_\alpha$) and a gpf ($gpf_\alpha$). The precondition is a predicate specifying when the action is enabled, the postcondition specifies what that action does ($x' = x - 1$ for instance expresses that the action decreases the value of $x$ by 1), the ns flag has the value NS if the action is guaranteed to always succeed, and NNS if the action may fail. The gpf is a predicate specifying what is however guaranteed to be true if the action fails.

**Definition 2 (Goal decomposition).** A GDT goal is either a leaf goal or an intermediate goal. An action is attached to a leaf goal, whereas an intermediate goal is decomposed into several subgoals, thanks to a decomposition operator. A list of decomposition operators can be found in [5].

Among others, we can introduce the following decomposition operators:

- **SeqOr**: Sequential Or. It decomposes the parent goal $N$ into several subgoals $N_i$. Subgoals are executed from the left to the right. If the considered subgoal succeeds, $N$ is achieved and the execution of the decomposition ends. But if it fails, the next subgoal is considered. If the last subgoal is executed and fails, the satisfaction condition of $N$ must be evaluated to know if $N$ is however achieved or not.
- **SeqAnd**: Sequential And. It decomposes the parent goal $N$ into several subgoals $N_i$. Subgoals are executed from the left to the right. If the considered subgoal succeeds, the next one is executed. If the last subgoal is executed and succeeds, $N$ is achieved. But if a subgoal fails, the satisfaction condition of $N$ must be evaluated to determine whether $N$ is achieved or not.

- **SyncSeqOr** and **SyncSeqAnd**: These operators are similar to the SeqOr and SeqAnd operator, but a subset of environment variables can be locked during the whole execution of the parent goal decomposition.

An example of GDT is given in figure 1. In this figure, goals are described by their satisfaction condition. Moreover, $NS$ goals are surrounded by a double ellipse. In satisfaction conditions, $x$ and $x'$ respectively represent the value of the variable $x$ before and after the goal execution.

Fig. 1. Simple GDT

3.4 Proof system: general principles
The proof system for GDT4MAS relies on *Proof Schemas* (PS). Applying a proof schema generates *Proof Obligations* (PO), that may be proven by an automatic prover, such as PVS [6]. At the moment, PS allow us to prove several kinds of properties. We first prove invariants at the agent-type level and at the system-level. Moreover, the proof system of the method verifies that goal decompositions are valid. Most PS rely on goal contexts. These contexts are computed automatically starting from the root goal. Intuitively, the context $C_G$ of a goal $G$ is a predicate summarizing the state in which goal $G$ will be executed.

3.5 Notations
In this section, we present two notations of GDT4MAS that will be used in the sequel.

**Notation 2 ( Priming )** Let $f$ be a predicate/expression. If $f$ contains at least one primed variable, then $pr(f) = f$. Otherwise, $pr(f)$ is the predicate/expression derived from $f$ where each unsubscripted variable is primed.

Examples: $pr((x = x_0)) \equiv (x' = x_0)$ and $pr((x = x')) \equiv (x = x')$. 

257
Notation 3 (Invariant) Let $A$ an agent situated in an environment $E$. We write:

- $i_A$ the invariant regarding variables of the agent;
- $i_E$ the invariant associated to the environment variables;
- $i_{EA}$ the conjunction of $i_A$ and $i_E$.

4 A new proof mechanism dedicated to leads-to properties

This section presents the proof mechanism we propose to verify that some leads-to properties are established by an agent. Here, we only consider leads-to properties that are associated to an agent; no other agent is required to establish this property.

In this section, we consider a leads-to property $L$ defined so:

$$L \equiv \Box(P_L \rightarrow \Diamond Q_L)$$

A classical way to establish that a leads-to property is verified by a specification consists in associating a variant and a witness to this property [2].

Informally, a variant expresses the progress towards the establishment of $Q_L$. If it is proven that an agent makes a variant decrease and that when this variant reaches its lower bound, $Q_L$ is established, then the leads-to property $L$ is proven.

A witness is a property that represents the fact that $P_L$ has been true, and thus, that $Q_L$ must be established. In this article we propose to adapt this mechanism to verify leads-to properties of agents.

4.1 Definitions and notations

We begin by a formal definition of a variant:

**Definition 3 (Variant).** A variant is a decreasing sequence defined in a well-founded structure.

Of course, this definition requires to define what a well-founded structure is.

**Definition 4 (Well-founded Structure).** A well-founded structure $(S, <)$ is a set $S$ with an order relation $<$ such that every decreasing sequence in $S$ has a lower bound. For instance, $(\mathbb{N}, <)$ is a well-founded structure.

**Corollary 1.** A variant has a lower bound. We write $V_{0L}$ the lower bound of a variant $V_L$.

In this article, we will only consider variants defined on $(\mathbb{N}, <)$. This property must be added to the invariant of the agent.

**Definition 5 (Witness).** Let $L \equiv \Box(P_L \rightarrow \Diamond Q_L)$ a leads-to property. A witness is a property that must be true when $P_L$ is true, and remains true until $Q_L$ is true.
Notation 4 (Variant and Witness) Let $L$ a leads-to property associated to an agent $A$. We write:

- $V_L$ the variant we associate to $L$ to prove it;
- $V_{0L}$ the lower bound of the variant $V_L$;
- $W_L$ the witness we associate to $L$ to prove it.

4.2 Sketch of the proof process

Thanks to the variant and witness we associate to a leads-to property, proving that an agent establishes a leads-to property $L$ consists in proving that:

1. The chosen variant is a variant:
   - when it has reached its lower bound, $Q_L$ is established;
   - once $P_L$ has been true and until $Q_L$ becomes true, the agent must execute its gdt.
   - there is no other agent that increases the variant;
2. The chosen witness is a witness:
   - it is true when $P_L$ is true;
   - when $W_L$ is true, it remains true until $Q_L$ becomes true.
3. The agent progresses: when the agent executes its gdt, it makes the variant decrease or it establishes $Q_L$.

In the next parts of this section, we detail each of these steps.

4.3 The chosen variant is... a variant!

To prove that $V_L$ is a variant, we have to prove that, when it has reached its lower bound, the desired property is satisfied. So, we have to add the following proof obligation:

$$i_e A \land (V_L = V_{0L}) \rightarrow Q_L \tag{1}$$

Moreover, we also have to prove that once $P_L$ has been true, and until $Q_L$ becomes true, the agent is activated, and thus executes its GDT. This is established by proving the following property, where $TC_A$ is the triggering context of the agent:

$$i_e A \land W_L \land \neg Q_L \rightarrow TC_A \tag{2}$$

Finally, we also have to prove that no other agent makes the variant increase once $P_L$ has been established until $Q_L$ is established. So, for each other agent $A$ in the system, we have to check for every action $\alpha$ used in a leaf goal $G$ (we recall that $post$ and $gpf$ of actions contain primed variables):

$$i_e A \land C_G \land W_L \land (post_\alpha \lor gpf_\alpha) \rightarrow pr(V_L) \leq V_L \tag{3}$$

259
4.4 The witness property... is a witness!

As explained before, we associate to our leads-to property $L$ a witness property $W_L$ that verify both following properties:

- **Initialisation**: $W_L$ must be true when $P_L$ is true;
  The property that must be verified is the following:
  \[
  i_{\mathcal{E}_A} \land P_L \rightarrow W_L
  \]  

- **Finalization**: $W_L$ remains true until $Q_L$ becomes true.
  For each agent, we have to establish that, when it modifies the environment (that is to say, when it performs an action, whether it succeeds or not), if the witness is true before the action, then it is still true after the action has been performed, unless $Q_L$ has become true. So, for each action $\alpha$ associated to a leaf goal $G$ of each agent $A$, we have to verify:
  \[
  i_{\mathcal{E}_A} \land C_G \land W_L \land (\text{post}_\alpha \lor \text{gpf}_\alpha) \rightarrow pr(W_L \lor Q_L)
  \]  

4.5 The agent progresses

In order to prove that each execution of the GDT of an agent defines a progress towards the establishment of property $Q_L$, we have to prove that the execution of the main goal performs such a progress, that is to say, the main goal of the agent is a **progress goal**.

**Definition 6 (Progress goal (pg)).** We call Progress Goal a goal that either makes the variant decrease or establishes property $Q_L$. For a leads-to property $L$, we associate to each goal $G$ a boolean $pg_LG$ that is true if and only if $G$ is a progress goal.

Determining that a goal is a progress goal can be done by inference rules relying on the structure of the gdt, once we know which leaf goals make progress. Moreover, as the gdt execution depends on the success status of goals, we must determine, for each goal, if it is a **success progress goal** and if it is a **failure progress goal**.

**Definition 7 (Success Progress Goal (spg)).** We call Success Progress Goal a goal that either makes the variant decrease or establishes property $Q_L$ when it is executed and succeeds. For a leads-to property $L$, we associate to each goal $G$ a boolean $spg_LG$ that is true if and only if $G$ is a success progress goal.

**Definition 8 (Failure Progress Goal (fpg)).** We call Failure Progress Goal a goal that either makes the variant decrease or establishes property $Q_L$ when it is executed and fails. For a leads-to property $L$, we associate to each goal $G$ a boolean $fpg_LG$ that is true if and only if $G$ is a failure progress goal.

**Corollary 2.** A goal is a progress goal if and only if it is a success progress goal and a failure progress goal. So, for every goal $G$, we have $pg_LG = spg_LG \land fpg_LG$. 

260
In the following paragraphs, we first present how we determine spg and fpg leaf goals, and then, we show how we infer these properties for non-leaf goals. Finally, we give proof schemas that we have to associate to non-spg and non-fpg leaf goals.

**Determining the set of spg and fpg leaf goals** To determine if a goal is a spg goal, we have to check that when this goal succeeds (and so, establishes its satisfaction condition), it either makes the variant decrease or establishes $Q_L$. Of course, we must only consider executions of this goal performed when $P_L$ has been true, which is specified by the fact that $W_L$ is true. Hence the following property that must be established by each non lazy\(^1\) spg leaf goal $G$:

\[
(i \in A \land W_L \land C_G \land pr(sc_G)) \rightarrow ((pr(V_L) < V_L \lor pr(Q_L))
\] (6)

In the same way, a goal $G$ is a fpg leaf goal if and only if it verifies the following property:

\[
(i \in A \land W_L \land C_G \land pr(gpf_G)) \rightarrow ((pr(V_L) < V_L \lor pr(Q_L))
\] (7)

Please notice that, the gpf of an NS goal being *false*, such goals are fpg goals.

**Inference of spg and fpg properties** A first way to ensure that a non-leaf goal is spg or fpg consists in demonstrating that it is a consequence of the decomposition. In this article, we only detail this process for the SeqAnd/SyncSeqAnd and SeqOr/SyncSeqOr operators.

**SeqAnd and SyncSeqAnd** : Let $G$ a goal decomposed into $G_1$ SeqAnd $G_2$.

$G$ is a spg goal, if, in all the cases where $G$ may succeed, the variant decreases. Goal $G$ may succeed in three cases, detailed below:

- Of course, $G$ succeeds when $G_1$ then $G_2$ succeed. In this case, if either $G_1$ or $G_2$ are spg goals, goal $G$ makes the variant decrease.
- Because of side effects, $G$ may also succeed even if $G_1$ has failed. Then, $G_1$ must be fpg.
- Finally, $G$ may also succeed when $goal_1$ has succeeded, leading to the execution of $G_2$, which has failed. In this case, if $G_1$ is spg or $G_2$ is fpg, then the variant decreases.

So, we are guaranted that goal $G$ is spg if:

\[
\begin{cases}
spgL_{G_1} \land spgL_{G_2} \\
fpgL_{G_1} \\
spgL_{G_1} \land fpgL_{G_2}
\end{cases}
\]

\(^1\) In this article, we only focus on non lazy goals, that is to say goals that are always executed even if their satisfaction condition is already true when the goal is considered.
As a consequence, here is a sufficient condition to determine that a goal is a spg goal:

\[ fpGLG_1 \land (spgLG_1 \lor pgLG_2) \rightarrow spgL_G \] (8)

Now, to determine if \( G \) is a fpg goal, we consider both cases where it can fail, that is to say when its first subgoal fails or when its second subgoal fails after the first one has succeeded. Hence:

\[ fpGLG_1 \land (spgLG_1 \lor fpGLG_2) \rightarrow fpGL_G \] (9)

**SeqOr and SyncSeqOr**: Let \( G \) a goal decomposed into \( G_1 \text{ SeqOr } G_2 \).

Goal \( G \) may succeed in the three following cases:

- Goal \( G_1 \) succeeds;
- Goal \( G_1 \) fails, and then, goal \( G_2 \) succeeds.
- Goal \( G_1 \) fails, and then, goal \( G_2 \) fails but, because of side effects, goal \( G \) succeeds anyway.

So, we have:

\[ spgLG_1 \land (fpGLG_1 \lor pgLG_2) \rightarrow spgL_G \] (10)

The only case where Goal \( G \) may fail is when \( G_1 \) and \( G_2 \) fail. So, the fact that one of these goals is spg ensure that \( G \) is spg. Hence:

\[ fpGLG_1 \lor fpGLG_2 \rightarrow fpGL_G \] (11)

**Using satisfaction conditions to determine spg goals** Inference rules 8 and 10 to determine if a goal is spg give sufficient properties, but these properties are not always necessary. A typical example is when the satisfaction condition of a non-leaf goal directly establishes either \( Q_L \) or makes the variant decrease. So, for every non-leaf goal \( G \) that has not been characterized as a spg goal by inference rules described above, we will also verify if property 6 is true. If this is the case, goal \( G \) can be identified as an spg goal.

**Non-fpg and non-spg leaf goals** When a leaf goal \( G \) is not a spg goal, we however must prove that this goal does not make the variant increase when it succeeds. So, for each non-spg goal, we have to prove the following formula:

\[ i_{E_A} \land W_L \land C_G \land pr(sc_G) \rightarrow pr(V_L) \leq V_L \] (12)

In the same way, for each goal \( G \) that is not a fpg goal, we must prove:

\[ i_{E_A} \land W_L \land C_G \land gpf_G \rightarrow pr(V_L) \leq V_L \] (13)

Indeed, this is necessary to guarantee that between two steps during which the agent makes the variant decrease, it is not increased in another way.
5 Application on a small example

We choose here of course a very simple example, in order to be able to present all the principles of the proof. We consider a “multiagent system” with only one agent modifying the variant.

Please notice that the system may contain several other agents. In this case, as explained in section 4.3, it has to be proven that their actions do not increase the variant. Taking into account the dynamicity of the environment relies on the same principle, because, as explained in previous articles, the dynamicity of the environment can be modeled by an agent modifying the state of the environment.

The environment contains two variables, \( x \) and \( d \), and is specified by the following invariant:

\[
\begin{align*}
\forall \varepsilon \in \mathbb{E} & = \begin{cases}
x \in \mathbb{N} \\
d \in \mathbb{B} \\
d \leftrightarrow (x > 0 \land x \leq 10)
\end{cases}
\end{align*}
\]  

(14)

Our agent has a behaviour described by the GDT given in figure 2. In this figure, goals names (from \( A \) to \( E \)) and their simplified satisfaction conditions are given. By simplified SC, we mean that we did not write the part specifying that the value of other variables are not modified. For instance, the full SC of node \( D \) is \( y' = 2 \land x' = x \land d' = d \).

Informally, the goal of this agent is to decrease the value of the environment variable \( x \), by 2 if possible, and otherwise by 1.

Moreover, the triggering context of the agent, its invariant and the gpf of node \( E \) are defined so:

\[
\begin{align*}
TC_a & \triangleq d \\
I_a & \triangleq (y \in \mathbb{N}) \\
gpf_E & \triangleq x' = x \land d' = d \\
gpf_B & \triangleq x' = x
\end{align*}
\]  

(15) (16) (17) (18)

We want to prove that this agent establishes the following leads-to property:

\[
\Box (x = 10 \rightarrow \Box x = 0)
\]  

(19)

We will use \( x \) as the variant and \( d \) as the witness. To conform to the notation used in the previous section, we have:

\[
\begin{align*}
P_L & \triangleq (x = 10) \\
Q_L & \triangleq (x = 0) \\
V_L & \triangleq (x) \\
V_0 & \triangleq (0) \\
W_L & \triangleq (d)
\end{align*}
\]  

(20) (21) (22) (23) (24)
This article being focused on the proof of liveness properties, we do not present other proofs that must be performed to guarantee the correctness of this specification.

Moreover, in order to give readable formulae, we do not give full contexts of nodes and thus, hypotheses in theorems to prove are simplified.

5.1 Determining leaf progress goals

**goal D** As goal $D$ is a NS goal, it is a fpg goal.

To determine if it is spg, we must establish property 6 for this goal. Thus, we have:

$$\begin{align*}
W_L & \triangleq d \\
C_D & \triangleq d \\
pr(sc_D) & \triangleq (y' = 2)
\end{align*}$$

Of course, the conjunction of these properties with the invariant does not imply $x' < x$ or $x' = 0$. So, $D$ is not an spg goal. So:

$$\begin{align*}
spg_D & = false \quad (25) \\
fpg_D & = true \quad (26)
\end{align*}$$

**goal E** When goal $E$ is considered, we have:

$$\begin{align*}
W_L & \triangleq d \\
C_E & \triangleq \begin{cases} 
  d_{-2} \land y_{-1} = 2 \land x_{-1} = x_{-2} \\
  d_{-1} = d_{-2} \land y = y_{-1} \land x = x_{-1}
\end{cases} \\
pr(sc_E) & \triangleq x' = x - y \land (d' \leftrightarrow x' \neq 0)
\end{align*}$$
The context of goal $E$ given above is calculated by the context inference rules of the GDT4MAS method. It expresses the fact that goal $E$ is considered only after goal $D$ has succeeded when it has been executed in its context.

To establish that $E$ is a spg goal, according to 6, we must demonstrate that the conjunction of these properties imply that the variant decreases, that is to say $x' < x$. This is obvious because, from $C_E$, we can deduce $y = 2$ and from $pr(sc_E)$, we can deduce $x' = x - y$. So, $E$ is a spg goal.

We also have to determine if $E$ is a fpg goal, thanks to rule 7. Among the hypotheses of this rule, we have $gp_fE$ (which implies $x' = x$, see 18) and requires as conclusion either $x' < x$ (which cannot be true!) or $x' = 0$ which cannot be guaranteed because the context does not provide any knowledge about the value of $x$. So, goal $E$ is not a fpg goal.

So, we have:

$$spg_E = true$$ \hspace{1cm} (27)

$$fpg_E = false$$ \hspace{1cm} (28)

**goal $C$** About goal $C$, we have the following properties:

$$W_L \triangleq d$$

$$C_C \triangleq (d_{-2} \land x_{-1} = x_{-2} \land x = x_{-1})$$

$$pr(sc_C) \triangleq (x' = x - 1 \land (d' \leftrightarrow x' \neq 0))$$

To establish that goal $C$ is a spg goal, we must try to establish rule 6. This rule requires to prove, from the conjunction of the above properties, that the variant decreases ($x' < x$) or that property $Q_L$ is true. This is obvious because, from $sc_C$, we deduce that $x' = x - 1$, which implies $x' < x$. So, goal $C$ is a spg goal. Moreover, as this goal is a NS goal, this is also a fpg goal. So we have:

$$spg_C = true$$ \hspace{1cm} (29)

$$fpg_C = true$$ \hspace{1cm} (30)

**Conclusion** As a conclusion, we know that no leaf goal make the variant increase. Moreover, spg goals and fpg goals are respectively the following:

$$SPG = \{C, E\}$$ \hspace{1cm} (31)

$$FPG = \{C, D\}$$ \hspace{1cm} (32)

$$PG = \{C\}$$ \hspace{1cm} (33)
5.2 Inference of the progress property

**Goal B** To determine if goal $B$ is a spg goal, we apply rule 8 that provides the following sufficient condition to guarantee that goal $B$ is spg:

$$f_{pgD} \land (spgD \lor pgE)$$

However, $D$ is not a spg goal and $E$ is not a pg goal. Thus, with this rule, we cannot determine that goal $B$ is spg. So, we try to apply rule 6. Considering goal $B$, we have:

$$W_L \triangleq d$$
$$C_B \triangleq d$$
$$pr(sc_B) \triangleq x' = x - 2$$

And we have to establish that the conjunction of these formulae implies either $x' < x$ or $x' = 0$. As $sc_B$ implies $x' = x - 2$, we obviously have $x' < x$. So, $B$ is a spg goal.

We now have to determine if goal $B$ is a fpg goal, applying rule 9:

$$f_{pgD} \land (spgD \lor fpgE)$$

As goal $E$ is not a fpg goal and $D$ is not a spg goal, we can deduce that goal $B$ is not a fpg goal. So we have:

$$spg_B = true$$
$$fpg_B = false$$

**Goal A** To determine if goal $A$ is a spg goal, we apply rule 10, which gives:

$$spg_B \land (fpg_B \lor pgC) \rightarrow spg_A$$

As we have established before that goal $B$ is spg (34) and that goal $C$ is pg (33), we can establish that goal $A$ is a spg goal.

**Conclusion** Goal $A$ being a NS goal and a spg goal, we now know that each execution of the GDT of the agent makes the variant decrease.

5.3 The chosen variant is a variant

**Correctness** According to equation 1, to prove that the chosen variant is effectively a variant, we have to prove:

$$i_{\varepsilon_A} \land x = 0 \rightarrow x = 0$$

This is obviously true!
**Activation** According to equation 2, we have to prove:

\[ i \in \mathcal{A} \land d \land \neg(x = 0) \rightarrow (x = 10 \lor d) \]

Once again, this formula is obviously true.

### 5.4 the witness is a witness

**Initialisation** From formula 4, we have to verify:

\[ x = 10 \land (d \leftrightarrow (x > 0 \land x \leq 10)) \rightarrow d \]

This is still an obviously true formula.

**Finalization** We have to apply proof schema 5 for every leaf goal (and we recall here that, according to GDT4MAS principles, the gpf of an NS action is false).

**Goal D** The NS action \( \delta \) associated to goal \( D \) is defined by:

- \( \text{post} \delta \equiv y' = 2 \land d' = d \)
- \( \text{gpf} \delta \equiv \text{false} \)

So, with the context of goal \( D \) given above, we must establish:

\[ i \in \mathcal{A} \land d \land (d' = d \land y' = 2) \lor \text{false} \rightarrow \	ext{pr}(d \lor x = 0) \]

That can be simplified into:

\[ i \in \mathcal{A} \land d \land d' = d \land y' = 2 \rightarrow d' \lor x' = 0 \]

This property is obviously true (as \( d \) and \( d' = d \) can be found among the hypotheses).

**Goal E** The action \( \eta \) associated to goal \( E \) is defined by:

- \( \text{post} \eta \equiv x' = x - y \land (d' \leftrightarrow x' \neq 0) \)
- \( \text{gpf} \eta \equiv x' = x \land d' = d \)

Using \( C_E \) given above, applying proof schema 5, we obtain the following proof obligation:

\[ i \in \mathcal{A} \land d_{-2} \land y_{-1} = 2 \land x_{-1} = x_{-2} \\
\land d_{-1} = d_{-2} \land y = y_{-1} \land x = x_{-1} \land d \\
\land ((x' = x - y \land (d' \leftrightarrow x' \neq 0)) \lor (x' = x \land d' = d)) \\
\rightarrow \text{pr}(d \lor x = 0) \]
In order to simplify the explanation of the demonstration (that can be however easily performed by an automatic prover), we remove useless hypotheses. So, we have to prove:

\[(d \land x' = x - y \land (d' \leftrightarrow x' \neq 0)) \lor (d \land x' = x \land d' = d)\]

\[\rightarrow d' \lor x' = 0\]

The structure of this formula being \(a \lor b \rightarrow c\), we will successively demonstrate \(a \rightarrow c\) and \(b \rightarrow c\).

- \((d \land x' = x - y \land (d' \leftrightarrow x' \neq 0)) \rightarrow d' \lor x' = 0\)

We use a proof-by-case on the value of \(d'\). Either \(d'\) is true, and so, the goal is true, or \(d'\) is false. In the latter case, according hypothesis 2, \(x' = 0\), and so the goal is true. QED.

- \((d \land x' = x \land d' = d) \rightarrow d' \lor x' = 0\)

As \(d\) and \(d' = d\) are hypotheses, we obviously deduce \(d'\). QED.

So the proof obligation generated by applying proof schema 5 to goal \(E\) is true.

**Goal C** The action associated to goal \(C\) is defined by:

\[\begin{align*}
\text{post}_C & \triangleq x' = x - 1 \land (d' \leftrightarrow x' \neq 0) \\
\text{gpf}_C & \triangleq \text{false}
\end{align*}\]

Using \(C_C\) and applying proof schema 5 to goal \(C\), we have to prove:

\[
\begin{align*}
& (i_{E_A} \land d_{-2} \land x_{-1} = x_{-2} \land x = x_{-1} \land d) \\
& \quad \{ (x' = x - 1 \land (d' \leftrightarrow x' \neq 0)) \lor \text{false} \} \rightarrow \text{pr}(d' \lor x = 0)
\end{align*}
\]

In order to simplify the explanation of the demonstration (that can be however easily performed by an automatic prover), we remove useless hypotheses. So, we have to prove:

\[(d' \leftrightarrow x' \neq 0) \rightarrow (d' \lor x' = 0)\]

The proof is obvious: either \(d'\) is true, and so, the goal is true, or \(d'\) is false and so, from hypotheses, \(x' \neq 0\) is false, and so, \(x' = 0\). QED.

5.5 Conclusion

Following the proof system described in section 4, we have been able to establish that an agent whose behaviour is described by the gdt given in figure 2 satisfies a liveness property that is not a part of its main goal.
6 Comparison with other works

Several formal specification languages dedicated to multiagent systems exist. However, they are often not dedicated to the proof. This is for instance the case of 2apl [7], that is finally more a programming language than a specification language suited to proof. MetateM [8] gives the developer a way to specify properties, and the system controls that the execution does not violate these properties. However, this is a proof-by-construct process; this means that the proof is performed only at the execution time, and if the initial conditions change, a new proof (consisting in an execution of this new initial state) must be performed.

Finally, most works dealing with the verification of multiagent systems rely on model-checking principles. One of the most recent work in this area is the definition of AJPF [9], a model-checker relying on JPF [10] and the Agent Infrastructure Layer AIL. This is, as far as we know, the only system that proposes a way to verify leads-to properties on multi-agent systems. However, a first drawback of the method is the time taken by the system to establish the property (several hours for a very simple system). Of course, a more optimized model-checker such as spin [11], may greatly reduce the time required. However, such systems remain dedicated to small-size systems. Moreover, such systems have a more serious drawback: also they can be used to prove a property such as the property we have proven in section 5: $\square(x = 10 \rightarrow ◇x = 0)$, they cannot be applied when the left-hand side property (here, $x = 10$) characterize an infinite number of states. For instance, if we would be interested in proving the following leads-to property: $\square(x \geq 10 \rightarrow ◇x = 0)$, a model-checking-based method would fail, whereas the process we propose would be as efficient as it is in the given example.

The same problem can be found with MCMAS [12], which moreover does not provide a way to verify leads-to properties. This model-checking technique tries to verify formulae specified in propositional logic, as AJPF. The main disadvantage of this technique is that, relying on propositional logic, proofs cannot be generalized on systems of any size. For instance, in the cited article, it is shown that the verification of the dining cryptographers must be performed for each number of cryptographers we are interested in. Moreover, even if the time taken for 10 cryptographers is quite good, performances decrease dramatically when the number of cryptographers increase. Finally, with such a technique, to prove that the MAS work with any number of cryptographer, an infinite number of verifications must be performed, requiring, of course, an infinite time.

Indeed, as model checking techniques may be applied on systems with several millions of states, their complexity is a critical aspect that must be taken into consideration. But with theorem proving techniques, this criterion is quite less important. Indeed, each proof requires a very short time, and the number of proofs is very low, compared to the number of states generated in model checking techniques (for instance, even on a very large industrial system, less that 50,000 proofs had to be verified [13]). For instance, with the GDT4MAS model, if we call $n(t)$ the number of nodes of an agent type $t$ and $T$ the set of agent types, the number of proofs to perform is approximately $2\Sigma_{t\in T}n(t)$. 

269
7 Conclusion and Perspectives

In this article, we have shown that the GDT4MAS model, that was mainly dedicated to the proof of invariant properties, can be extended to prove liveness properties such as lead-to properties. As other proof obligations of the GDT4MAS framework, the new proof obligations generated are easily proven by an automatic theorem prover such as PVS.

This kind of proof can help in analyzing the behaviour of a MAS. In the work presented here, we have only considered liveness properties associated to a single agent. Of course, more general liveness properties at the system level will have to be considered, especially properties that are established not only by a single agent, but by a subset of the agents in the system. This is a short-term perspective. Moreover, at it is classically performed in standard verification systems, our proof system can only prove leads-to properties \( P \leadsto Q \) for which there is a continuous progress to \( Q \) once \( P \) has been true. In a multiagent system where agents are fully autonomous, we also have to consider properties for which this progress is not continuous. This is a long-term perspective for us.

References

10. NASA: Java Path Finder http://babelfish.arc.nasa.gov/trac/jpf.

270
Mutation Testing for Jason Agents

Zhan Huang, Rob Alexander, and John Clark

Department of Computer Science, University of York, York, United Kingdom
(zhan.huang,robert.alexander,john.clark)@cs.york.ac.uk

Abstract. Most multi-agent system (MAS) testing techniques lack empirical evidence of their effectiveness. Since finding tests that can reveal a large proportion of possible faults is a key goal in testing, we need techniques to assess the fault detection ability of test sets for MAS. Mutation testing offers a direct and powerful way to do this: it generates faulty versions of the program following mutation operators then checks if some test set can distinguish the original program from those versions. In this paper, we propose a set of mutation operators for the Jason agent-oriented programming language, and then introduce a mutation testing system for individual Jason agents that implements some of our proposed mutation operators. We use our implemented mutation operators to assess a test set for a small Jason system, and show that the test set that meets a combination of existing coverage criteria do not kill all mutants.

Keywords: Test Evaluation, Mutation Testing, Agent-Oriented Programming, Jason

1 Introduction

Multi-agent systems (MAS) are getting increasing attention in academics and industry as an emerging paradigm for engineering autonomous and distributed systems. In MAS engineering, testing is a challenging activity because of the increased complexity, large amount of data, irreproducibility, non-determinism and other characteristics involved in MAS [9]. Although many techniques have been proposed to address the difficulties in MAS testing, most of them lack empirical evidence of their effectiveness [10].

Effective testing requires tests that are capable of revealing a high proportion of faults in the system under test (SUT). It can be difficult to find real faulty projects to verify the real fault detection ability of the tests, however, test coverage criteria or simulation of real faults can be used to evaluate it.

For coverage based test evaluation, the executions of the tests are measured against some coverage criteria based on some model of the SUT; if these executions traverse all model elements defined in the coverage criteria, the tests are said to be adequate for the coverage criteria – in other words, they examine the involved model elements thoroughly. Existing coverage criteria for MAS testing include Low et al.’s plan and node based coverage criteria for BDI agents [1], Zhang et al.’s plan and event based
coverage criteria for Prometheus agents [2], and Miller et al.’s protocol and plan based coverage criteria for agent interaction testing [3].

Simulation of real faults offers a more direct way to assess the fault detection ability of the tests than test coverage criteria: faults can be hand-seeded or seeded by mutation [12], which is a systematic and automatic way of generating modified versions of the SUT (“mutants”) following some rules (“mutation operators”). After seeding faults, each test is executed against first the original SUT then each faulty version of the SUT. For each faulty version, if its behaviour differs from the original SUT in at least one test, it will be marked as “killed” to indicate that the faults in it can be detected by the tests. Therefore, the fault detection ability of the tests can be assessed by the “kill rate” – the ratio of the killed faulty versions to all generated faulty versions: higher the ratio is, more effective the tests are.

Mutation is more commonly used to seed faults than the hand-seeded way because it has solid theoretical foundation, and empirical studies suggest that it provides an efficient way to seed faults that are more representative of real faults than hand-seeded ones [13]. However, the mutation operators used to guide mutant generation may lead to a huge number of mutants so that comparing the behaviour of each mutant with that of the original SUT in each test is computationally costly. Another problem is that mutation unpredictably produces equivalent mutants – alternate implementations of the SUT which are not actually faulty, and thus which must be excluded from test evaluation. Although the process of detecting equivalent mutants may be partially automated, manual work is still required.

This process of using mutation to assess tests is called mutation testing. The key to success is to design an effective set of mutation operators that can simulate an adequate set of realistic faults in a reasonable (computationally tractable) number of generated mutants.

There is some preliminary work on mutation testing for MAS. Nguyen et al. [4] use standard mutation operators for Java to assess tests for JADE agents (which are implemented in Java). In contrast to standard operators for existing languages, it is likely that MAS-model-specific mutation operators will better simulate MAS-specific faults. In this vein, Adra and McMinn [5] propose a set of mutation operator classes for agent-based models. Saifan and Wahsheh [6] propose and classify a set of mutation operators for JADE mobile agents. Similarly, Savarimuthu and Winikoff [7, 8] systematically derive a set of mutation operators for the AgentSpeak BDI agent language and another set for the GOAL agent language. None of the above papers on MAS-specific mutation operators, however, actually implement and evaluate their operators except [8].

We aim to explore the use of mutation testing for MAS because mutation testing is widely thought to be a more rigorous test evaluation technique than coverage-based approaches [11], with the intention that our work can be used to assess and enhance the tests derived from the existing test generation/evaluation techniques (that are based on some coverage/mutation criteria) for MAS. This paper presents our preliminary work. In Section 2 we propose a set of mutation operators for Jason [14], which is a practical implementation of the AgentSpeak language; in Section 3 we introduce our mutation testing system for individual Jason agents that implements some of our
proposed mutation operators; in Section 4 we show the use of our implemented mutation operators in assessing and enhancing the tests for a Jason project; we end with a summary of our work, a discussion of the relationships to previous related work and some suggestions for where this work could go in the future.

2 Mutation Operators for Jason

Mutation operators are rules to guide mutant generation. For instance, a mutation operator for procedural programs called Relational Operator Replacement (ROR) requires that each occurrence of one of the relational operators \(<, \leq, >, \geq, =, \neq\) is replaced by each of the other operators [11]. A mutant usually only contains a simple, unary fault (e.g., in the above example, each generated mutant only replaces a single relational operator by another), because of the two underlying theories [12] in mutation testing: the Competent Programmer Hypothesis states that programmers create programs that are close to being correct; the Coupling Effect states that tests that can detect a set of simple faults can also find complex faults.

Since mutation is typically performed at source code level, a set of mutation operators is specific to a given programming language (C, Java, etc.). To design mutation operators for a programming language, it is common to start by proposing a large set based on the syntax and features of the language, and then to refine an effective set through evaluation.

Savarimuthu and Winikoff [7] applied the guidewords of HAZOP (Hazard and Operability Study) into the syntax of AgentSpeak, to systematically derive a set of mutation operators for AgentSpeak. Now we build on their work: we propose mutation operators for an implementation of AgentSpeak called Jason [14], which implements AgentSpeak’s operational semantics and extends AgentSpeak with various features useful for practical agent implementation. In contrast to their systematic method that may produce a large amount of mutation operators, we have used our judgment and borrowed the ideas of existing mutation operators to refine our operator set so as to preferentially implement and evaluate it, in the hope of avoiding implementing some ineffective operators. It can be seen that our mutation operators contain some shared with Savarimuthu and Winikoff for the core AgentSpeak language and others for the Jason specific features.

We base our work on Jason’s Extended Backus–Naur Form (EBNF), where a list of production rules is defined that describe Jason’s grammar. The EBNF we use is a simplified version that excludes some advanced features of Jason such as the use of directives and allowing conditional/loop statements in the plan body. These could, of course, be considered in further work. We divide these production rules into high-level and low-level ones – the high-level production rules specify the main syntactical concepts that are closely related to how Jason agents generally work, while the low-level ones specify the logical representations forming the Jason syntactical concepts. Accordingly our mutation operators for Jason can also be described as high- or low-level. In the following two subsections we present these mutation operators according to which production rules they are derived from.
2.1 High-Level Mutation Operators for Jason

Fig. 1 shows the high-level production rules in Jason’s EBNF; from this, we have derived 13 high-level mutation operators.

<table>
<thead>
<tr>
<th>Production rule number</th>
<th>EBNF rule</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>agent -&gt; (belief) * (init_goal) * (plan) *</td>
<td>Production rule 1 states that an agent is specified in terms of beliefs, initial goals and plans.</td>
</tr>
<tr>
<td>2</td>
<td>belief -&gt; literal [“<em>: log_expr “:</em>]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>init_goal -&gt; “* literal “:*</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>plan -&gt; [label] triggering_event [“<em>: context “:</em> context “:* body “:*]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>label -&gt; “* atomic_formula</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>triggering_event -&gt; (“*”</td>
<td>“.”</td>
</tr>
<tr>
<td>7</td>
<td>context -&gt; log_expr</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>body -&gt; body_formula [“:* body_formula “:*]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>body_formula -&gt; (“*”</td>
<td>“*”</td>
</tr>
<tr>
<td>10</td>
<td>action_for_comm -&gt; “.” (</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“send(“receiver “:<em>,” illocutionary_force “:</em>,” message_content</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[“:<em>,” reply “:</em>,” timeout “:*”)”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“broadcast(“illocutionary_force “:<em>,” message_content “:</em>)”</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>receiver -&gt; agent_id [“:* agent_id “:* agent_id “:*]</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>illocutionary_force -&gt; tell</td>
<td>untell</td>
</tr>
<tr>
<td>13</td>
<td>message_content -&gt; propositional_content</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“[“ propositional_content “:<em>,” propositional_content “:</em>”]”</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>propositional_content -&gt; belief</td>
<td>triggering_event</td>
</tr>
</tbody>
</table>

Fig. 1. High-level production rules in Jason’s EBNF (Rule 1–9 are adapted from [14], 10–14 are we add for specifying Jason agent communication)

Production rule 1 states that an agent is specified in terms of beliefs, initial goals and plans. From this rule we derive the following three mutation operators:

- **Belief Deletion (BD):** A single belief in the agent is deleted.
- **Initial Goal Deletion (IGD):** A single initial goal in the agent is deleted.
- **Plan Deletion (PD):** A single plan in the agent is deleted.

Production rule 2 states that a belief can be a literal representing some fact, or a rule representing some fact will be derived if some conditions get satisfied. The introduction of rules enables Jason to perform theoretical reasoning [16]. From this production rule we derive the following mutation operator:

- **Rule Condition Deletion (RCD):** The condition part of a rule is deleted.

A rule that RCD is applied to will only have its conclusion part – a literal – left, as a belief held by the agent regardless of whether the (now deleted) conditions get satisfied.

Production rule 6 states that the triggering event of a plan consists of a literal following one of the six types: belief addition (+), belief deletion (-), achievement goal
addition (+!), achievement goal deletion (-!), test goal addition (+?) and test goal deletion (-?). It can be seen that an event that can be handled by Jason plans represents a change – addition or deletion (represented using + or – operator respectively) – to the agent’s beliefs or goals. From this rule we derive the following mutation operator:

- **Triggering Event Operator Replacement (TEOR):** The triggering event operator (+ or -) of a plan is replaced by the other operator.

Production rule 7 states that the context of a plan can be a logical expression or set true (the latter is equivalent to the context not being specified at all). The plan context defines the condition under which the plan that has been triggered becomes a candidate for commitment to execution. From this production rule we derive the following mutation operator:

- **Plan Context Deletion (PCD):** The context of a plan is deleted if it is non-empty or not set true.

Production rule 8 states that the body of a plan can be a sequence of formulae, each of which will be executed in order, or set true (the latter is equivalent to the body not being specified at all). From this rule we derive the following three mutation operators:

- **Plan Body Deletion (PBD):** The body of a plan is deleted if it is non-empty or not set true.
- **Formula Deletion (FD):** A single formula in the body of a non-empty plan is deleted.
- **Formulae Order Swap (FOS):** The order of any two adjacent formulae in the body of a plan that contains more than one formula is swapped.

In many cases, PBD is equivalent to PD (Plan Deletion). However, since the plan context can contain internal actions that may cause changes in the agent’s internal state, the plan that PBD is applied to may still have an effect on the agent although its body has been deleted, in this case PBD is not equivalent to PD.

Production rule 9 states that a body formula can be one of the six types: achievement goal (!literal or !!literal), test goal (?literal), mental note (+ literal, - literal, - + literal), action (atomic_formula), internal action (atomic_formula) and relational expression. The former three types are involved in generating internal events that correspond to changes in achievement goals, test goals and beliefs respectively. Similar to how we derived Triggering Event Operator Replacement (TEOR) operator, from this production rule we derive the following mutation operator:

- **Formula Operator Replacement (FOR):** The operator of an achievement goal formula (! or !!) is replaced by the other operator, so is that of a mental note formula (+, -, -+).

It is worth noting that the achievement goal formula has two types: “!” is used to post a goal that must be achieved before the rest of the plan body can continue execution, “!!” allows the plan containing the goal to run alongside the plan for achieving the
goal. In the latter case, the two plans can compete for execution due to the normal intention selection mechanism.

Production rules 10–14 (marked with asterisks) are the ones we added for specifying Jason agent communication. It can be seen that two internal actions: .send and .broadcast, are used by Jason agents to send messages. The main parameters in these actions include the message receiver(s) (only used in .send action) that can be a single or a list of agents identified by the agent ID(s), the illocutionary force (tell, untell, achieve, etc.) representing the intention of sending the message and the message content that can be one or a list of propositional contents. From these production rules we derive the following three mutation operators:

1. **Message Receiver Replacement (MRR):** The receiver or the list of receivers in a .send action is replaced by another agent ID (or some subset of all the agent IDs in the MAS). If the action is .broadcast, it will be first converted to its equivalent .send action and then applied this mutation operator.

2. **Illocutionary Force Replacement (IFR):** The illocutionary force in an action for sending messages is replaced by another illocutionary force.

3. **Propositional Content Deletion (PCD2):** A single propositional content in the message content is deleted.

It is worth noting that a propositional content is some component of another type (e.g., belief, plan, etc.). Therefore, the mutation operators for these components can also be applied for mutating agent communication.

### 2.2 Low-Level Mutation Operators for Jason

Fig. 2 shows the low-level production rules in Jason’s EBNF; from this, we have derived 11 low-level mutation operators, most of which are borrowed from the existing ones for conventional programs.

```
1: literal -> ['"' ] atomic_formula
2: atomic_formula -> (ATOM | <VAR>) ["(" term ("", term ")") ] ["(" term ("", term ")") ]
3: term -> literal | list | arithm_expr | <VAR> | <STRING>
4: log_expr -> simple_log_expr | "not" log_expr | log_expr "&" log_expr | log_expr "|" log_expr | log_expr "(" log_expr ")"
5: simple_log_expr -> ( literal | rel_expr | <VAR> )
6: rel_expr -> rel_term [ ("<" | "<=" | ">" | ">=" | "=" | ">") "=" | ">") ]
7: rel_term -> literal | arithm_expr
8: arithm_expr -> arithm_term [ ("*" | "+" | "+" | "+" | "/" | "mod" | "mod") arithm_term ]
9: arithm_term -> <NUMBER> | <VAR> | "." arithm_term | (" arithm_expr ")
```

Fig. 2. Low-level production rules in Jason’s EBNF (Source: [14])

Production rule 1 states that a literal is an atomic formula or its strong negation (~l). Strong negation is introduced to overcome the limitation of default negation in logic.
programming: an agent can explicitly express that something is false by using strong negation, or express that it cannot conclude whether something is true or false using default negation (i.e. by the simple absence of a belief on the matter). From this production rule we derive the following mutation operator:

- **Strong Negation Insertion/Deletion (SNID):** The form of a literal (affirmative or strong negative) is transformed to the other form.

Production rule 2 and 3 state that an atomic formula consists of a relation followed by a list of annotations. Annotations can be used to provide further information about the relation, e.g., source is an important annotation that is appended to some atomic formulae automatically by Jason is used to represent where the atomic formulae (or its represented component) come from by taking one of the three parameters: percept, self or an agent ID. For instance, belief likes(rob, apples)[source(tom)] implies the information that rob likes apples comes from agent tom. From these production rules we derive the following two mutation operators:

- **Annotation Deletion (AD):** A single annotation of an atomic formula is deleted, if one exists.
- **Source Replacement (SR):** The source of an atomic formula is replaced by another source, if it exists.

Production rule 4 and 5 define logical expressions. From these rules we derive the following three mutation operators:

- **Logical Operator Replacement (LOR):** A single logical operator (& or |) is replaced by the other operator.
- **Negation Operator Insertion (NOI):** The negation operator ("not") is inserted before a (sub) logical expression.
- **Logical Expression Deletion (LED):** A single sub logical expression is deleted.

Production rule 6 and 7 define relational expressions. From these rules we derive the following two mutation operators:


278
We have developed a mutation testing system for individual Jason agents called mu-Jason\(^1\), where we have implemented the 13 high-level mutation operators via Jason APIs and Java reflection, both of which can be used to access and modify the architectural components of the agents and the state of the MAS at runtime. The class diagram and the user interface of muJason are shown in Fig. 3 and Fig. 4 respectively. Next we will introduce muJason from the perspective of the users.

\(^1\) [http://mujason.wordpress.com](http://mujason.wordpress.com)
A user can launch muJason by running the MutationSystem class and passing the name of the Jason project configuration file (postfixed with “.mas2j”) as the parameter. Then muJason will load the Jason project and display the mutation testing control panel (as shown in Fig. 4), where the user can configure, start and observe a mutation testing process.

Before initiating a mutation testing process, the user needs to specify the tests that need evaluation, the killing mutant criterion for each test and the TTL (Time to Live) of the original agent and each mutant for each test in the deploy(testID), isMutantKilled(testID) and getAgentTTL(testID) methods provided by the TestBed class (as shown in Fig. 3), respectively. Each of these methods is described as follows:

- **deploy(testID)**: this method sets up the initial configuration of the Jason system prior to each test run. The method is called each time by taking an ID identifying one of the tests, and the user can write code to set up the tests corresponding to the passed test IDs.
- **isMutantKilled(testID)**: this method is used to determine whether a mutant under some test is killed. It is called after each mutant terminates, and is passed the ID of the current test. Therefore, in this method the user can write code to check whether the mutant has been killed by each individual test. An alternate approach would have been to compare all the behaviour of the original agent and that of the mutant, but that would have been computationally expensive and prone to declaring mutants “killed” when the behaviour variation was of no consequence. With the approach taken here, the user can just specify the important aspects that need observation and comparison, via Jason APIs or Java reflection that can access the state of the MAS, or other techniques.
- **getAgentTTL(testID)**: this method is used to specify the lifetime of the original agent and its mutants as the return value for each test. Since agents usually run indefinitely, the original agent or each mutant can only be allowed to run for a certain period of time so that the next one can run. The whole Jason project will restart as soon as the original or mutated agent terminates, so that next time the agent can be observed from (and mutated at) the (same) initial point of the MAS. The lifetime or TTL of an agent is measured by the number of the reasoning cycles the agent can perform; it must be enough for the agent to expose all the behaviour involved in the process of killing mutants. The TTL for a test is actually part of the killing mutant criterion for that test. Although there may be ways to automatically evaluate the TTL or to automatically terminate the mutant once it is observed being killed, for simplicity in the beginning, the TTL for each test is fixed and manually set depending on the user’s experience.

After specifying the tests, the killing mutant criteria and the TTL, the user can configure and start a mutation testing process in the mutation testing control panel through the following steps (as shown in Fig. 4):

1. Select an agent and its mutation domain. Since muJason aims at individual agents, the user needs to select one from the MAS, and then the user can choose which belief(s), initial goals(s) and plan(s) of the selected agent the mutation operators will
be applied into. The user can ignore the agents/components unnecessary for testing, e.g., the GUI agent and the plans pre-defined by Jason for enabling agent communication, etc.

2. Select the mutation operators. After specifying the mutation domain of an agent, the user can select the mutation operators that will be applied into the mutation domain.

3. Start the mutation testing process. After configuration, the user can start the mutation testing, observe its process in the mutation testing control panel and wait for its result. The mutation testing process can be described using the following pseudo-code:

```plaintext
1: For each defined testID:
2:   Set up the test identified by the testID
3:   Get the specified TTL for the test
4:   Run the original Jason project for the TTL
5:   Restart the Jason project
6:   Create a mutant generator taking the selected agent, mutation domain and mutation operators
7:   While the generator can generate another mutant:
8:      Generate the next mutant
9:      Run the modified Jason project for the TTL
10:     Check if the mutant is killed under the current test, if so mark it “killed”
11:     Restart the Jason project
```

4 Evaluation

To perform a preliminary evaluation of our implemented mutation operators, we use them to guide the generation of the mutants for an agent in a Jason project, then examine whether a test set designed using some existing agent test coverage criteria can kill all the non-equivalent mutants. If it cannot kill all those mutants, that means this test set cannot reveal the faults simulated by the non-killed non-equivalent mutants, thereby demonstrating that our operators are effective in finding the weaknesses of this test set.

The Jason project we choose is available on the Jason website\(^2\), and is called Cleaning Robots. It involves a cleaner agent, an incinerator agent and several pieces of garbage located in a gridded area as shown in Fig. 5 (R1 represents the cleaner agent, R2 represents the incinerator agent, G represents the garbage). When this project is launched, the cleaner agent will move along a fixed path that covers all grid squares (move from the leftmost square to the rightmost one in the first row, then “jump” to the leftmost square in the second row and move to the rightmost one in the same row, and so on). If it perceives that the square it is in contains garbage, it will pick it up, carry it and then move to the square where the incinerator agent is along a

\(^2\) http://jason.sourceforge.net/wp/examples/
shortest path (diagonal movement is allowed). The cleaner agent will drop the garbage in the incinerator agent for burning, and after that it will return to the square where it just found the garbage along a shortest path (diagonal movement allowed), then continue moving along the fixed path until it reaches the last square.

In order to test the cleaner agent, we generate tests that each describe a different environment for the cleaner agent. We design the tests according to the test coverage criteria proposed by Low et al. [1]. Their criteria are based on plans and nodes (actions) in BDI agents, which are suitable for Jason agent paradigm. Fig. 6 shows the subsumption hierarchy of their criteria, from which it can be seen the topmost criteria represent the most rigorous ones. Since this Jason project is simple and doesn’t concern plan and action failure, after analyzing the AgentSpeak code of the cleaner agent we design ten tests that collectively meet the node path coverage criterion, the plan context coverage criterion and the plan path coverage criterion (we use 0-1-many rule for cyclical path coverage), and accordingly we extract three variables from each test: the location of the incinerator agent, the locations of garbage and the probability the cleaner agent has to pick up each piece of garbage successfully when it attempts to.

![Fig. 5. The Cleaning Robots example](image)

![Fig. 6. The subsumption hierarchy of the coverage criteria proposed by Low et al. (Source: [1])](image)
Since the environment is hard-coded into a java file, we use text replacement and class reload techniques in the deploy(testID) method to implement and deploy each test. We consider a mutant to be killed by a test if, at the end of the test, there is any garbage uncollected (in contrast, the non-mutated version always collects all the garbage). To implement this, we use Jason APIs and Java reflection in iMutantKilled(testID) method to check whether all the squares in the environment are empty except the two taken by the cleaner agent and the incinerator agent respectively. In addition, in getAgentTTL(testID) method, for each test, we set the lifetime of the original agent and each mutant to an amount that is enough for the cleaner agent to finish cleaning and reasonable for efficiency, through observing the original agent.

Next we configure a mutation testing process for the cleaner agent as shown in Fig. 4: first we choose r1 which is the name of the cleaner agent, and then all of its three beliefs, one initial goal and nine non-predefined plans; next we check all the implemented operators that will be applied into the chosen mutation domain. After these we start and observe the mutation testing itself.

After the mutation testing, the result is displayed, as shown in Fig. 7. From the result we can see that the three operators for agent communication – Message Receiver Replacement (MRR), Illocutionary Force Replacement (IFR) and Propositional Content Deletion (PCD2) – are not useful because this Jason project doesn’t concern agent communication. We also observe that some mutants are not killed. We track these non-killed mutants in the log of the mutation testing process and analyze their corresponding changes in the code. It appears that most of them are equivalent mutants.

For instance, BD operator produces 2 equivalent mutants. This is because two out of the three beliefs we choose for mutation – pos(r2, 3, 3) and pos(r1, 0, 0) representing the position of the incinerator agent and the initial position of the cleaner agent respectively (see Figure 4), are not specified in the cleaner agent code – they are specified in the environment code and come from immediate perception of the environment when the cleaner agent starts running, so that they can be perceived by the cleaner agent again although deleted. FOR operator produces 3 equivalent mutants, all of which are caused by replacing goal formula type “!” by “!!” or vice versa (see Section 2 for their uses). This is due to that in some cases they can be replaced by each other with no differences (or tiny differences only in efficiency of reasoning).

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>No. of Generated Mutants</th>
<th>No. of Killed Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belief Deletion</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Initial Goal Deletion</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Plan Deletion</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Rule Condition Deletion</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Triggering Event Operator Replacement</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Plan Context Deletion</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Plan Body Deletion</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Formula Deletion</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>Formula Order Swap</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Formula Operator Replacement</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Message Receiver Replacement</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Illocutionary Force Replacement</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Message Content Deletion</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>70</td>
<td>60</td>
</tr>
</tbody>
</table>

Fig. 7. The result of the mutation testing
One non-equivalent mutant that is not killed is one that deletes the formula \( \text{drop}(S) \) that is used to drop the carried garbage into the incinerator agent. This happens because the killing mutant criteria we set for each test don’t check whether the cleaner agent drops the carried garbage — it can pick all garbage without dropping any and still pass the tests.

Another non-equivalent mutant that is not killed is one that replaces the formula
\[
-\text{pos}(\text{last},X,Y) \quad \text{in plan} \quad +\text{carry_to}(R) \quad \text{by} \quad +\text{pos}(\text{last},X,Y).
\]
The former formula is used to update the belief that keeps last location where garbage was found, so that the agent can retrieve and return to this location after it drops garbage at the incinerator agent, so as to continue checking the remaining squares along the fixed path. However, when the formula is changed to \( +\text{pos}(\text{last},X,Y) \), each time the cleaner agent finds garbage, it will add a new belief representing the location of the garbage into the belief base rather than replacing the old one.

The above mutation is a fault, because it means that the agent will end up with several versions of “last location at which I found garbage” stored in its memory. In many cases, this is not a problem. When the cleaner agent has finished at the incinerator agent, it will try to take a shortest route back to last location where it found garbage. To do this, it queries for its belief about last location, and it will always retrieve the correct one because Jason’s default belief selection mechanism will always select the matching one that is added to the belief base most recently.

After each movement step, however, the agent will query “does my current location correspond to last location I found garbage” i.e. should it stop fast movement and go back to its slow side-to-side sweep of the map? If the agent is at any location where it previously found garbage, Jason’s belief query mechanism will cause the answer to that question to be “yes” - all of the “last garbage location" beliefs will be checked for a match. At that point, it will go back into its slow sweep, even though (in this simple world) there’s no chance of finding new garbage before it reaches actual last garbage location. As a consequence, the whole collection process will take longer and the agent may not collect all the garbage within its specified time-to-live.

This fault cannot be detected by any of our tests designed for the cleaner agent, because in our tests (by chance) it never passes through a previous garbage location when returning to last collected garbage location (Fig. 5 shows an example where it would happen). In order to detect this fault, we add a test that satisfies the following three conditions:

- A piece of garbage \( G_1 \), is located in a shortest path between the incinerator agent and another piece of garbage \( G_2 \).
- \( G_1 \) is found prior to \( G_2 \). This requires \( G_1 \) and \( G_2 \) must be located after where the incinerator agent is along the fixed path for the agent to check all the squares.
- \( G_1 \) and \( G_2 \) are not in the same row. This enables us to observe that the agent does indeed return to where \( G_1 \) was found after dropping either garbage for burning.

Fig. 8. shows a case in which the fault of multiple last locations can be detected. In this case, the cleaner robot will always return to the location where the garbage closer to the incinerator agent is after dropping either garbage, it will then continue moving along the fixed path from this location.
Discussion and Conclusion

In this paper we presented our preliminary work on mutation testing for multi-agent systems: we proposed a set of mutation operators for the Jason agent-oriented programming language; we described a mutation testing system called muJason for individual Jason agents, which implements the high level subset of our operators. We then used our implemented operators to assess a test set (for an example agent) that satisfies some coverage criteria proposed by Low et al. [1], and found two non-equivalent mutants that were not killed. We are hence able to improve the killing mutant criteria for killing one of these two mutants and add a test to the test set for killing the other mutant (and, probably, similar mutants or faults).

Our work draws on and expands that of Savarimuthu and Winikoff [7]: we proposed mutation operators for a specific implementation of AgentSpeak, implemented some of them and conducted a preliminary empirical assessment. They derived mutation operators systematically, while we selected our operators using our judgment so that we can preferentially implement and assess them, in the hope of avoiding implementing some ineffective ones. It can be seen that their operator set contains ours for the core AgentSpeak, but ours contain some to cover Jason-specific features useful for practical agent implementation, such as Rule Condition Deletion (RCD), Annotation Deletion (AD) and Source Replacement (SR).

Another important related work is also Savarimuthu and Winikoff’s [8]. They used the same approach proposed in [7] (i.e., applying HAZOP guidewords into the syntax of the language) to systematically derive a set of mutation operators for the GOAL agent language (like AgentSpeak, GOAL is another language for programming cognitive agents). They evaluated their set by comparing the bugs their set generates with the real ones found in students’ assignments. In contrast, we have evaluated ours by comparison with other test criteria. We believe that the combination of both evaluation approaches can bring about convincing results, so we will also use theirs in the future.

Another related work is Adra and McMinn’s [5]. Although they used a rather different agent model, some of their ideas are relevant to our work. They proposed four
mutation operator classes, among which their class for agent communication (Mis-
communication, Message Corruption) corresponds to our operators for agent commu-
nication (Message Receiver Replacement, Illocutionary Force Replacement, Proposition
Content Deletion and other involved high- and low-level operators), and their
class for an agent’s memory corresponds to our operators for beliefs (Belief Deletion,
Rule Deletion and other involved low-level operators). Their mutation operator class
for agent’s function execution does not directly correspond to our operators since our
agent model adopts the BDI reasoning mechanism, while theirs does not. As to their
mutation operator class for the environment, it is not relevant in our operators for
agents, although agent environments are an important dimension of MAS that act as
the input source of agents, and we plan to mutate environments in future work.

In the future, we will derive mutation operators for Jason’s advanced features (e.g.,
the use of directives, etc.), and implement them (along with the low-level ones we
proposed in this paper but did not implement in muJason so far). We will also apply
our approach to more complex Jason systems, and generate tests using other existing
test criteria to further evaluate the effectiveness of our proposed mutation operators.
There are challenges here – it is difficult to implement the low-level mutation opera-
tors because we need to extract the logical representations that these operators are
applied into from different agent’s architectural components.

At the same time as the above, we will study the computational cost of our muta-
tion testing and improve muJason in different aspects such as more flexible test setup
and killing mutant criteria specification, and automatic measurement on when to kill
the mutant. After that we will expand muJason to support JaCaMo [15], which is a
complete MAS programming paradigm that adopts Jason for programming agents,
Moise for programming organizations and CArtAgO for programming environments.
This will allow us to explore the mutation of organizational and environmental di-
ensions of MAS.

References

neering (ENASE-07), pp. 10–18 (2007)
3. Miller, T., Padgham, L., Thangarajah, J.: Test coverage criteria for agent interaction test-
on Agent Oriented Software Engineering, pp. 1–12 (2010)
cedings of the 3rd International Conference on Information and Communication Systems,
ICICS (2012)
Tractable Reasoning about Group Beliefs

Barbara Dunin-Kęplicz\(^1\), Andrzej Szałas\(^1,2\) and Rineke Verbrugge\(^3\)

\(^1\) Institute of Informatics, Warsaw University, Poland, keplicz@mimuw.edu.pl
\(^2\) Dept. of Computer and Information Science, Linköping University, Sweden, andrzej.szalas@mimuw.edu.pl, liu.se
\(^3\) Institute of Artificial Intelligence, University of Groningen, The Netherlands, rineke@ai.rug.nl

Abstract. In contemporary autonomous systems, like robotics, the need to apply group knowledge has been growing consistently with the increasing complexity of applications, especially those involving teamwork. However, classical notions of common knowledge and common belief, as well as their weaker versions, are too complex. Also, when modeling real-world situations, lack of knowledge and inconsistency of information naturally appear. Therefore, we propose a shift in perspective from reasoning in multi-modal logics to querying paraconsistent knowledge bases. This opens the possibility for exploring a new approach to group beliefs. To demonstrate expressiveness of our approach, examples of social procedures leading to complex belief structures are constructed via the use of epistemic profiles. As an implementation tool we choose 4QL, a four-valued rule-based query language, to achieve tractability without compromising the expressiveness. This permits both to tame inconsistency in individual and group beliefs and to execute the social procedures in polynomial time. Therefore, a marked improvement in efficiency has been achieved over systems such as (dynamic) epistemic logics with common knowledge and ATL, for which problems like model checking and satisfiability are PSPACE- or even EXPTIME-hard.

Keywords: Cooperation, reasoning for robotic agents, formal models of agency, knowledge representation, tractability

1 A New Perspective on Beliefs

Classical approaches to common knowledge capture the essence of the mutuality involved in what it means to deal with common knowledge, as contrasted with distributed knowledge. According to the usual understanding, the essence of these notions is consensus between group participants. This is clearly visible in the notion of general knowledge E-KNOW\(_G\) (every agent in group \(G\) knows), and propagation of consensus, through iterations E-KNOW\(_G^k\) up to common knowledge C-KNOW\(_G\), which informally can be seen as an infinitely iterated stack of general knowledge operators. This manner

\(^*\) This work was partially supported by Polish National Science Centre grants 2011/01/B/ST6/02769, 2012/05/B/ST6/03094 and Vici grant NWO 227-80-001.
of building common knowledge, originating from epistemology and modal logic, captures “what every fool knows” [16, 25], [12, Chapter 2]. Indeed common knowledge is helpful in drawing common consequences from commonly known premises, which is invaluable in creating models of others. But this comes at the price of super-polynomial complexity, causing grave problems when engineering multi-agent systems for use in time-critical situations [3, 13].

As the role of group knowledge has recently evolved, it may instead be useful for participants to preserve their individual beliefs, while at the same time being a member of a larger group structure with group beliefs that govern the group’s behavior. Instead of “what every fool knows”, group knowledge would then tend to express synthetic information extracted from the information delivered by individuals. Thus, more so than in classical epistemic and doxastic logical approaches, there should be a clear distinction between agents’ individual informational stances and the groups’ ones. Consensus is not a requirement anymore, as group members do not necessarily adopt group conclusions. It suffices that during the group’s lifetime they obey them.

In autonomous systems, the need to apply group knowledge has been growing with the increasing complexity of real-world applications, especially those involving cooperation or teamwork. A field that particularly expanded recently is robotics. In fact, contemporary robotics has now advanced so far that it has become necessary to investigate performance issues. Since more and more intelligent robots are able to autonomously perform sophisticated and precise maneuvers, we inevitably approach the era of strict cooperation among robots, software agents and people. Typical examples of such cooperation are emergency situations or catastrophes [1, 7, 13, 18, 27].

During robots’ cooperation, an attempt to create consensus seems to be superfluous. Instead, in time-critical situations it is essential to reduce the complexity of both communication and reasoning. It is often too computationally costly to establish and reason about common beliefs and common knowledge. Especially when the information derives from different sources and is imprecise, problems arise due to the properties discussed in [11], including limited accuracy of sensors and other devices, restrictions on time and other resources, unfortunate combinations of environmental conditions, and limited reliability of physical devices. This combination of properties inevitably introduces inconsistencies on many different levels: in the information available to individual agents, between different agents, as well as between agents and groups.

Even though in classical logical approaches, inconsistency immediately trivializes reasoning — “Ex falso sequitur quodlibet” — we intend to avoid such an effect. Robots are often sent to unknown terrains and face a need to sensibly proceed regardless of their ignorance and/or inconsistent information. This leads us to a paraconsistent approach, i.e., an approach that tolerates inconsistencies. Thus, instead of fighting with inconsistencies, we treat them as first-class citizens. Typically, they need to be resolved sooner or later, depending on the situation in question, but in some reasonable cases they can even remain unresolved (see, e.g., [17]).

How to formally model such complicated situations? First of all, Dunin-Keplicz and Szalas [9, 11] have proposed a shift in perspective: from reasoning in multi-modal

---

Paraconsistency has a long tradition and is intensively investigated (see, e.g., [2]).
systems of high complexity to querying (paraconsistent) knowledge bases. This has led to a novel formalization of complex beliefs. In order to bridge the gap between idealized logical approaches and their actual implementations, the novel notion of epistemic profile serves as a tool for transforming preliminary beliefs into final ones.

An epistemic profile reflects an agent’s individual reasoning capabilities: it defines a schema in which an agent reasons and deals with conflicting information and ignorance. These skills are achievable by combining various forms of reasoning, including belief fusion, disambiguation of conflicting beliefs, and completion of lacking information. More formally, an epistemic profile corresponds to a function mapping finite sets of ground literals to ground literals (see Definition 3). As epistemic profiles can be devised analogously both on an individual and a group level, we achieve a uniform treatment of individual and group beliefs.

Various challenges occurring when building epistemic profiles can be solved with the use of 4QL, a four-valued rule-based query language designed by Małuszyński and Szałas [21, 23, 33]. Our approach builds on ideas underlying 4QL, which allows for negation in premises and conclusions of rules. It provides simple, yet powerful constructs (modules and external literals) [21, 22] and more general multisource formulas [33] for expressing non-monotonic rules reflecting, among others, lightweight forms of default reasoning [31], auto-epistemic reasoning [26], defeasible reasoning [29], and the local closed world assumption [15]. Importantly, 4QL enjoys tractable query computation and captures all tractable queries (see [22] for details). Therefore, 4QL is a natural implementation tool opening the space for a diversity of applications by providing firm foundations for paraconsistent knowledge bases used by external applications. This paper is part of a larger research program started in [8–11, 14]. The main contributions of this article are (see also Table 1):

- Providing a tractable methodology for modeling group beliefs that ensures a proper treatment of inconsistent or lacking information, while avoiding unwanted effects like omniscience;
- Implementing examples of social procedures, leading to complex belief structures, via the use of epistemic profiles and 4QL;
- Showing how to tame inconsistency and incompleteness in individual and group beliefs;
- Showing that social procedures for creating group beliefs, expressed in 4QL and using lightweight forms of non-monotonic reasoning, can be executed in deterministic polynomial time.

In this paper we focus on belief formation rather than belief maintenance and revision. Such dynamic aspects, for which 4QL is eminently suitable, will be presented in future work.

The rest of the paper is structured as follows. Section 2 presents a robot rescue scenario to be used as running example, while Section 3 presents the logical background on belief structures, epistemic profiles and 4QL. The heart of the paper includes Section 4, which introduces methods for creating group beliefs in 4QL according to agents’

---

5 See also http://4ql.org, which provides an open source experimental interpreter of 4QL.
Table 1. Shift in perspective on group beliefs.

<table>
<thead>
<tr>
<th>Traditional approaches</th>
<th>The new approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>“What every fool knows”</td>
<td>Synthetic information extracted from individuals or other groups</td>
</tr>
<tr>
<td>Holistic knowledge</td>
<td>Selected aspects only</td>
</tr>
<tr>
<td>Consensus</td>
<td>Group members not forced to adopt group conclusions: only required to obey them during the group’s lifetime</td>
</tr>
<tr>
<td>Omniscience</td>
<td>Incomplete/inconsistent beliefs allowed</td>
</tr>
<tr>
<td>Monotonicity</td>
<td>Non-monotonic resolution of incomplete/inconsistent beliefs offered</td>
</tr>
<tr>
<td>Homogeneity (typically)</td>
<td>Heterogeneity: reasoning is individualized; heterogeneous information sources allowed</td>
</tr>
<tr>
<td>Reasoning intractable</td>
<td>Tractability: reasoning in deterministic polynomial time</td>
</tr>
</tbody>
</table>

and groups’ epistemic profiles. Section 5 focuses on solving the problem of conflicting information at the group level. Section 6 provides a formalization of the robot rescue scenario. Section 7 discusses the influence of group beliefs on members’ individual beliefs. In Section 8, we show that social procedures expressible in 4QL are in fact tractable. We end with a discussion and topics for future research in Section 9.

2 Running Example Scenario: Robot Rescue

Consider a group of robots, each equipped with a temperature sensor. In our running example, their beliefs, as hardwired by the robots’ manufacturer, are expressed by the following rules:

- if \((temperature \leq 65^\circ C)\) then operating is safe; \(1\)
- if \((65^\circ C < temperature \leq 80^\circ C)\) then risk of damage is serious; \(2\)
- if \((80^\circ C < temperature)\) then it is certain that operating is impossible. \(3\)

Assume that there is fire in certain regions, resulting in a high temperature in these regions and their neighborhoods. Let a surveillance team \(team = \{r_1, \ldots, r_k\} (k > 1)\) of robots be formed, whose group beliefs include the one that searching for victims is more important than preserving robots. An example of a group belief can be:

- enter the affected region and search for victims unless it is certain that operating in the region is impossible. \(4\)

To formalize these and related rules we shall use the following relations, where \(R\) represents regions:

- \(temp(R, T)\): temperature in \(R\) is \(T\);
- \(risk(R)\): situation in \(R\) is risky;
- \(allowed(R)\): entering \(R\) is allowed (perhaps also in a risky situation);
search(R): search for victims in R.

Let us emphasize that each agent (robot) is equipped with its individual knowledge base, so it has individual beliefs about these relations. We also assume that geographic information system (GIS)-based information about subregions and robots’ locations is available via the following relations:

- close(P, R): robot P is close to R;
- subreg(S, R): S is a subregion of R.

We use this robot rescue scenario throughout the paper.

3 Preliminaries

In what follows we assume that domains of objects are finite and that agents’ reasoning is grounded in knowledge bases rather than in arbitrary theories. That is, in reasoning we allow rules and facts and consider well-supported models only.

3.1 Language, Belief Structures and Epistemic Profiles

We view epistemic profiles as the general means to express a variety of strategies for belief acquisition and formation. In order to apply them here, we present a summary of some of the most important definitions from [9–11, 21, 23]. The semantical structures constituents and consequents reflect the processes of agents’ belief acquisition and formation. An agent starts with constituents, i.e., sets of beliefs acquired by perception, expert-supplied knowledge, communication with other agents, and many other ways. Next, the constituents are transformed into consequents according to the agent’s individual epistemic profile. Consequents contain final, “mature” beliefs.

In a multi-agent system, for each group, the group epistemic profile is set up, where consequents of group members become constituents at the group level and such constituents are further transformed into group consequents. Observe that in this way, various perspectives of agents involved are taken into consideration and merged. Similarly, groups may be members of larger groups, perhaps containing individuals, too, etc.

As to the language, we use the classical first-order language over a given vocabulary without function symbols, presented in [11, 21, 33]. We assume that Const is a fixed set of constants, Var is a fixed set of variables and Rel is a fixed set of relation symbols.

**Definition 1.** A literal is an expression of the form \( R(\bar{\tau}) \) or \( \neg R(\bar{\tau}) \), with \( \bar{\tau} \) being a sequence of arguments, \( \bar{\tau} \in (\text{Const} \cup \text{Var})^k \), where \( k \) is the arity of \( R \). Ground literals over Const, denoted by \( \mathcal{G}(\text{Const}) \), are literals without variables, with all constants in \( \text{Const} \). If \( \ell = \neg R(\bar{\tau}) \) then \( \neg \ell \overset{\text{def}}{=} R(\bar{\tau}) \).

Though we use classical first-order syntax, the semantics substantially differs from the classical one as truth values \( t, i, u, f \) (true, inconsistent, unknown, false) are explicitly
present; the semantics is based on sets of ground literals rather than on relational structures. This allows one to deal with lack of information as well as inconsistencies. Because 4QL is based on the same principles, it can directly be used as implementation tool.

The semantics of propositional connectives is summarized in Table 2. Observe that definitions of $\land$ and $\lor$ reflect minimum and maximum with respect to the ordering:

$$f < u < i < t,$$

as argued in [6, 21, 37]. Similarly, the semantics of quantifiers in formulas $\forall x.A(x)/\exists x.A(x)$ is defined using ordering (5), by taking the minimum (respectively, maximum) of the truth values of $A(a)$ for $a \in \Delta$, where $\Delta$ is the domain of $x$.

<table>
<thead>
<tr>
<th></th>
<th>$\land$</th>
<th>$\lor$</th>
<th>$\rightarrow$</th>
<th>$\neg$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$f$</td>
<td>$u$</td>
<td>$i$</td>
<td>$t$</td>
</tr>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>$i$</td>
<td>$i$</td>
<td>$i$</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>$t$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

Table 2. Truth tables for $\land$, $\lor$, $\rightarrow$ and $\neg$ (see [21, 37]).

Let $v : \text{Var} \rightarrow \text{Const}$ be a valuation of variables. For a literal $\ell$, by $\ell(v)$ we understand the ground literal obtained from $\ell$ by substituting each variable $x$ occurring in $\ell$ by constant $v(x)$.

**Definition 2.** The truth value $\ell(L, v)$ of a literal $\ell$ w.r.t. a set of ground literals $L$ and valuation $v$, is defined by:

$$\ell(L, v) \overset{\text{def}}{=} \begin{cases} 
  t & \text{if } \ell(v) \in L \text{ and } (\neg \ell(v)) \notin L; \\
  i & \text{if } \ell(v) \in L \text{ and } (\neg \ell(v)) \in L; \\
  u & \text{if } \ell(v) \notin L \text{ and } (\neg \ell(v)) \notin L; \\
  f & \text{if } \ell(v) \notin L \text{ and } (\neg \ell(v)) \in L.
\end{cases}$$

**Belief structures** can now be defined as in [9, 11]. Here, the concept of an epistemic profile is the key abstraction involved in belief formation. If $S$ is a set, then $\text{Fin}(S)$ represents the set of all finite subsets of $S$.

**Definition 3.** Let $C \overset{\text{def}}{=} \text{Fin}(G(\text{Const}))$ be the set of all finite sets of ground literals over constants in $\text{Const}$. Then:

- a constituent is any set $C \in C$;
- an epistemic profile is any function $E : \text{Fin}(C) \rightarrow C$;
- by a belief structure over epistemic profile $E$ is meant a structure $B^E = (C, F)$; here $C \subseteq C$ is a nonempty set of constituents and $F \overset{\text{def}}{=} E(C)$ is the consequent of $B^E$. $hd$

Importantly, final beliefs are represented as consequents.
3.2 The 4QL Rule Language

The rule language 4QL has been introduced in [21] and further developed in [23, 33]. Beliefs in 4QL are distributed among modules, illustrated by the following example.


The 4QL language allows for negation in premisses and conclusions of rules. It is based on the four-valued logic described in Section 3.1. The semantics of 4QL is defined by well-supported models [21–23, 33], i.e., models consisting of (positive or negative) ground literals, where each literal is a conclusion of a derivation starting from facts. For any set of rules, such a model is uniquely determined:

“Each module can be treated as a finite set of literals and this set can be computed in deterministic polynomial time” [21, 23].

Thanks to this correspondence and the fact that 4QL captures PTIME, the constituents and consequents of Definition 3, being PTIME-computable, can be directly implemented as 4QL modules (see also Theorem 1).

Remark 1. Note that this prevents the unfortunate effects of the omniscience problem (for a survey of the problem, see, e.g., [16, 19, 25, 32]): to check whether a formula $A$ belongs to a set of beliefs of an individual or a group, one only has to determine what is its truth value in the respective consequent. Formula $A$ can be considered as a query to a corresponding 4QL module, so tractability is preserved. As 4QL allows to express PTIME-computable queries only, intractable/uncomputable classes of valid formulas (e.g., expressing the consequences of the Peano axioms for first-order arithmetic) cannot be expressed as valid beliefs unless explicitly added to knowledge bases.

For specifying rules and querying modules, we adapt the language of [33]. To define the language we need the notion of multisource formulas defined as follows.

Definition 4. A multisource formula is an expression of the form: $m.A$ or $m.A \in T$, where:

- $m$ is a module name;
- $A$ is a first-order or a multisource formula;
- $T \subseteq \{t, i, u, f\}$.

We write $m.A = v$ (respectively, $m.A \neq v$) to stand for $m.A \in \{v\}$ (respectively, $m.A \notin \{v\}$).

The intuitive meaning of a multisource formula $m.A$ is:

“return the answer to query expressed by formula $A$, computed within the context of module $m$”.

294
The value of ‘\(m.A \in T\)’ is:
\[
\begin{cases}
  t & \text{when the truth value of } A \text{ in } m \text{ is in the set } T; \\
  f & \text{otherwise}.
\end{cases}
\]

Let \(A(X_1, \ldots, X_k)\) be a multisource formula with \(X_1, \ldots, X_k\) being its all free variables and \(D\) be a finite set of literals (a belief base). Then \(A\), understood as a query, returns tuples \((d_1, \ldots, d_k, tv)\), where \(d_1, \ldots, d_k\) are database domain elements and the value of \(A(d_1, \ldots, d_k)\) in \(D\) is \(tv\).

Example 2. The following formula:
\[
\exists S (gis.\text{subreg}(S, R) \land \text{temp}(S, T) \land T > 65)
\]
states that there is a subregion of \(R\) with the temperature \(T\) exceeding 65. The ‘gis’ module stores information about subregions; the part ‘gis.\text{subreg}(S, R)’ of (6) uses this module. More precisely, formula (6), understood as a query, returns triples \((\text{region}, \text{temperature}, \text{value})\) such that the truth value of formula (6) is \(\text{value}\) when \(R = \text{region}\) and \(T = \text{temperature}\).

The formula:
\[
(\exists S (gis.\text{subreg}(S, R) \land \text{temp}(S, T) \land T > 65)) \in \{t, i, u\}.
\]
is true when the value of formula (6) is \(t\), \(i\) or \(u\), and is false otherwise. ◯

Definition 5.
- **Rules** are expressions of the form:
\[
\text{conclusion} : \neg \text{premises}.
\]
where \(\text{conclusion}\) is a positive or negative literal and \(\text{premises}\) are expressed by a multisource formula.
- A **fact** is a rule with empty premises (such premises are evaluated to \(t\)).
- A **module** is a syntactic entity encapsulating a finite number of facts and rules.
- A **4QL program** is a set of modules, where it is assumed that there are no cyclic references to modules involving multisource formulas of the form \(m.A \in T\).

Openness of the world is assumed, but rules can be used to close it locally or globally close. Rules may be distributed among modules. Here follows an example, using the robot rescue scenario of Section 2.

Example 3. Consider the following rules within a module, say \(m\), of a given robot:
\[
risk(R) :\neg \text{close}(R) \land [\text{formula (7)} = t].
\]
\[
\neg \text{allowed}(R) :\text{temp}(S, T) \land T > 80.
\]
Rule (9) expresses the fact that region \(R\) is risky for the robot if it is close to \(R\) and formula (7) is true. Rule (10) states that the robot is not allowed to enter regions, where the temperature exceeds 80°C.

One can query \(m\) using multisource formulas like \(m.\text{risk}(R)\), \(m.\text{allowed}(R)\), \(m.\text{risk}(R) \in \{t, i\}\), etc. ◯

\footnote{It is assumed that formulas without a module label refer to the current module.}
4 Between Individual and Group Beliefs

Group beliefs gather conclusions of reasoning processes of the agents involved. Therefore, they are generally more synthetic than beliefs of group members, and deal with selected aspects only. If not stated differently, group beliefs prevail over individual ones. If a group belief about some aspect is missing or is inconsistent, an agent should be able to grab adequate information from its individual belief base or possibly complete it non-monotonically. These features should be reflected in the epistemic profiles (as discussed in Section 3.1).

4.1 Adjusting 4QL to Epistemic Profiles

To simplify formalization of epistemic profiles in 4QL, we shall identify consequents of robot \( r \) (or group of robots \( G \)) with a 4QL module having the same name \( r \) (respectively, \( G \)). For a truth value \( w \), we write:

- \( m.A = w \) to stand for \( m.A \in \{ w \} \);
- \( m.A \neq w \) to stand for \( m.A \in \{ t,i,u,f \} - \{ w \} \).

Although all phenomena presented in this paper are expressible in 4QL, we shall also use notation extending 4QL, yet simplifying formalizations we need. For a group of robots \( G = \{ r_1, \ldots, r_k \} \) \((k \geq 1)\):

- \( \exists r \in G[A(r)] \stackrel{\text{def}}{=} A(r_1) \lor \ldots \lor A(r_k) \);
- \( \forall r \in G[A(r)] \stackrel{\text{def}}{=} A(r_1) \land \ldots \land A(r_k) \);
- \( \#\{ r \in G \mid A(r) \} \) is the number of members of \( G \) making \( A \) true (\( A \) is assumed here not to have free variables other than \( r \)); we shall also use the abbreviation \( \#G \stackrel{\text{def}}{=} \#\{ r \in G \mid t \} \) (the number of members of \( G \)).

Example 4. Consider the robot rescue scenario. Typical rules for the robots can be:

\[ search(R) :\neg \text{team.search}(R) = t. \] \hspace{1cm} (11)

\[ \neg search(R) :\neg \text{temp}(R, T) \land T > 80. \] \hspace{1cm} (12)

The first rule states that the robot should start searching for victims in region \( R \) if \( search(R) = t \) is a belief of \( \text{team} \). If the temperature excludes the possibility of robots’ operation (see rule (3)) then the conclusion is \( \neg search(R) \). Of course, rules (11)–(12) may lead to inconsistency when the temperature in a given region is over \( 80^\circ C \) and \( \text{team} \) still believes that searching that region is in order. This inconsistency can easily be resolved. If rules (11)–(12) are in a module, say \( m \), then the robot may use a rule like:

\[ \neg search(R) :\neg m.search(R) = i. \] \hspace{1cm} (13)

Of course, one can define more refined solutions than (13). 

\[ < \]
4.2 Establishing Group Belief

Common knowledge and its weaker approximations, such as iterated general knowledge, can be viewed as a paradigmatic form of group knowledge. However, for many applications this is too much to ask for. After all, when using standard modal logics, such as in [12, 25], the levels of iterated general beliefs harbor the risk of combinatorial explosion. Even for a group as small as three agents, $G = \{1, 2, 3\}$, we have:

\[
E-B_{G}(p) \Leftrightarrow \text{BEL}(1, p) \land \text{BEL}(2, p) \land \text{BEL}(3, p);
\]

\[
\text{for } k \geq 1: E-B_{G}^{k+1}(p) \Leftrightarrow E-B_{G}(E-B_{G}^{k}(p)).
\]

Observe that (15), when written in full, has $3^{k+1}$ conjuncts, so the complexity of building levels of general belief is exponential in the number of required levels, therefore not computable in polynomial time. Thus, for time-critical applications, one should completely change the approach to group belief.

Actually, full-fledged general and common belief is not needed for many real-world applications. The necessary shared belief state may result from agreement, some example methods of which will be listed in Section 4.3. On the other hand, the notion of distributed knowledge is sometimes referred to as “what a wise person would know”. This wise person would pull together the individual knowledge of group members, and draw only classical conclusions from the combined information [25]. Distribution of reasoning is also an important feature of our approach but why should we limit ourselves to classical reasoning only? Group knowledge may go even further than traditional distributed knowledge or belief: when starting from the same individual beliefs of the group members, a variety of reasoning methods and other techniques may lead to much more far-reaching conclusions. Epistemic profiles are introduced to encapsulate the variety of techniques used.

4.3 Building Epistemic Profiles

Creation of group beliefs takes place in the broader context of producing derivatives, understood here as a complex process of drawing conclusions by different, temporarily existing, virtual subgroups or intermediate views [10, 11]. When the final consequent has been reached, the virtual subgroups involved may disintegrate, while the consequent itself is spread among initial group members. This whole process, reflected in Figure 1 (from [10] with permission), can take place at any level of group aggregation.

Using well-known heuristics, agents and groups have the possibility to complete their knowledge. Several reasoning methods can be used in the context of 4QL, as discussed in [8, 10, 14]:

- non-monotonic reasoning including the local closed-world assumption;
- default reasoning, circumscription, etc.;
- defeasible reasoning;
- methods inspired by argumentation theory.

A variety of social procedures, in combination with the reasoning methods above, may be used to establish different types of group knowledge or belief:
– public announcements [36];
– different voting methods [28];
– methods involving power relations [4, 35].

Example 5. Assume that agents in group $G$ vote about the truth value of the formula:

$$temp(R, T) \land T > 65.$$  \hfill (16)

A simple way to encode such majority voting is:

$$risk(R) := \#\{r \in G \mid r.[(16)] = t\} > \#\{r \in G \mid r.[(16)] = f\}.$$ 

The above rule can be made more subtle, e.g., by setting:

$$risk(R) := \#\{r \in G \mid r.[(16)] \in \{t, i\}\} > \#\{r \in G \mid r.[(16)] \in \{f, u\}\}.$$ 

Of course, such voting may be made more context-dependent by using relations other than those occurring in (16).

In appropriate circumstances, one may choose seeing rather than communicating as a method to create group belief. This can be seen as an analogy to “co-presence” [5]: by joint attention, the information is seen by everybody and everybody knows that the others in the group see this, and so on. Formally, this is more restrictive than the majority voting of the above example; for the robot rescue example such “co-presence” could follow the rule:

$$risk(R) := \#\{r \in G \mid r.[(16)] = t\} = \#G.$$ 

The relevant combination of social procedures and reasoning techniques is to be implemented as individual and group epistemic profiles by means of multisource formulas and 4QL modules.
4.4 Creating Virtual Groups

Sometimes a virtual group is created (among other reasons) to create appropriate group beliefs. Whenever this happens, the virtual group’s reasoning method has to be fixed, either implicitly or explicitly, and then represented in the virtual group’s epistemic profile.

However complex the process of drawing a consequent from the constituents may be in terms of subgroups involved, at the end, the resulting consequent is seen by members of the initial group only. Analogously, in order to answer a question in daily life, you may look at Wikipedia, ask experts, and ask friends what they think about the issue. When you have finally drawn your conclusion, you often forget about the details of this process and do not necessarily communicate your final conclusion to all people involved, but only to those who need to know. This makes the process less complex and safer from the perspective of information security and, not to forget, also more relaxed.

The next important issue is a proper organization of reasoning processes and information sharing between different groups and/or agents belonging to different groups at the same time. As in everyday life, during an agent’s reasoning and activities as a member of one group, the beliefs of other groups the agent belongs to are temporarily suspended or hidden. In this situation, the agent sees only its individual and the current group beliefs. This way switching between groups becomes simple and computationally efficient. When a group belief is formed, this does not force each member to change its individual informational stance (Section 7). Relaxing this postulate creates an important difference from the attainment of common knowledge in the modal logic framework.

5 Conflicting Information

Whenever conflicting information appears, it may be resolved on the individual or group level in a similar way. If there is no means to resolve it within given time and other constraints, the group can resort to less resource demanding kinds of heuristics. As to timing, there are at least three strategies:

- “Killing inconsistency at the root”: to solve them as soon as possible;
- On the other extreme, “living with inconsistency”: postpone disambiguation to the last possible moment (or even forever);
- Intermediate: solving inconsistency each time new relevant information appears.

In the sequel, we focus on techniques for resolving inconsistencies as those are generally independent of timing strategies.

5.1 Examples of Techniques

The context of the following simple examples is a group of robots in the rescue scenario deciding on the truth value of $\text{search}(X)$, which is crucial in their decision-making about whether action is needed.
Example 6. One can resolve potential inconsistencies using one of the following example policies.

- No matter what, if a group member evaluates $search(R)$ to be $t$, then do search:

$$search(R) :\exists r \in \text{team}[r.search(R) = t].$$

(17)

- Search if no group member claims that one should not:

$$search(R) :\forall r \in \text{team}[r.search(R) \neq f].$$

(18)

- Search if at least one group member claims one should and no group member claims that one should not:

$$search(R) :\exists r \in \text{team}[r.search(R) = t] \land (18).$$

(19)

Of course, there are many other reasonable ways for resolving inconsistencies, some of them discussed below.

In more complex scenarios, techniques for resolving inconsistency may reflect knowledge about the application domain involving legal regulations, argumentation, or other accepted strategies, such as the social procedures on which we focus next.

5.2 Social Procedures Solving Inconsistencies

In the subsequent example cases, the robots use different procedures to resolve inconsistent information about whether an area is risky, $risk(reg)$.

Case A: peer-to-peer Solving inconsistencies among peers may not be immediately possible. A possibility is to ignore the $i$-values and decide that on the group level, $risk(reg)$ is true. This solution takes the majority vote among the $t$ and $f$ votes only and is computationally very simple, as the following example solutions indicate.

Example 7. Suppose $G = \{r_1, r_2, r_3\}$ and one agent assigns value $i$ to $risk(reg)$ while two other agents assign $t$. It seems reasonable that the group then considers $risk(reg)$ to be true. The following rule formalizes this approach.

$$risk(reg) :\exists r \in G(r.risk(reg) = i) \land \#\{r \in G | r.risk(reg) = t\} = 2.$$  

Of course, this solution may be modified in particular cases, for example, when the agent voting for $i$ is much more reliable in estimating risk than other team members.

Example 8. Let again $G = \{r_1, r_2, r_3\}$. Now suppose two agents assign value $u$ to $risk(reg)$, while one agent assigns $t$ to it. What should be done with this lack of information? In case of majority voting, it seems fine to ignore the $u$ votes and restrict to taking the majority among the $t$ and $f$ votes. Also for larger groups, even if there are many agents assigning $u$ to the formula, it still makes sense to compute the majority among the $t$ and $f$ votes only, as done in Example 5. 

300
Case B: with authority or outside expert Let us describe several possible procedures using the framework of 4QL, in the context of the robot rescue scenario.

Procedure B1: A group belief identified with the leader’s or an expert’s belief

Suppose $\text{expLead}$ is a consequent of an expert or leader knowledge base, deciding whether certain regions are risky. If the expert’s or leader’s value of $\text{risk}(R) = t$, then the group value corresponds. The following rules can then be used to express team’s consequents as to the risk:

$$\text{risk}(R) := \text{expLead.risk}(R) = t.$$  

$$\neg \text{risk}(R) := \text{expLead.risk}(R) \in \{u, i, f\}.$$  

Procedure B2: Conditional choice between leader, expert, and majority

A safer choice is to use all information about $\text{risk}(R)$ based on trustworthiness:

“If there is an outside expert on $\text{risk}(R)$, then we take his decision that $\text{risk}(R) = t$ as the group decision; else, if the leader’s evaluation of $\text{risk}(R)$ is $t$, then we take on the leader’s decision as group belief; else, we cast a majority vote.”

This is reflected in the following rules, where $\text{exp}$ is a group of outside experts and $\text{lead}$ is the leader:

$$\text{risk}(R) := \exists e \in \text{exp}[e.\text{risk}(R) = t]. \quad (20)$$

$$\text{risk}(R) := \forall e \in \text{exp}[e.\text{risk}(R) \neq t] \land \text{lead.risk}(R) = t. \quad (21)$$

$$\text{risk}(R) := \forall e \in \text{exp}[e.\text{risk}(R) \neq t] \land \text{lead.risk}(R) \neq t \land \text{‘risk}(R) = t \text{ wins voting’}. \quad (22)$$

Note that the voting in the last line can be formalized along the lines of Example 5.

To infer negative conclusions as to $\text{risk}(R)$, one can add rules negating conclusions and premisses of (20)–(22). For example, adding such negations in rule (20) we obtain:

$$\neg \text{risk}(R) := \neg \exists e \in \text{exp}[e.\text{risk}(R) = t].$$

One could also close the relation $\text{risk}$ in various ways. If rules (20)–(22) are defined in module $m$, then the simplest closure can be obtained using the following rule (in a module other than $m$):

$$\neg \text{risk}(R) := m.\text{risk}(R) \neq t.$$  

6 Robot Scenario Formalized

Let us now formalize an illustrative example of an epistemic profile for team using the robot rescue scenario of Section 2. Recall that 4QL modules can be identified with sets of literals. In what follows we use this identification.
The team’s belief structure consists of constituents:

– consequents of each robot \(r_1, \ldots, r_k\);
– the gis module.

To define team’s epistemic profile, we use the following derivatives:

– allClose, containing the relation risk, calculated according to votes of all agents close to a given region;
– safe, containing the relation allowed, stating that searching a given region is allowed (no certainty of damaging robots there).

The above derivatives are used for illustration purposes only. The module allClose contains the following rules:

\[
\text{risk}(R) :- \# \{ r \in \text{team} \mid \text{gis.close}(r, R) = t \wedge r.\text{risk}(R) = t \} > \# \{ r \in \text{team} \mid \text{gis.close}(r, R) = t \wedge r.\text{risk}(R) \neq t \}.
\]

\[
\neg \text{risk}(R) :- \# \{ r \in \text{team} \mid \text{gis.close}(r, R) = t \wedge r.\text{risk}(R) = t \} \leq \# \{ r \in \text{team} \mid \text{gis.close}(r, R) = t \wedge r.\text{risk}(R) \neq t \}.
\]

The module safe contains the rule:

\[
\neg \text{allowed}(R) :- \exists r \in \text{team} (\text{gis.close}(r, R) = t \wedge r.\text{temp}(R, T) = t \wedge T > 80).
\]

The team’s consequent can be defined, for example, by the following rules:

\[
\text{risk}(R) :- \text{allClose}.\text{risk}(R). \tag{23}
\]
\[
\neg \text{risk}(R) :- \text{allClose}.(\neg \text{risk}(R)) \wedge \text{safe.allowed}(R) \neq \text{f}. \tag{24}
\]
\[
\text{search}(R) :- \text{safe.allowed}(R) \neq \text{f}. \tag{25}
\]

Of course, robots may have individual beliefs about risk and search(R) contradicting (23)–(25). These inconsistencies can be resolved by a rule similar to (13), concluding that a robot cannot search regions where it cannot operate without being damaged.

### 7 From Groups Down to Agents

Group belief may be naturally used to clarify agents’ individual beliefs. For example, if for some agent \(r\) the value of \(\varphi\) is \(u\) or \(i\), and for group \(G\) the value became one of \(t, f\), then generally it makes sense for \(r\) to adopt this latter truth value. Formally, this could be handled by a default rule in the agent epistemic profile, where we distinguish between a constituent of \(r\), denoted by \(c\), and its consequent, denoted by \(r\):

If \(c.P \in \{i, u\}\) and \(G.P \in \{t, f\}\) (prerequisite) and it is consistent that “special situation (S) does not occur” (justification), then \(r.P\) becomes \(G.P\):

\[
r.P :- c.P \in \{i, u\} \wedge G.P = t \wedge S \in \{f, u\}.
\]

\[
\neg r.P :- c.P \in \{i, u\} \wedge G.P = f \wedge S \in \{f, u\}.
\]
This way, coherence of knowledge can be maintained. The process does call for calculating a new well-supported model. Such downward reflection is useful when the group decides about critical situations. Then each individual should follow this.

When a decision is not life-critical, different opinions remain possible. For example, when a jury decides that the Best Paper Prize should be given to A while an individual jury member would have preferred B, (s)he can keep her/his opinion while the group decision stands. Similarly when a program chair decides that a certain paper is acceptable for the proceedings, individual program committee members do not need to agree to the group decision. The mode of adaptation to group beliefs needs to be included in everyone’s epistemic profile. This real-world model of the information-flow between a group and its individual members fits to many contexts better than common knowledge.

8 Complexity

Consider a static situation without knowledge base updates. Thus, we have a snapshot of a system consisting of, say, \( k \) individuals and \( n \) groups, each of them computing its consequents according to its epistemic profile (Definition 3). Since data complexity of 4QL is PTIME and 4QL captures PTIME (see [22, 23]), we have the following result, where as usually finite domains are assumed.

**Theorem 1.** Assume that the number of constituents of each individual as well as the number of belief structures associated with each individual/group is bounded by a constant.

- If each constituent and epistemic profile involved is implemented in 4QL, then the complexity of computing them all is \( O((k+n)\cdot p(|\text{Const}|)) \), where \( p \) is a polynomial and \( \text{Const} \) is the set of constants occurring in constituents and epistemic profiles.
- Every epistemic profile/belief structure computable in deterministic polynomial time (PTIME with respect to data complexity) can be expressed in 4QL (assuming linear ordering on \( \text{Const} \) is given).  

Note that in this way, tractability is achieved. Though complexity depends on \( k \) and \( n \), these parameters reflect the numbers of individuals and groups involved in a given mission. Such individuals and groups must have been generated somehow, so we can safely assume the existence of computational capacity to handle them.

If system dynamics is considered, Theorem 1 guarantees that every time updates need to be performed, they can be done in deterministic polynomial time. In fact, the role of 4QL is to provide firm foundations for knowledge bases used by applications external to 4QL. Therefore, there may be loops in managing beliefs when circular dependencies among individuals and groups occur in applications. However, it is the responsibility of application designers to avoid such loops.
9 Discussion and Conclusions

In the current literature on knowledge and beliefs, modal logic-based approaches are dominant. Even though they suit very well to idealized epistemic theories, they are hardly applicable to real-world complex scenarios. In contrast, in the current paper we offer a novel approach to group beliefs, intended to bridge the gap between theory and applications.

We also introduce a variety of social procedures for creating group beliefs within a paraconsistent four-valued framework offered by 4QL, allowing for tractable reasoning. Importantly, our approach does not share unwanted omniscience effects like consequential closure or irrelevant belief handling.

To the best of our knowledge, a paraconsistent approach to beliefs has so far mainly been pursued in the context of belief revision [24, 30], not the creation of group beliefs. These other approaches substantially differ from ours. Their models are based on criteria and rationality indexes [30] or on relevant logic [24].

Accepting four rather than two logical values considerably simplifies our approach where one is not forced to find general embeddings of \( \{ t, i, u, f \} \) into \( \{ t, f \} \) that would work in all considered contexts. Instead, we offer a framework in which such embeddings can much easier be obtained either totally or partially, or even avoided altogether, in a highly context- and user-dependent manner. To our knowledge, such flexibility, expressiveness and at the same time tractability has not been achieved before.

We have taken into account that agents are heterogeneous in the ways that they reason; this in contrast to classical epistemic logics, which view agents as if they were homogeneous; a recent exception is the work [20]. Agents’ reasoning patterns may differ significantly, which is reflected in the epistemic profiles of individual agents as well as of different (sub-)groups.

We have also proposed some extensions to 4QL, allowing one to express a rich repertoire of combinations of social procedures with non-monotonic reasoning techniques and inconsistency disambiguation, based on the possibilities of 4QL. Although these extensions can be expressed in “pure” 4QL, we have achieved their substantial simplification here, which also is a novel contribution.

We have represented epistemic profiles, belief structures and social procedures for creating group belief in 4QL, discussing a number of example procedures of increasing intricacy. Theorem 1 then shows that all these aspects can be executed in polynomial time. This is a marked improvement over some of the most well-known logics for multi-agent systems. More precisely, for modal logics incorporating common knowledge or common belief, model checking is PSPACE-complete, while the satisfiability problem is EXPTIME-complete [12, 25, 36]. For logics of propositional control and coalition logics, both model checking and satisfiability are PSPACE-complete [35]. Finally, for alternating-time temporal logic (ATL), both model checking and satisfiability are even EXPTIME-complete [34, 38]. In real-time applications like time-critical teamwork, the advantages of using a tractable approach such as the one advocated here are essential.

This paper is part of a larger research program. Here, we focus on belief formation in heterogeneous groups, while dynamical aspects, such as maintenance of group beliefs

304
and belief revision, are left to future research. A general problem in robotics is how the activities of different groups dovetail and interleave together. This needs to be smartly organized to allow agents to smoothly switch between activities in different groups. While the focus of this paper is agents’ reasoning via individual and group epistemic profiles, in future work we will discuss the organizational part of group activities.

References


305


Abstract. Integrating knowledge representation approaches with agent programming and automated planning is still an open research challenge. To explore the combination of those techniques, we present a semantic model of planning domains that can be converted to both agent programming plans as well as planning problem definitions. Our approach allows the representation of agent plans using ontologies, enabling the integration of different formalisms since the knowledge in the ontology can be reused by several systems and applications. Ontologies enable the use of semantic reasoning in planning and agent systems, and such semantic web technologies are significant current research trends. This paper presents our planning ontology, exemplifies its use with an instantiation, and shows how to translate between ontology, agent code, and planning specifications. Algorithms to convert between these formalisms are shown, and we also discuss future directions towards the integration of semantic representation, automated planning, and agent programming.

Keywords: ontology, knowledge representation, agent plan, automated planning

1 Introduction

Knowledge representation approaches using ontologies are being studied as promising techniques to enable semantic reasoning, knowledge reuse, interoperability, and so on. However, the use of ontologies integrated with agent systems and planning formalisms is still a research path at its initial steps. To investigate this issue, we present a semantic model to represent the knowledge about planning domains.

More specifically, we developed an ontology encoded in OWL (Web Ontology Language) [1] to model planning domains based on the HTN (Hierarchical Task Network) paradigm [2]. This conceptualisation was instantiated in the Protégé[1] ontology editor to model a classical problem, known as “Gold Miners”. This example demonstrates how planning domains can be modelled in our ontology, and we also show the equivalent agent plans and planning specifications generated from this scenario.

\[1 \text{http://protege.stanford.edu/}\]
Furthermore, we propose algorithms to convert the OWL planning ontology to different formalisms, such as agent programming plans in AgentSpeak [3] and planning problem domain specifications in SHOP (Simple Hierarchical Ordered Planner) [4]. These algorithms to automatically translate from OWL to other formalisms (and vice-versa) were implemented in Java using the OWL API [5]. Therefore, planning domains instantiated in the ontology can be automatically converted to AgentSpeak [3] or SHOP [4] code (and the other way around) using the aforementioned methods. This work aligns the fields of knowledge representation and reasoning with the domain of automated planning, and this opens the path to interesting research directions that are still beginning to emerge in the relevant communities.

For instance, our approach enables to derive planning domain models and agent programming plans from existing ontological knowledge, and also to convert again from these formalisms to ontology representations. In other words, this work investigates the integration of ontologies with agent programming and other planning formalisms in order to explore semantic representations of planning domains. Thus, our goal is to explore and demonstrate the utilisation of ontologies more expressively than previous work in automated planning and agent-oriented development.

This paper is organised as follows. Next section provides a comprehensive background on ontologies, focusing on preparing the reader to relate ontologies with agent-oriented programming and planning formalisms. A section of related work is presented afterwards to map the state of the art on using ontologies in planning systems. Then, a section explaining our conceptualisation (TBox, i.e., Terminological Box) is presented. This conceptualisation is composed of classes and properties to represent planning domains. Next, we show an instantiation (ABox, i.e., Assertion Box) of this TBox in order to demonstrate how to use the proposed ontology to model a corresponding planning problem. We explain how to convert from our planning ontology to AgentSpeak [3] plans; and also from the ontology to SHOP [4] domain definitions. Algorithms coded in Java with the OWL API [5] to make these conversions are discussed afterwards. Then, we conclude this paper and point out other possible investigations and research directions towards the integration of ontology, planning and agent development.

2 Ontologies

Ontology was born as a philosophical study of reality aiming at defining which things exists in reality and what we can say about them. Researchers in Artificial Intelligence and Computer Science define ontology as an “explicit specification of a conceptualisation” [6]. In this context, a conceptualisation stands for an abstract model of some aspect of the world which defines properties of important concepts and relationships. From this definition, we can observe that an ontology is a knowledge representation structure composed of concepts, properties, individuals, relationships and axioms [7], as described in sequence. A concept is an abstract group, set, class or collection of objects that share common properties. This component is represented in hierarchical graphs, such that it looks similar to object-oriented systems. A property is used to express relationships between concepts in a given domain. More specifically, it describes the relationship between the first concept (i.e., the domain), and the second, which rep-
represents that property range. For example, “study” could be represented as a relationship between the concept “person” (as the property domain) and “university” or “college” (as range). An individual is the “ground-level” component of an ontology which represents a specific element of a concept or class. Individuals are also called instances, objects or facts. A relationship is an instance of a property, which relates two individuals: one in the relationship domain, and one in its range. It is important that those individuals obey the constraints represented in the property specification in order to guarantee the consistency of the ontology instantiation. An axiom is used to impose constraints on the values of classes or individuals, so axioms are generally expressed using logic-based languages, such as first-order logic. Axioms, also called rules, are used to verify the consistency of the ontology and to perform inferences.

The use of ontology empowers the execution of some interesting features, such as semantic reasoners and semantic queries. Semantic reasoners, for example Pellet [8], provide the functionalities of consistency checking, concept satisfiability, classification and realisation. Consistency checking ensures that an ontology does not contain contradictory facts; concept satisfiability checks if it is possible for a concept to have instances; classification computes the subclass relations between every named class to create the complete class hierarchy; and realisation finds the most specific classes that an individual belongs to [8]. In other words, semantic reasoners are able to infer logical consequences from a set of axioms. Reasoners are also used to apply rules such as the ones coded in SWRL (Semantic Web Rule Language). Moreover, ontologies can be semantically queried through SQWRL (Semantic Query-enhanced Web Rule Language), which is a simple and expressive language for implementing semantic queries in OWL [9]. OWL is a semantic web standard formalism intended to explicitly represent the meaning of terms in vocabularies and the relationships between those terms [10].

OWL is based on Description Logics (DL), which formed the basis of several ontology languages [7]. The name DL is motivated by the fact that the important notions of the domain are specified by concept descriptions, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL. DLs are usually equipped with a terminological and an assertional formalism [7]. Terminological axioms introduce names (abbreviations) for complex descriptions, and terminological axioms compose the TBox. The assertional formalism states properties of individuals and such assertions form the ABox [7]. DL systems provide various inference capabilities to deduce implicit knowledge from the explicitly represented knowledge [7]. For example, the subsumption algorithm determines subconcept-superconcept relationships; the instance algorithm infers instance relationships; and the consistency algorithm identifies whether a knowledge base (consisting of a set of assertions and a set of terminological axioms) is non-contradictory. Therefore, the classes, properties and axioms of an ontology compose its TBox, while the individuals and relationships comprise its ABox.

OWL is a language based on DL for processing web information that became W3C recommendation in February 2004 [11]. OWL basic components are classes, properties and individuals. We can say that a class is disjoint from other classes using the owl:disjointWith element, and equivalence between classes can be defined using a owl:equivalentClass element. Considering the definition of concepts, suppose we
wish to declare that the class C satisfies certain conditions, that is, every instance of C satisfies these restrictions, and/or that every instance that satisfies these restrictions can be inferred as belonging to C. OWL provides the following elements to represent these class conditions [10]: owl:allValuesFrom to define the class of possible values that the property can take (in terms of logic, it is an universal quantification, i.e., all values of the property must come from this class); owl:minCardinality to represent a cardinality restriction, requiring a minimum number of relationships (it is the opposite of the owl:maxCardinality, which imposes an upper limit of relationships); owl:someValuesFrom to represent the existential quantification; and owl:hasValue to state that the property must have a specific value. OWL was defined with two kinds of properties [10]: object properties, which relate objects (instances of classes, that is, interesting elements in the domain of discourse) to other objects; and datatype properties, which relate objects to datatype values. Also, OWL allows the definition of some characteristics of property elements directly [10], such as if the property is transitive, symmetric, functional, and so on.

Ontologies and rules are two established paradigms in knowledge modelling [11], and OWL ontologies can be combined with rules, such as SWRL [12]. To improve the expressiveness of OWL ontologies, SWRL was proposed as a rule extension of OWL that adheres to the open-world paradigm. SWRL adds to the expressive power of ontologies by allowing the modelling of certain axioms which lie outside the capability of OWL DL [11], based on a high-level abstract syntax for Horn-like rules. The rules are of the form of an implication between an antecedent (body) and a consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold (be true). A rule has the form [12]: antecedent \( \Rightarrow \) consequent, where both antecedent and consequent are conjunctions of atoms written \( a_1 \land \ldots \land a_n \). Variables are indicated using the standard convention of prefixing them with a question mark (e.g., ?x). Using this syntax, a rule asserting that the composition of parent and brother properties implies the uncle property would be written as follows [12]:

\[
parent(?x, ?y) \land brother(?y, ?z) \Rightarrow \text{uncle}(?x, ?z)
\]

Both the antecedent and the consequent of a rule might consist of zero or more atoms. On one hand, an empty antecedent is treated as trivially true (i.e., satisfied by every interpretation), so the consequent must also be satisfied by every interpretation. On the other hand, an empty consequent is treated as trivially false (i.e., not satisfied by any interpretation), so the antecedent must also not be satisfied by any interpretation. Multiple atoms are treated as a conjunction [12]. A SWRL atom can be unary (such as a class expression) or binary (such as an object property). Moreover, the arguments in atoms are of the form of individuals or data values.

Given this technological development, it is natural to think that there would be many advantages in using it more expressively in agent-oriented software engineering. The work reported in [13] pointed out to the following advantages of such integration: (i) more expressive queries in the belief base, since its results can be inferred from the ontology and thus are not limited to explicit knowledge; (ii) refined belief update given
that ontological consistency of a belief addition can be checked; (iii) the search for a plan to deal with an event is more flexible because it is not limited to unification, i.e., it is also possible to consider subsumption relationships between concepts; and (iv) agents can share knowledge using ontology languages, such as the case of OWL.

This section presented a background on ontologies, where we can observe that several advantages can emerge by using them more expressively in agent-oriented software engineering and planning. Next section investigates the state of the art regarding related studies integrating ontologies with artificial intelligence planning approaches.

3 Related Work

The work in [14] explains how an OWL reasoner can be integrated with an artificial intelligence planner. Investigations on the efficiency of such integrated system and how OWL reasoning can be optimized for this context were also presented. In their approach, the reasoner is used to store the world state, answer the planner’s queries regarding the evaluation of preconditions, and update the state when the planner simulates the effects of operators. Also, they described the challenges of modelling service preconditions, effects and the world state in OWL, examining the impact of this in the planning process. Specifically, the SHOP2 HTN planning system was integrated with the OWL DL reasoner Pellet to explore the use of semantic reasoning over the ontology [14].

A generic task ontology to formalise the space of planning problems was proposed in [15]. According with its authors, this task ontology formalises the nature of the planning task independently of any planning paradigm, specific domains, or applications and provides a fine-grained, precise and comprehensive characterization of the space of planning problems. The OCML (Operational Conceptual Modelling Language) was used to formalise the task ontology proposed in [15], since it was argued that this language provides both support for producing sophisticated specifications, as well as mechanisms for operationalising definitions to provide a concrete reusable resource to support knowledge acquisition and system development.

Another related work [16] defines a series of translations from ontologies to planning formalisms: one from OWL-S process models to SHOP2 domains; and another from OWL-S composition tasks to SHOP2 planning problems. They describe an implemented system which performs these translations using an extended SHOP2 implementation to plan with over the translated domain, and then executing the resulting plans. In summary, the work of [16] explored how to use the SHOP2 HTN planning system to do automatic composition in the context of Web Services described in OWL-S ontologies.

Reference [17] proposes a planning and knowledge engineering framework based on OWL ontologies that facilitates the development of domains and uses Description Logic (DL) reasoning during the planning steps. In their model, the state of the world is represented as a set of OWL facts (i.e., assertions on OWL individuals), represented in an RDF (Resource Description Framework) graph; actions are described as RDF graph transformations; and planning goals are described as RDF graph patterns. Their planner integrates DL reasoning by using a two-phase planning approach that performs DL reasoning in an off-line manner, and builds plans on-line, without doing any reasoning.
Their planner uses a subset of DL known as DLP (Description Logic Programs) that has polynomial time complexity and can be evaluated using a set of logic rules.

We can observe from this literature review on related work that several authors are proposing semantic representation of planning domains in ontologies. Also, approaches to translate among planning formalisms and ontologies are usually explored, so as the use of semantic reasoners before or during the planning steps. However, to the best of our knowledge, our work is the first to address the integration of ontologies in OWL [1] not only with the HTN [2] formalism but also with agent programming plans to propose the planning ontology presented in next section. We explored how this ontology can generate both agent plans in AgentSpeak [3] and SHOP [4] specifications of planning problem domains.

4 The Planning Ontology Conceptualisation

In classical planning, the main aim of the planning task is to attain a goal-state, which is usually specified in terms of a number of desired properties of the world. To model this domain, we developed an ontology, encoded in OWL [1] and built with Protégé, to represent HTN planning domains. Protégé is an open source ontology editor which also enables the visualisation of ontologies in different ways, the execution of semantic reasoners, and several other interesting features. The concepts and properties modelled in our proposed HTN planning ontology can be visualised in Figure 1. The conceptualisation was created based on the definitions of [2], [18] and [19], and a description of these concepts can be found next:

- **DomainDefinition**: A domain definition is a description of a planning domain, consisting of a set of methods, operators, and axioms.

- **Operator**: Each operator indicates how a primitive task can be performed. It is composed of: name, parameters, preconditions, a delete list and an add list giving the operator’s negative and positive effects.

- **Method**: Each method indicates how to decompose a compound task into a partially ordered set of subtasks, each of which can be compound or primitive. The simplest version of a method has three parts: the task for which it is to be used, the preconditions, and the subtasks that need to be done in order to accomplish it.

- **Axiom**: Axioms can infer preconditions that are not explicitly asserted in the current state. The preconditions of methods or operators may use conjunctions, disjunctions, negations, universals and existential quantifiers, implications, numerical computations and external function calls.

- **Predicate**: A predicate has a name and it contains any number of parameters. Predicates are used to represent the preconditions and postconditions of actions, as well as the state of the world (i.e., the state of affairs).

- **Parameter**: A parameter is a variable symbol whose name begins with a question mark (e.g., as ?x or ?agent), and it is used by operators, methods and predicates.

- **MethodFlow**: The flows of a method specify how it can be decomposed based on the current state of the world (which is represented in predicates). Thus, each method flow contains an ordered list of preconditions and an ordered list of methods or operators invocations. Each method must contain at least one flow.
- **ProblemDefinition:** Planning problems are composed of logical atoms (i.e., initial state) and task lists (high-level actions to perform), which means, a set of goals.
- **Goal:** Goals in HTN are method invocations with specific parameters that the planner will have to decompose in a sequence of operators (i.e., a plan).
- **InitialState:** An instance of initial state models the problem by means of predicates that represent the state of the world at the beginning of the simulation.

![Fig. 1. Concepts and properties of the planning ontology](image)

The concepts that are used as domain or range of each property in the proposed HTN planning ontology are presented in Table 1. Some object properties have only one concept as domain and/or range (e.g., the property `has-operator` has `DomainDefinition` as domain and `Operator` as range). However, logical expressions were also used to include more than one concept in this slot, such as the case of the `has-postcondition` property that has the `MethodFlow` concept as domain and the expression “Operator or Method” as range.

Besides the classes and properties, OWL annotations were used to represent additional information in the relationships of this ontology instantiations. When representing relationships with predicates or parameters, the order in which they have to appear must be known, which is annotated when a property targeting one of them is instantiated. Annotations are also the best choice to model logical expressions among predicates and which parameters are required when a method or operator instance relates with a predicate. Three new annotations were designed with this purpose, named: `position`, `logicalExpression` and `parameters`. The `position` annotation stores the location where that element must be written in the corresponding files, and it can be used in the following properties: `has-flow`, `has-precondition`, `adds-predicate`, `deletes-predicate`, `uses-parameter` and `has-parameter`. The `logicalExpression` annotation was created to be used only in relationships involving the `has-precondition` property. Finally, the `parameters` annotation must be used only within the properties `has-precondition`, `adds-
Table 1. Domain and range of each property in the planning ontology

<table>
<thead>
<tr>
<th>Domain</th>
<th>Property</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainDefinition</td>
<td>has-operator</td>
<td>Operator</td>
</tr>
<tr>
<td>DomainDefinition</td>
<td>has-method</td>
<td>Method</td>
</tr>
<tr>
<td>DomainDefinition</td>
<td>has-axiom</td>
<td>Axiom</td>
</tr>
<tr>
<td>InitialState</td>
<td>has-predicate</td>
<td>Predicate</td>
</tr>
<tr>
<td>Method</td>
<td>has-flow</td>
<td>MethodFlow</td>
</tr>
<tr>
<td>Operator</td>
<td>adds-predicate</td>
<td>Predicate</td>
</tr>
<tr>
<td>Operator</td>
<td>deletes-predicate</td>
<td>Predicate</td>
</tr>
<tr>
<td>Predicate</td>
<td>uses-parameter</td>
<td>Parameter</td>
</tr>
<tr>
<td>ProblemDefinition</td>
<td>has-domain</td>
<td>DomainDefinition</td>
</tr>
<tr>
<td>ProblemDefinition</td>
<td>has-goal</td>
<td>Goal</td>
</tr>
<tr>
<td>Method, Operator or Predicate</td>
<td>has-parameter</td>
<td>Parameter</td>
</tr>
<tr>
<td>MethodFlow or Operator</td>
<td>has-precondition</td>
<td>Predicate</td>
</tr>
<tr>
<td>MethodFlow</td>
<td>has-postcondition</td>
<td>Operator or Method</td>
</tr>
</tbody>
</table>

Predicate and deletes-predicate. This annotation was employed in order to relate instances of predicates used to define specific operators and methods with instances of parameters.

Figure 2 illustrates the concepts and properties (with their domain and range) in a more intuitive way using the OntoGraf plug-in, which can be found in Protégé. In this representation, the ontology is viewed as a graph, where the nodes are concepts and the edges represent object properties relating the concepts. This section presented how we model the concepts and properties of our HTN planning ontology using OWL. Next section shows an instantiation (ABox) of this previously explained ontology conceptualisation (TBox) to model a specific scenario. Then, we show the equivalent agent programming plans in AgentSpeak [3] and planning domain specifications in SHOP [4] derived from our ontology representation.

5 Instantiating the Planning Ontology

To investigate the feasibility of defining a planning domain as an instantiation of our OWL ontology, we also used the Protégé ontology editor to create a simple definition of a planning problem domain scenario. We modelled a well-known multi-agent scenario known as gold miners [3], where agents playing the role of miners have to move in an environment, and search specific positions. Our scenario includes only one instance of the Operator concept (named move) and one instance of Method (named pursuitPosition). The operator move has two preconditions, one negative effect and one positive effect, all represented as predicates. The method pursuitPosition has two different flows, each one with its corresponding preconditions and effects. A snapshot of the instantiation using this scenario (gold miners) can be seen in Figure 3. It is important to highlight that Figure 3 illustrates the ontology instantiation in Protégé that corresponds exactly to the

2 http://protegewiki.stanford.edu/wiki/OntoGraf
previously explained specification. Next we demonstrate that it is possible to convert from our ontology formalism both to planning specifications and to agent plans. In fact, this paper explains methods for converting among these different formalisms.

Fig. 3. Instantiating our planning ontology according to the goldminers specific planning domain

An advantage of using ontology editors is the capability of enhancing the graphic visualisation of planning problem domains instances as well as agent plans and their relationships, as illustrates Figure 4. This visualisation was obtained using a Protégé plug-in known as OntoGraf. However, it is possible to explore the ontologies using different approaches and editors. Moreover, an ontology representation makes possible to
explore features such as rules in SWRL and inferences empowered by semantic reasoners. Next sections show how to convert from our planning ontology in OWL both to agent programming plans in AgentSpeak [3] and to artificial intelligence planners specifications in SHOP [4].

Fig. 4. Visualising the instances of our planning ontology in Protégé (OntoGraf plug-in)

5.1 Converting from our OWL Planning Ontology to AgentSpeak Plans

Most techniques for Multi-Agent System development are heavily inspired by the BDI architecture (Beliefs, Desires and Intentions). For example, the AgentSpeak [20] language was introduced in 1996 as a formalisation of BDI agents to enable agent programs to be written using a notation similar to (guarded) horn clauses. Agents achieve their goals through the use of plans that can be composed of sub-plans and that are ultimately converted into actions. This approach is similar to the one used in the HTN planning formalism, where methods are decomposed into operators. A plan body coded in AgentSpeak [3] is typically a sequence of actions to be executed and further goals to be achieved. AgentSpeak plans have three distinct parts [3]: the \textit{triggering event}, the \textit{context}, and the \textit{body}. Together, the \textit{triggering event} and the \textit{context} are called the head of the plan. The three plan parts are syntactically separated by ‘:’ and ‘<’ as follows:

\begin{verbatim}
triggering_event : context <- body.
\end{verbatim}
The following code (miner.asl) corresponds to a plan in AgentSpeak generated from our planning ontology instantiation. The scenario is the gold miners previously explained, and this example respects the presented AgentSpeak plan syntax [3]. Every instance of the Operator concept is mapped to an agent plan: its name becomes the triggering event, its preconditions form the context and its effects becomes the body. Similarly, each instance of Method is also translated to an AgentSpeak plan, with its corresponding preconditions and decomposition scheme. Both the operators and methods maintain their parameters when being converted from the ontology to agent code.

Our gold miners scenario instantiated in the ontology generates the miner.asl code depicted below. It can be noted that the move Operator becomes a plan with the triggering event +!move(Agent, From, To). The context of this plan is composed of a conjunction of two beliefs: at(Agent, From) and next(From, To). The body of this plan is to execute the external action move(Agent, From, To) in the environment, to remove the belief at(Agent, From), and to add the belief at(Agent, To). Similarly, our scenario depicts how a Method in our ontology is converted to an AgentSpeak plan. The main difference from the Operator previously explained is that the plan body is composed of goals to be achieved by the agent.

```plaintext
miner.asl (AgentSpeak code generated from our planning ontology)

1 +move(Agent, From, To) :
2   at(Agent, From) & next(From, To) <-
3     move(Agent, From, To);
4   -at(Agent, From);
5   +at(Agent, To).
6
7 +!pursuitPosition(Agent, From, To) :
8   at(Agent, From) & next(From, To) <-
9     !move(Agent, From, To).
10
11 +!pursuitPosition(Agent, From, To) :
12   at(Agent, From) & next(From, X) <-
13     !move(Agent, From, X);
14     !pursuitPosition(Agent, X, To).
```

5.2 Converting from our OWL Planning Ontology to SHOP Domain Definitions

SHOP is a HTN planning system based on ordered task decomposition whose syntax and semantics are given in [4]. In other words, SHOP is a HTN-planner implementation which enables domain-independent automated planning. We briefly highlight SHOP syntax in the code below to facilitate the understanding of how an instantiation can be converted from our ontology to SHOP specifications. Similarly to our ontology, the SHOP formalism is composed of operators and methods, which can contain preconditions and effects. In HTN planning, the objective is create a plan to perform a set of tasks (abstract representations of things that need to be done), starting with an initial
state-of-the-world. HTN planning is done by problem reduction: planners recursively decompose tasks into subtasks until they reach primitive tasks that can be performed directly by planning operators. A set of methods is required in order to tell the planner how to decompose nonprimitive tasks into subtasks, where each method is a schema for decomposing a particular kind of task into a set of subtasks (provided that the preconditions are satisfied).

Syntax of SHOP Planning Domain Definitions

```
(defdomain domain_name (  
  (:operator (operator_name ?parameters)  
  (preconditions ?parameters))  
  (negative_effects ?parameters))  
  (positive_effects ?parameters)))

(:method (method_name ?parameters)  
  (preconditions ?parameters))
  (method_or_operator ?parameters))
```

The following code illustrates the corresponding SHOP domain definition (named gold miners) which corresponds to the previous explained scenario instantiated in our ontology as example. We can observe that the instances of Operator and Method (and its corresponding relationships) are converted in the generated miner.jshop specification depicted below. More details about the algorithms to convert from our planning ontology to the SHOP planning domain specifications (and vice-versa) can be found in the next section of this paper.

```
(defdomain goldminers (  
  (:operator (!move ?agent ?from ?to)  
  (at ?agent ?from) (next ?from ?to))  
  (at ?agent ?from)  
  (at ?agent ?to))  

(:method (pursuitPosition ?agent ?from ?to)  
  (at ?agent ?from) (next ?from ?to)  
  (!move ?agent ?from ?to)))

(:method (pursuitPosition ?agent ?from ?to)  
  (at ?agent ?from) (next ?from ?x)  
```

miner.jshop (SHOP code generated from our planning ontology)
6 Planning and Ontology Conversions

This section demonstrates, in a high level of abstraction, the algorithms implemented in Java to convert OWL ontologies to SHOP specification files, and vice-versa, which is from SHOP domain definitions to the corresponding OWL ontology instances. Thus, we established a bidirectional mapping among the elements of our OWL planning ontology and the elements represented in the SHOP domain specifications.

6.1 Converting from the OWL Ontology to SHOP

The OWL API [5] was used to read the ontology in Java and write the corresponding jshop file. OWL API is an open source Java API (Application Programming Interface) for creating, manipulating and serialising OWL ontologies. The instances, concepts, properties and annotations in the ontology previously presented are queried and the corresponding SHOP component is generated to that specific ontology element to construct the corresponding jshop file. The algorithm for converting the OWL to a jshop file is the following:

```
for each instance df of DomainDefinition concept do
    create the jshop corresponding file
    operators ← has-operator relationships of df
    for each Operator op in operators do
        extract op information from the ontology
        write op parameters, conditions and effects in order
    end for
    methods ← has-method relationships of df
    for each Method met in methods do
        extract met information from the ontology
        write met parameters and flows in order
    end for
end for
```

6.2 Converting from SHOP to the OWL Ontology

The OWL API [5] was also used to write the ontology, after implementing a parser in Java to read and interpret the jshop file. This approach makes the opposite direction from the previous one, which converted from the OWL planning ontology to an SHOP specification. In this algorithm, for each component found when parsing the jshop file, such as a new operator, method or axiom, then the equivalent OWL individual is created with the OWL API and included in the ontology instantiation being created (which can be instances, object properties, data properties or annotations). The algorithm to convert a jshop file to a corresponding instantiation of our OWL planning ontology is the following:

```
for each instance df of DomainDefinition concept do
    create the jshop corresponding file
    operators ← has-operator relationships of df
    for each Operator op in operators do
        extract op information from the ontology
        write op parameters, conditions and effects in order
    end for
    methods ← has-method relationships of df
    for each Method met in methods do
        extract met information from the ontology
        write met parameters and flows in order
    end for
end for
```
while there are tokens remaining in the jshop file do
    token ← nextToken()
    if token = defdomain then
        create corresponding DomainDefinition instance
    end if
    if token = operator then
        create corresponding Operator instance
        read its parameters, preconditions and effects
        create the corresponding ontology elements
    end if
    if token = method then
        create corresponding Method instance
        read its parameters and flows
        create the corresponding ontology elements
    end if
end while

Previous section demonstrated how one example is converted from our ontology both to SHOP specifications and AgentSpeak code. This section shows the algorithms to convert both from the ontology to SHOP domain, and vice-versa, which are already implemented. However, the algorithms to convert between ontology and agent plans are currently being developed, but we already exemplified how this conversion can be made in this paper.

7 Final Remarks

We presented an investigation towards the integration of agent-oriented programming and automated planning with semantic technologies. More specifically, this paper proposed an ontology to represent planning formalisms. Our ontology was developed in OWL [1] to represent HTN [2] domains and problems in the context of automated planning and agent-oriented programming. The proposed ontology was instantiated to exemplify its use and to demonstrate its feasibility. Also, we presented algorithms to convert specifications between different formalisms such as OWL [1] and SHOP [4]. The algorithms have been coded in Java using the OWL API [5].

Given the similarities among planning formalisms and agent programming plans, we also explored how to generate a corresponding AgentSpeak [3] code, which is a logical language to program agent plans. As examples of relations between concepts in these two formalisms we can currently highlight: method & plan; precondition & context; and operator & external action. Thus, we also explored how to convert from our OWL [1] planning ontology to AgentSpeak [3] plans, and vice-versa. In other words, our approach enables new ways to derive both planning specifications and agent code.

As pointed out in [17], the use of OWL ontologies as a basis for modelling domains allows the reuse of knowledge in the semantic web. However, research in this direction is still in their initial steps. We have briefly discussed the state of the art of approaches
that integrate ontologies with planning and agent-oriented programming, commenting on their findings and contributions.

As future work, we plan to investigate ontology reasoning mechanisms and semantic technologies features within the scope of our planning ontology. One example would be creating rules (e.g., in SWRL [12]) to infer knowledge such as inconsistencies in ontology instantiations. The ability to use ontologies to infer and generate knowledge over a domain is a motivation to investigate how ontology representations can be integrated with planning and agent-oriented programming. Thus, as next step in this direction, we will explore advantages of using the semantic reasoning enabled by ontologies.

Another interesting area to explore is extending the planning ontology to address further planning characteristics, such as non-deterministic HTN planning formalisms. However, if the conceptualisation changes, the parsers may have to be adjusted accordingly to handle new concepts and properties in the ontology. Currently, we plan to continue assessing the correctness of our algorithms (for converting between OWL [1] to SHOP [4]) by testing them with more examples. Moreover, we are currently coding the algorithms to convert between the ontology and AgentSpeak [3].

This work investigated the conversion from OWL [1] ontologies to both SHOP [4] and AgentSpeak [3], but we emphasise that different planning systems and agent programming languages could also be explored. The inclusion of ontology-based semantic technologies in such complex multi-agent platforms is expected to bring together the power of knowledge-rich approaches and complex distributed systems.

Acknowledgements

Part of the results presented in this paper were obtained through research on a project titled “Semantic and Multi-Agent Technologies for Group Interaction”, sponsored by Samsung Eletrônica da Amazônia Ltda. under the terms of Brazilian federal law No. 8,248/91.

References

Abstract. Normative systems offer a means to govern agent behaviour in dynamic open environments. Under the governance, agents themselves must be able to reason about compliance with state- or event-based norms (or both) depending upon the formalism used. This paper describes how norm awareness enables a BDI agent to exhibit norm compliant behaviour at run-time taking into account normative factors. To this end, we propose N-Jason, a run-time norm compliant BDI agent framework supporting norm-aware deliberation as well as run-time norm execution mechanism, through which new unknown norms are recognised and bring about the triggering of plans. To be able to process a norm such as an obligation, the agent architecture must be able to deal with deadlines and priorities, and choose between plans triggered by a particular norm. Consequently, we extend the syntax and the scheduling algorithm of AgentSpeak(RT) to operate in the context of Jason/AgentSpeak(L) and provide ‘real-time agency’, which we explain through a detailed examination of the operational semantics of a single reasoning cycle.

Keywords: Norms, BDI, Agent Programming Language, Normative System

1 Introduction

In conventional BDI agents, norm compliance is typically achieved by design. That is by specifying plans that are triggered by detached norms, because the agent programmer knows which norms the agent will adopt, and then prioritising those rules so that those supporting norms are chosen over those preferred by the agent’s mental attitudes, in order to suppress conflicts between the normative and the agent’s existing goals. This creates an undesirable dependence between the agent implementation and the norm implementation, which creates two issues:

1. When an agent encounters new and unknown norms, which were not taken into account at design time, there is typically no plan to deal with those norms in the
plan library at run-time. Hence, norm compliant behaviour cannot normally be exhibited because the norms are unavoidably ignored. Yet worse, agents may suffer a punishment from the enforcement of the normative system as a result of a violation caused by their incapacity to process the normative event.

2. The hierarchical prioritisation of normative over ordinary plans deprives an agent of its autonomy, since the norms in effect are treated as hard constraints, whose violation is not possible.

We believe that such tensions can be resolved by the use of an extended model of norm awareness. In the literature on BDI agents, norm awareness, which is a precursor to norm compliance, is exhibited broadly by two approaches: (i) at the perception level, by taking new unknown norms into account as part of the generic execution mechanism [13, 14] and (ii) at the deliberation level, by attempts to resolve the conflict between normative factors and agents’ mental attitudes [1, 9]. We propose to coalesce these approaches into one ‘sense–think–act’ reasoning cycle informed by the concept of awareness, which Charlton [4] describes as the capacity “to select and integrate relevant inputs from a complex environment to enable humans or animals to choose between a large repertoire of behavioural responses”. This definition reminds us that, in order to be norm aware, agents should have knowledge (or understanding) about norms in respect of: (i) what (state) the norms intend to reach or to achieve, (ii) which action plans are appropriate to execute norms and (iii) which behaviour agents should prefer between normative goals and the agent’s own interests.

Thus, this paper addresses the convergence of these approaches in the context of the BDI agent architecture, in order to be able to ground the discussion of how the extended model of norm awareness enables a BDI agent to exhibit norm compliant behaviour at run-time. To do so, we propose N-Jason, a run-time norm-compliant BDI agent framework supporting a run-time norm execution mechanism, under which new and unknown norms are recognised and enable the triggering of an appropriate plan (if present), in conjunction with norm-aware deliberation [1]. To be able to process a norm such as an obligation, the agent architecture should be able to deal with deadlines and priorities, and choose between plans triggered by a particular norm. Consequently, we extend the syntax and the scheduling algorithm of AgentSpeak(RT) [15] to operate in the context of Jason/AgentSpeak(L) [3] and provide ‘real-time agency’, which we explain through a detailed examination of the operational semantics of a single reasoning cycle.

The paper is organised as follows. In §2 an institutional framework and semantics of norms considered in N-Jason are introduced. It is followed by §3, where we present a run-time norm compliant BDI agent framework including programming language and interpreter. After the operational semantics in §4, related work and the contribution of this work are contrasted in §5. The conclusion and future work are discussed in §6.

2 Institutional Framework

Normative frameworks can be viewed as a kind of external repositories of (normative) knowledge from which (normative) guidance may be delivered to agents. Usually, a normative framework is composed of a set of rules whose purpose is to model the normative
positions established by the actions of agents and hence realise the governance of individual agents in the society. These rules are not hard-coded recipes presenting reactive behaviours, such as those in the static expert systems, but rather describe consequences arising from observations for the purpose of reasoning about the current context, resulting in situation-specific norms. The framework identifies not only correct and incorrect actions but also norms such as obligations, permissions and prohibitions through the institutional trace that records its evolving internal state, subject to observed external events representing actions in the external world.

Depending on the formalism of the normative system, norms can be categorised as state- or event-based. State-based norms usually express higher level norms that impose desirable or required states on the system (or an environment), often as a logical combination of institutional facts, which should be brought about by the actions of agents [8]. In contrast, event-based norms generally represent relatively lower level activities addressing possibly executable events (or actions) at the individual agent level [7]. In this paper, we use Cliffe’s institutional model [5] for the purpose of providing detached event-based norms, upon which we develop the run-time norm compliance model presented here.

The institutional framework provides a formal action language InstAL to specify norms, describing coordinations and interactions between agents and (or) environments in the context of an institution. The normative specification is translated to a computational model that utilises Answer Set Programming (ASP) [10], which enables reasoning about the current context described in the institution. The institution is composed of a set of institutional states, evolving over time triggered by the occurrence of both internal and external events. An institutional state is a set of fluents which are present (denoting true) or absent (denoting false) at a given time instant. In addition, such institutional fluents are divided into domain fluents and normative fluents which latter is made up of: (i) power ($W$) – indicates events that are empowered to bring about institutional change (ii) permission ($P$) – indicates events that can occur without violation, and (iii) obligations ($O$) – specifies events that are obliged to happen before the occurrence of a deadline (e.g. a timeout), or else a violation occurs.

These normative fluents represent the normative consequences of particular behaviours which should be achieved by agents in a certain context. For example, if an agent $X$ is obliged to carry out an action $act$ by deadline $deadline$ otherwise the violation event $violation$ is generated, the form of the normative information is represented as:

$$obl(act, deadline, violation)$$

(obligation)

Also if an agent $X$ is permitted to perform an action $act$, then the representation is:

$$perm(act)$$

(permission)

The determination of those normative consequences is carried out using an answer set solver driven by a rule-based specification (InstAL) which explores all possible outcomes derivable from the institutional state arising from the occurrence of a single
as determined by the generation and consequence rules that comprise the institutional model. Lee et al. [12] demonstrate a governance mechanism using this institutional model that shows how the normative consequences of particular actions can be delivered to agents’ minds as percepts (to conventional Jason agents rather than the variety described here) either on request or by subscription, making them available for the agent reasoning process.

Van Riemsdijk et al. suggest in [14], that one feasible approach for run-time norm execution is the use of “pre-existing capabilities” in the agent program when an agent encounters new and unknown norms. This assumes that event-based norms can identify the associated necessary actions, since event-based norms typically refer to relatively low-level activities that address possibly executable events (or actions) at the individual agent level [7]. If appropriate information can be extracted from the detached norm, such that it is recognisable to an agent, an agent presumably may execute unknown norms and so exhibit norm compliance at run-time. For example, the act term in an obligation represents a similar level of knowledge to plans or events in a BDI agent program. If an agent can retrieve and recognise what action (or event) is required to be achieved, then it can trigger certain plans and attempt to carry out such behaviour even though the norm is not handled explicitly in the agent specification. With regard to the norm-aware reasoning, an agent may deduce a preference, if it is able to know the relative priorities, and critical impact or the deadline of normative factors by extracting deadline and violation information. This norm-aware reasoning may allow an agent to pursue its own preferences between its own goals, norms and sanctions by measuring feasibility, as proposed by Alechina et al. [1]. In this paper, we only use obligations for such purpose, in order to focus on the essential aspects of the agent’s internal reasoning process. Additionally, we consider the handling of prohibitions for the compatibility with other normative systems, however they are not explicit in the institution mechanism employed here.

3 The N-Jason BDI Agent Framework

In this section we outline N-Jason, a norm aware BDI agent interpreter and its programming language for run-time norm compliant agent behaviour. In principle, it extends Jason/AgentSpeak(L) syntactically, semantically and in the reasoning process of the interpreter. In practice, N-Jason is conceptually similar to AgentSpeak(RT) [15], which is capable of dealing with deadlines and priorities and scheduling intentions with the aim of providing real-time agency. N-Jason is conceptually a superset of AgentSpeak(RT), to which it adds normative concepts (i.e. obligations, permissions, prohibitions, deadlines, priorities and durations) and norm aware deliberation.

We firstly examine work to date with regards to the programming language aspect. This is followed by an informal explanation of the N-Jason reasoning cycle. Subsequently, we show how the extended model of norm awareness in BDI agents is estab-

3 Note: the institutional model can also function as a normative oracle for an agent, if presented with a sequence of events, in which case it derives all the possible outcomes from all possible orderings of those events, subject to whatever constraints are specified on the ordering.
lished by the combination of the run-time norm execution mechanism and norm-aware deliberation.

3.1 The N-Jason Agent Programming Language

A N-Jason agent consists of four main components: beliefs, goals, events and a set of plans. Beliefs and goals are identical to those in standard Jason, while events and plans are extended. We now give a brief summary of the extended features of the basic elements in the agent specification.

Belief: A belief represents knowledge about the environments wherein agents are situated. It is composed of percepts observed by agents, messages containing the information about other agents, and norms delivered from normative frameworks. Typically, a belief is represented as a grounded atomic formula. The collection of beliefs is referred to as a belief base, which contains belief literals decomposed into belief atoms and negations.

Goal: A goal is one of two basic types: an achievement goal and a test goal. The former are usually specified as predicates prefixed by the ‘!’ operator. This specifies a certain state of the environment that the agent wants to achieve, which is indicated when the predicate associated with its achievement goal is true. The latter test goal, for which the prefix is the ‘?’ operator, indicates that agents want to know whether the associated predicate is a true belief.

Event: An event is the main component for triggering agent’s plans. In principle, changes in agent’s mental attitudes (i.e. beliefs, goals and intentions) give rise to events. There are two types of events: one is an addition event denoted by ‘+’, which means the addition of a belief or an achievement goal. The other is a deletion event denoted by ‘–’, referring to a recantation of a base belief. As in Jason, an addition event is categorised by a belief addition event denoted by ‘+’ and a goal addition event jointly denoted by ‘+’ and ‘!’.

Support for normative concepts is provided by an extension of the syntax for an event by the addition of deadline and priority information. The deadline is a real time value indicating a deadline by which an intention should be achieved. It is expressed in some adequate unit of real world time. When the deadline is passed, it is no longer feasible to achieve an intention or to give a response with a belief change. The priority is a positive integer value that expresses the relative importance between the achievement of an intention and responding to changes in a belief. A larger value reflects a higher priority. Both can be specified optionally in the annotation (a list of terms in between square brackets “[” and “]”) at the end of an event. For example the event:

+!at(X, Y)[deadline(900), priority(10)]
specifies the goal adoption that an agent moves to the coordinate (X, Y), by the
deadline 900, with priority 10. The deadline is taken as infinity and the priority as
zero, by default. Note that those annotated deadline and priority are not involved in
an unification at plan selection stage.

Plan: A plan is a sequence of actions (and subgoals) which is a means to achieve a
(main) goal or a means to respond with changes in beliefs by agents. The plan typ-
ically consists of a head and a body, but sometimes an optional plan label, which
defines an index, a name and other information, can be specified. The head is com-
piled of a triggering event, which specifies an event for which the plan is to be
used and a context specifying the condition which must be true for the plan to be a
candidate for execution. The body is a series of actions and subgoals to achieve a
main goal.

The plan is extended to support normative concepts. Given the three main elements,
a duration is proposed in N-Jason, specifically in order to enable assessment of
the feasibility of the plan associated with the deadline (see §3.4). The duration is
a non-negative integer value representing a required time to execute the plan. In
principle, the duration may be determined by the summation of an execution time
of each external action in the plan body. For simplicity, we follow the assumption
described in [1], that the estimated time for each external action is fixed and already
known. Like deadline and priority, a duration can be optionally specified in the plan
label in the form of an annotation (a list of terms in between square brackets “[”
and “]”). For example, the plan:

```plaintext
@plan[duration(50)]
+!at(X, Y) : req(ag)
<- move_toward(X, Y); !ack(ag).
```
is triggered by the request from the agent ag to move to the coordinate (X, Y), and
then to send back an acknowledgement to ag. The required (or estimated) execution
time of the plan is 50.

3.2 The N-Jason Interpreter

The interpreter plays an important role in the operationalisation of agent programs.
The agent’s belief base, intentions and events are manipulated by the interpreter, and
practical reasoning consisting of deliberation and means-ends reasoning is performed
to achieve a goal or to respond to environmental changes.

During a single reasoning cycle, run-time norm compliance is accomplished by an
extended model of norm awareness that has three steps:

1. **Event Reconsideration**, to find out what the norm is intended to achieve or to reach,
2. **Option Reconsideration**, to identify which plan is the most appropriate in response
to the norm and
3. **Intention Scheduling** to confirm the decision about which behaviour agent would
prefer between goals, norms and sanctions.

We now give an informal explanation of one reasoning cycle in the interpreter (see
Algorithm 1). At the beginning (lines 1–4), an agent perceives some knowledge about
Algorithm 1 \textit{N-Jason} Interpreter Reasoning Cycle

1: $B := B_0$ /* $B_0$ are initial beliefs */
2: $G := G_0$ /* $G_0$ are initial goals */
3: $E := E \cup \text{goal-events}(G) \cup \text{belief-events}(B)$
4: $P := P \cup N$
5: for all $p \in P$ and $p \notin P_0$ and $P_0 \subset B$ do
6:   $te_p := \text{create-tevent}(p)$
7:   $\pi\theta := \text{SR}(te_p)$ where $\theta$ is a mgu for $te_p$ and plan $\pi$
8:   if $\pi\theta \neq \emptyset$ then
9:     $E := \text{add-event}(E, te_p)$
10:    else if $\pi\theta = \emptyset$ and type($p$) = (obl | prob) then
11:       $E := \text{EVT-RECONSIDERATION}(p)$
12:    end if
13:  $B := \text{update-belief}(B, p)$
14: end for
15: for all $(te, \tau) \in E$ do
16:   $\pi\theta' := S_o(te)$ where $\theta'$ is a context unifier for $te$ and plan $\pi$
17:   if $\pi\theta' = \emptyset$ then
18:     $\pi\theta' := \text{OPT-RECONSIDERATION}(te)$
19:   end if
20:   if $\pi\theta' \neq \emptyset$ and $\tau \notin I$ then
21:     $I := I \cup \pi\theta'$
22:   else if $\pi\theta' \neq \emptyset$ and $\tau \in I$ then
23:     $I := (I \setminus \tau) \cup \text{push}(\pi\theta' \sigma, \tau)$ where $\sigma$ is an mgu for $\pi\theta'$ and $\tau$
24:   else if $\pi\theta' = \emptyset$ and $\tau \in I$ then
25:     $I := (I \setminus \tau)$
26: end if
27: $I := \text{SCHEDULE}(I)$
28: if $I \neq \emptyset$ then
29:   $I := \text{EXECUTE}(I)$
30: end if
31: end for

an environment ($P$) as well as a list of norms ($N$) (e.g. obligations) delivered from one or more institutional frameworks. Although those are separated entities at the agent level, it is unified into a set of percepts at the interpreter level (line 4). Thus, the interpreter may not be able to distinguish norms from percepts at this stage.

The belief base ($B$) is updated by $P$ in the belief update process (belief-update-function (buf) more precisely) (see lines 6–13). This belief update involves the creation of events in response to each new percept. Once a percept ($p$) is encoded as a triggering event ($te_p$), the interpreter checks whether $te_p$ has relevant plans in the plan library $\Pi$ using the plan selection function $S_R$ (for more details about $S_R$, see [3]). If relevant plans are retrieved, then the event base ($E$) is updated with $te_p$. If no relevant plan is retrieved, $te_p$ is ignored but $B$ is updated in any case with $p$. The same approach is taken for norms when the norms and its relevant plans are already specified in the agent program. Otherwise, the event reconsideration process (line 11) starts to find out what the norms are intended to achieve, as the first step in run-time norm execution.
Next, the interpreter starts the reasoning process in order to determine an applicable plan in response to the $te$ selected by the event selection function $S_E$. The selection function $S_O$ chooses a single option from the applicable plans as a result of the unification of event and context. If $S_O$ retrieves nothing, then the interpreter follows exactly the same path as described above. The option reconsideration process (line 18) tries to find out which action plans are appropriate to execute unknown norms, as the second step in run-time norm execution. See lines 16–19.

If one single applicable plan is successfully retrieved by $S_O$, then the means-ends reasoning adds the applicable plan ($\pi$) as an intended means ($IM$) on top of an intention ($I$). If $te$ of $\pi$ is an internal event then $\pi$ added in the existing $I$, otherwise a new $I$ is created with $\pi$ to be added in there (line 20–26). This is followed by the intention scheduling process which returns a preference maximal set of intentions in deadline order (line 27). Afterwards, one intention selected by the intention selection function $S_I$ is finally executed (line 29). The details of the remainder are exactly the same as in [3] or [15].

The internal operation of the N-Jason interpreter is shown in Algorithm 1 extended from [15]. We use the same notations as in [15] for consistency and comparability. The functions EVT- and OPT-RECONSIDERATION accomplish the run-time norm execution mechanism described in §3.3. The function edp constructs an event using the terms in the event-based norm, if the type of a percept $p$ is a norm (e.g. obligations). The main algorithm of the SCHEDULE function is shown in §3.4.

### 3.3 Run-Time Norm Execution

In §3.2, we explained that run-time norm execution is realised by two steps: (i) event reconsideration and (ii) option reconsideration. Prior to defining those reconsideration processes, we firstly define a property of the executability of norms at run-time. We say that a norm such as $obl(evt, deadline, violation)$, is executable at run-time iff:

1. $p \in P$ and $type(p) = (obligation \mid prohibition)$, where $p$ is a percept, formed from a list of terms such as $term("\"\ term")$, in a set of newly observed percepts $P$ at run-time;
2. $te_p \notin E$, where $te_p$ is a triggering event generated from the percept $p$, and $E$ is an event base, which is a set of events $\{(te, \tau), (te', \tau'), \ldots\}$, where an event is a pair of a triggering event and an intention ($te, \tau$);
3. $edp(p) \neq \emptyset$ and $\{te_{edp(p)}\} \cap E \neq \emptyset$, where $edp(p)$ is a function extracting the obliged event together with its deadline and priority from $p$ and $te_{edp(p)}$ is a triggering event of the $edp(p)$, an event term in the norm, and
4. $R_{te_{edp(p)}} \neq \emptyset$, where $R_{te_{edp(p)}}$ is a relevant plans selection function.

The executability determines the necessity of further reconsideration for the new and unknown norms. If those norms are judged executable at the perception stage, the event-reconsideration process starts for the addition of such norms to the event base as triggering events. Similarly, the executability also enables the option-reconsideration in order to execute an applicable plan in relation to the triggering events derived from the norms.

330
Algorithm 2 Event Reconsideration

Require: $P := P \cup N$
Require: $te_{ep} = create-tevent(p)$
1: if $p \in P$ and type$(p) = obligation$ then
2: $te_{edp}(p) = create-tevent(edp(p))$
3: $\pi\theta := S_R(te_{edp}(p))$ where $\theta$ is a mgu for $te_{edp}(p)$ and plan $\pi$
4: if $\pi\theta \neq \emptyset$ then
5: $E := add-event(E, te_{p})$
6: end if
7: else if $p \in P$ and type$(p) = prohibition$ then
8: $\Xi := add-prohibition(\Xi, edp(p))$
9: end if

Event Reconsideration aims to verify that a norm perceived at run-time is executable although no corresponding plan exists in the agent program. If an event extracted from a detached norm has a relevance to a certain set of plans, it thus has potential to trigger specific ones, and it is then concluded that the norm is executable. If the norm is proven to be executable, the interpreter adds the norm to the event base $E$ as an achievement goal addition event. The procedure for event reconsideration is as follows (see Algorithm 2):

1. Extract the terms representing an obliged event, a deadline and its priority\(^4\) from the obligation by the function $edp$, whose practical implementation may vary, depending on norm representations in various systems (line 2),
2. Construct a new triggering event (an achievement goal addition event in this case) from the combination of extracted terms (line 2),
3. Query the existence of a set of relevant plans to $S_R$ with such a constructed triggering event (line 3),
4. Add such triggering event to $E$, if relevant plans are successfully retrieved (line 5) and
5. If the norm is a prohibition, then the extracted event is added into the prohibition base ($\Xi$) (line 7 - 8) and will be revisited at the norm deliberation stage\(^5\).

For example, suppose there is a detached obligation $obl(at(X, Y)$, $1030), 10)$. If relevant plans are not found in the agent program (plan library of an agent, to be precise) in response to the obligation, the function $edp$ firstly extracts the event ($at(X, Y)$), deadline ($1030$) and priority ($10$) from the obligation. Next, the interpreter constructs a new triggering event (an achievement goal addition event as described above) such as $+!at(X,Y)[\text{deadline}(1030), \text{priority}(10)]$ using the extracted information. Subsequently, the interpreter queries the existence of relevant plans to $S_R$ once again with a new triggering event, $+!at(X,Y)[\text{deadline}(1030), \text{priority}(10)]$.

\(^4\) In principle, the last term is an event which arises when a violation occurs. This value normally indicates the criticality of such a violation. Higher values represents a higher priority.
\(^5\) N-Jason supports prohibitions as described above, and is therefore compatible with normative systems supporting prohibitions, but we note that the institutional model described in §2 does not have an explicit representation of prohibition, but only the absence of permission.
If the retrieval of relevant plans is successful, then the original event, `+obl(at(X, Y), 1030, 10)`, is added to `E`.

One exceptional aspect in event-reconsideration is the addition of a deontic event `te_p` (which is a detached norm) instead of a normal event `tedp(p)` (which is a newly constructed triggering event) into the event base `E`. In so doing, we intend to distinguish norm-triggered intentions from ordinary intentions that normal events trigger, so as to facilitate norm-aware deliberation (see §3.4) in `N-Jason`. In principle, `Jason` creates different intentions in response to different triggering events. Given this characteristic, both a deontic and a normal event create a deontic and a normal intention in `N-Jason`, respectively. The intended means included in both intentions are identical since a deontic and a normal event trigger exactly the same plan in an agent program. However, the properties (e.g. deadline and priority) of each intention are different. The normal intention follows the original deadline and priority specified in the plan. In contrast, the deontic intention has different deadline and priority, which are inherited from those in the detached norm. As a result, these intentions are the main source of norm-aware deliberation. An agent is able to deliberate on norms and agent’s private goals through the evaluation of the relative importance and urgency using norm-triggered (i.e. deontic) intentions and ordinary event-triggered (i.e. normal) intentions.

Suppose a plan whose label is `example`, is specified in an agent program:

```
@example[duration(50)]
+!at(X, Y)[deadline(1000), priority(5)]
<- move_toward(X, Y); !ack(ag).
```

Assuming that a normal event triggering `example` is added to event base `E`. Then it creates a normal intention using a pair of normal event and its associated plan `plan_example`, whose deadline and priority are 1000 and 5, respectively. Later, a detached obligation `obl(at(X, Y), 1030, 10)` is received. Following Algorithm 2, the deontic event `+obl(at(X, Y), 1030, 10)` is added to `E`, since a relevant plan `example` is found. Consequently a deontic intention is created using a pair of a deontic event and its associated plan `example`. Its deadline and priority are 1030 and 10, respectively, which are different from those in the normal intention. Obviously, we have two intentions whose properties are different, although the intended means are absolutely same. Hence, `N-Jason` is able to carry out norm-aware deliberation on norms and the agent’s own goals using norm-triggered (i.e. deontic) intentions and ordinary event-triggered (i.e. normal) intentions.

`Option-Reconsideration` is a central element in the practical reasoning process whereas the event reconsideration happens at the perception stage. The main objective of option reconsideration is the determination of an applicable plan corresponding to the new and unknown norm – whose executability is already verified – and is thus added to `E` as an achievement goal addition event. If the applicable plan is chosen, then it will probably be used to enact a norm-compliant behaviour, unless it is infeasible as judged by intention scheduling (described in §3.4). The procedure is shown in Algorithm 3.

Like `Event-Reconsideration`, `te_p` is generated by a new and unknown norm that does not have any relevant plans `R_{te_p}` at this moment. Thus at the beginning of the option reconsideration, the interpreter carries out the same process for event reconsideration:
### Algorithm 3 Option Reconsideration

**Require:** \( \langle te_p, \tau \rangle \in E \) where \( te_p \) is an event and \( \tau \) is an intention

**Ensure:** \( \pi \theta' \theta \) where \( \theta' \) is a context unifier for \( te_{edp}(p) \) and plan \( \pi \)

1. if \( \text{type}(p) = \text{obligation} \) then
   2. \( te_{edp}(p) = \text{create-tevent}(edp(p)) \)
   3. \( \pi \theta := S_R(te_{edp}(p)) \) where \( \theta \) is a mgu for \( te_{edp}(p) \) and plan \( \pi \)
   4. if \( \pi \theta \neq \emptyset \) then
      5. \( \pi \theta' := S_o(te_{edp}(p)) \) where \( \theta' \) is a context unifier for \( te_{edp}(p) \) and plan \( \pi \)
   6. end if
5. end if

1. Extract the event term \( edp(p) \) of the norm in order to retrieve relevant plans \( R_{te_{edp}(p)} \) (as before), if the type of \( p \) is a norm (i.e. an obligation) (line 1 - 2),
2. Retrieve the relevant plans corresponding to the \( R_{te_{edp}(p)} \) by the unification of an atomic-formula in a triggering event and each plan in an agent (line 3),
3. Determine a set of applicable plans through the extended unification of a triggering event, a plan and a context and (line 5) and
4. Select a single applicable plan as an intended means to which to commit (line 5).

### 3.4 Norm Awareness in Deliberation

Norm awareness in the deliberation process is achieved by the scheduling of intentions with deadlines and priorities. We extend the algorithm proposed in [15] with the consideration of prohibitions in order to establish a conflict-free preference maximal set of intentions. In effect, this is like [1] who proposes a scheduling algorithm that brings about a preference maximal set of intentions, but that depends upon (N-)2APL’s parallel execution of plans, whereas here the scheduling algorithm for (N-)Jason has to take account of the single-threaded plan execution model in Jason.

The scheduling algorithm is introduced in Algorithm 4. A set of candidate intentions \( I_C = \{ \tau, \tau', \ldots \} \), which is sorted in descending order of a priority, is inserted into a scheduling process. If each intention is feasible, i.e. a plan on top of the intention can be executed before the deadline and is not prohibited by a set of prohibition \( \Xi = \{ \xi, \xi', \ldots \} \), then the intention is added to the preference maximal set \( \Gamma \) whose criteria are defined as follows:

1. An intention is feasible \( \text{iff} \) the execution of the intention is completed before its deadline, that is, for \( \tau \),
   \[
   ne(\tau) + et(\tau) - ex(\tau) \leq dl(\tau)
   \]
   where \( \tau \) denotes an intention, \( ne(\tau) \) is the time at which \( \tau \) will next execute, \( et(\tau) \) is the time required to execute \( \tau \), denoted in the plan label, \( ex(\tau) \) is the elapsed time to execute \( \tau \) to this point, and \( dl(\tau) \) is the deadline for \( \tau \) specified in the plan [1].
2. The intention should not be prohibited, that is, for \( \tau \neq \xi \in \Xi \) or
Algorithm 4 Scheduling of Intentions

1: \( \Gamma := \emptyset, \Xi' := \emptyset \)
2: for all \( \tau \in I \) in descending order of priority do
3:     if \( \{\tau\} \cup \Gamma \) is feasible then
4:         if \( \tau \notin \Xi \) then
5:             \( \Gamma := \{\tau\} \cup \Gamma \)
6:         else
7:             for all \( \xi \in \Xi \) do
8:                 if \( \tau = \xi \) then
9:                     \( \Xi' := \{\xi\} \cup \Xi' \)
10:             end if
11:         end for
12:         if \( \text{priority}(\tau) > \max\{\text{priority}(\xi), \forall \xi \in \Xi'\} \) then
13:             \( \Gamma := \{\tau\} \cup \Gamma \)
14:         end if
15:     end if
16: end for
17: sort \( \Gamma \) in order of increasing deadline
18: return \( \Gamma \)

\(- \tau \in \Xi, \text{then } \forall \xi \in \Xi, \tau = \xi \text{ and } \text{priority}(\tau) > \max\{\text{priority}(\xi), \forall \xi \in \Xi'\} \)

where \( \tau \) is an intention, \( \xi \) is a prohibited event in the prohibition base \( \Xi \) and \( \text{priority} \) is a priority retrieval function.

Scheduling in N-Jason is also pre-emptive in that the adoption of a new intention \( \tau \) may prevent scheduled intentions with lower priority than \( \tau \) (including currently executing intentions) being added to the new schedule just as in N-2APL and AgentSpeak(RT). Intentions that cannot meet their deadline are dropped.

3.5 Example

As an example, we consider robots serving beer in a pub, whose main role is to get an order and to deliver a beer to the customer. We assume the existence of some institutions delivering desirable social norms, subject to the observations of participants, and that all agents are governed by such systems. A part of the agent program is shown below:

```prolog
@P1[duration(5)]
+!at(X, Y) : not at(X, Y)
<- move_toward(X, Y).

@P2[duration(10)]
+!order(X, Y)
<- get(beer); move_toward(X, Y).
```

At time 100, the robot receives the following events:

E1: +!request(2, 3)[deadline(130), priority(20)] A request from customer seated at (2, 3). The deadline is 130 and the customer is important so the priority is 20.
E2: +request(1, 1)[deadline(115), priority(10)] A request from customer seated at (1, 1). The deadline is 120 and the priority is 10.
E3: +request(3, 3)[deadline(130), priority(10)] A request from customer seated at (3, 3). The deadline is 130 and the priority is 10.

These three events trigger the plan P2, and give rise to three possible intentions \( \tau_1 \) (P2 triggered by (2, 3)), \( \tau_2 \) (P2 triggered by (1, 1)), and \( \tau_3 \) (P2 triggered by (3, 3)). \( \tau_2 \) is not feasible, thus it is dropped, whereas \( \tau_1 \) and \( \tau_3 \) are feasible, so scheduled in deadline order: \( \tau_1 \) is scheduled first between 100 and 110 since it has an earlier deadline followed by \( \tau_3 \) between 110 and 120. Now the agent starts the execution of \( \tau_1 \).

Let consider an announcement of a fire alarm by one of the normative frameworks. It broadcasts an obligation containing the coordinates of an exit to all participants so they may escape from the building. Suppose the norm is \( \text{obl}(\text{at}(0, 0), 115, 100) \).

Although the obligation is not stated in the agent’s program, it is executable since the agent has a \textit{pre-existing} moving ability \( \text{at}(X, Y) \), which is enough to satisfy the obligation. With the event- and option-reconsideration, the event:

E4: +at(0, 0)[deadline(115), priority(100)] is generated from the obligation, thus adoption the plan P1, bringing about an intention \( \tau_4 \) (P1 triggered by (0, 0)). During the execution of \( \tau_1 \), \( \tau_3 \) and \( \tau_4 \) are inserted into a new schedule in deadline order: since the priority of \( \tau_4 \) is greater than \( \tau_3 \) and \( \tau_4 \) has a more urgent deadline, the agent starts to execute \( \tau_4 \), triggered by the obligation, before the execution of \( \tau_3 \).

Notwithstanding, that this example is extremely simple, it provides a useful in-principle illustration of norm-aware deliberation – as performed by intention scheduling – as well as the run-time norm execution mechanism in \textit{N-Jason}.

4 Operational Semantics

In this section, we present a theoretical foundation for the \textit{N-Jason} programming language with semantics based upon an extension of the operational semantics for \textit{Jason}/AgentSpeak(L). Given the formal semantics of \textit{Jason} we extend the transition rules which transform one extended configuration into another. To begin with, we show a configuration of individual \textit{N-Jason} agents which is almost unchanged except for norm configuration. In the following section, we describe the transition rules that give rise to a configuration change at each state in a single reasoning cycle. For consistency and comparability, we follow exactly the same notations as those in published \textit{Jason} descriptions excepting the normative aspects.

4.1 \textit{N-Jason} Configuration

The configuration of \textit{N-Jason} is a tuple \( \langle \text{ag}, C, N, T, s \rangle \) where:

\- \text{ag} is an agent program consisting of a set of beliefs \( bs \) and a set of plans \( ps \), as defined by the EBNF in [3].
An agent’s circumstance $C$ is a tuple $\langle I, E, A \rangle$, where $I$ is a set of intention $\{i, i', \ldots\}$, $E$ is a set of events $\{(te, i), (te', i'), \ldots\}$, in which event is a pair of a triggering event and an intention $(te, i)$ and $A$ is a set of actions an agent performs in the external environment.

$N$ is a tuple $\langle \Gamma, \Xi \rangle$ denoting normative consequences delivered from normative systems, where $\Gamma$ is a set of obligations $\{\gamma, \gamma', \ldots\}$ and $\Xi$ is a set of prohibition $\{\xi, \xi', \ldots\}$.

$T$ is a tuple $\langle R, A_p, i, \varepsilon, \rho \rangle$ defining a trace of provisional information required for subsequent steps within a single reasoning cycle, where $R$ is the set of relevant plans, $A_p$ the sets of applicable plans, and $i, \varepsilon$ and $\rho$ record an intention, event, and applicable plan (respectively) at a specific moment under consideration within the execution of a single reasoning cycle.

The current state $s$ within an agent’s reasoning cycle is denoted by $s \in \{\text{RcvNorm, SelEv, RelPI, ApplPI, AddIM, SchInt, SellInt, ExecInt, ClrInt}\}$.

4.2 Transition Rules

The execution of the $N$-Jason program leads the modification of the initial configuration of an agent via transition rules given below. For the sake of brevity, we do not repeat the communication semantics, since these are unaffected by the changes in relation to norms.

In general, the transition would normally start from the state ProcMsg, but we propose a preceding step RcvNorm, as described in §2. Thus, note that the initial configuration of this model is $\langle ag, C, N, T, \text{RcvNorm} \rangle$, where $ag$ is specified by the agent program and other all components are empty, and the reasoning cycle starts from RcvNorm with the transition rules given below.

Receiving detached norms: As described in §2, institutional frameworks may distribute norms via broadcasting when a norm is activated by the fulfillment of institutional states triggered by external events in the environment. As soon as the event-based norms are received, the norms effectively act like an ordinary event thus trigger the transition of the agent’s mental state. Rule 4.2 updates the agent belief base and an event base component $C_E$ associated with adding new norms, specifically in case of obligations in an obligation base $N_\Gamma$. Otherwise, only a prohibition is added into the prohibition base and there are no updates to other components.

$$\frac{N \neq \{\} \quad C_E \neq \{\}}{\langle ag, C, N, T, \text{RcvNorm} \rangle \rightarrow \langle ag', C', N', T, \text{SelEv} \rangle}$$ (RcvNorm)

where:

$$ag'_b = ag_b \cup \{\gamma\}$$
$$N'_\Gamma = N_\Gamma \cup \{\gamma\} \cup N'_\Xi = N_\Xi \cup \{\xi\}$$
$$C'_E = C_E \cup \{\langle \gamma, i \rangle\}$$

Relevant plans: If the transition of states (RcvNorm $\Rightarrow$ SelEv) is successful after RcvNorm and the state SelEv selects one event from the component $E$ of which event is either $(te, i)$ or $(\gamma, i)$, rule 4.2 starts to assign the set of relevant plans to component $T_R$ in the state RelPI. Rule 4.2 indicates the reconsideration situation where a new
triggers event extracted from the obligation is assigned to the component \(C_E\), where \(\text{Evt}(\gamma)\) is a function constructing a triggering event by the retrieval of information from \(\gamma\). Rule 4.2 assigns a set of relevant plans to \(T_R\) in respect of the reconsidered event. Rule 4.2 and 4.2 cope with the situation where no relevant plan is retrieved. In those cases, events (both ordinary event and reconsidered event) are simply ignored and the state returns to \(\text{SelEv}\).

\[
T_e = \langle te, i \rangle \quad \text{RelPlans}(ag_p, te) \neq \{\}
\]

where:
\[
T_R' = \text{RelPlans}(ag_p, te)
\]

\[
T_e = \langle \gamma, i \rangle \quad \text{RelPlans}(ag_p, \gamma) = \{\}
\]

where:
\[
C_E' = \{\langle \text{Evt}(\gamma), i \rangle\}
\]

\[
T_e = \langle \text{Evt}(\gamma), i \rangle \quad \text{RelPlans}(ag_p, \text{Evt}(\gamma)) \neq \{\}
\]

where:
\[
T_R' = \text{RelPlans}(ag_p, \text{Evt}(\gamma))
\]

\[
\text{RelPlans}(ag_p, te) = \{\}
\]

\[
\langle ag, C, N, T, \text{RelPl} \rangle \rightarrow \langle ag, C, N, T, \text{SelEv} \rangle
\]

Since transition rules between (\(\text{AppPl} \mapsto \text{AddIM}\)) are almost same as those in \textit{Ja-son} we give a brief description of each rule at each state from here. If \(T_R'\) is successfully assigned then it is followed by: (i) \(\text{AppPl}\) which assigns a set of applicable plans to \(T_A\) by retrieving those relevant plans whose contexts are believed to be true, (ii) \(\text{SelAppl}\) which assigns a particular intended means selected by an option selection function \(S_O\) to \(T_p\), and (iii) \(\text{AddIM}\) which adds a selected intended means to \(C_I\) which is an existing intention or a newly created one. If transitions fail between (\(\text{AppPl} \mapsto \text{AddIM}\)), then the state \(\text{SelInt}\) becomes the next step. For more information, see [3].

**Scheduling of intentions:** Rule 4.2 updates the component \(C_I'\) by the function \(\text{SCHEDULE}(C_I)\). Note that the scheduling function, \(\text{SCHEDULE}(C_I)\), sorts intentions in order of priority and deadline so as to determine the preference maximal set of intentions discussed in §3.4.

\[
T_p = \{\}
\]

\[
\langle ag, C, N, T, \text{SchInt} \rangle \rightarrow \langle ag, C', N, T, \text{SelInt} \rangle
\]

where:
\[
C_E' = \text{SCHEDULE}(C_I)
\]

After this step, the transition system follows the same rules as presented in [3] in order to execute an intended means in a particular intention selected by \(S_I\) in between \(\text{SelInt}, \text{ExecInt}\) and \(\text{ClrInt}\).
5 Related Works

There has been much research over a number of years on the matter of norm compliance through the combination of normative frameworks and classical (BDI-type) cognitive agents [2, 11]. However, research on compliance of norms at the individual agent level has received less attention. As discussed in §1, this problem can be decomposed into two perspectives: to facilitate a generic norm execution mechanism at run-time, and to focus on the rational decision making between norms and existing goals.

Alechina et al. [1] introduce N-2APL, a norm-aware BDI agent architecture and its programming language. It is able to carry out norm-aware deliberation, which aims to permit agents to resolve the conflicts between an agent’s own goals, normative goals and sanctions. This is accomplished by a deadline- and priority-based intention scheduling algorithm, which weighs the feasibility for all intentions that may bring about conflicts. The (potential) sanctions may affect agent decision making, but violations are possible in this approach. Given N-2APL, Dybalova et al. [9] demonstrate norm-compliant agents in location-based gaming environments in conjunction with the organisational framework, 2OPL [6]. There, once organisations have broadcast state-based norms to all participants, the individual agents achieve a state of the environment described in the norms using a design-based approach. N-Jason is also able to support norm-aware deliberation in conjunction with an institutional model, which is similar to the combination of N-2APL and 2OPL, but extends the concept of norm awareness to the whole reasoning cycle. As a result, it supports agents in being design-based norm compliant, but can additionally deliver run-time compliance through norm execution.

Meneguzzi et al. [13] focuses on norm awareness at the perception level, by extending the AgentSpeak(L) BDI architecture with a run-time plan modification technique. It enables agents to behave appropriately in response to newly accepted norms at runtime. However, it assumes that the norms are non-conflicting, so it does not consider scheduling of plans with regards to their deadlines or possible sanctions in accordance with existing goals in agents. Whereas [13] takes a rather practical perspective, van Riemsdijk et al. [14] introduce a formal framework for generic norm execution, which allows agents to be norm compliant by triggering or preventing actions in new and unknown norms at design time. However the agent in [14] works at the level of individual actions (its decision mechanism chooses actions rather than plans) and the norms are specified in terms of actions, making in effect a norm-reactive agent, and it is unclear how the decision mechanism can combine actions to achieve goals and thereby the objective of a norm-deliberative agent. In N-Jason, run-time norm execution is in practice accomplished at the level of plans to achieve goals, and norms indicate a sort of event that triggers plans. Moreover, in N-Jason run-time norm compliance is achieved on top of the norm aware decision making and in conjunction with the execution mechanism.

Notwithstanding the benefits of N-Jason, there are some issues to discuss, particularly regarding the mechanism for run-time norms. In the run-time norm execution, the norm compliance strategy is hard-coded in the semantics of the language, whereas such a strategy is programmable as agent plans (i.e. supporting the design of strategy by an agent programmer) in JaCaMo [2] and N-2APL [1]. Thus, the proposal in this paper somewhat simplifies normative reasoning, since it deprives agents of a flexibility to change the plans dynamically or mis-behave intentionally, based on rules the agent pro-
grammer designs. However, the mechanism we propose can enable legacy agents which have no compliance rule or strategy in their specification to become norm-aware automatically. Thus, those agents’ behaviour can be coordinated through the governance of normative frameworks without further engineering efforts.

Another issue lies in the simple mechanism for the operationalisation of norms in run-time norm execution. The approach described here means the ontology and syntax of norms that can be executed are limited to those present in the plan library of an agent. In consequence, some detached norms, that may correspond semantically to one of an agent’s plans, but which are ontologically different from the plan, will be ignored or violated. We are considering how to generalise the execution mechanism with the analysis of semantics of norms, following [14], in conjunction with plan synthesis.

6 Conclusion and Future Works

In this paper, we have presented a design for a norm-aware BDI agent, N-Jason, that enables the exhibition of norm compliance at run-time. Basically N-Jason offers a generic norm execution mechanism on top of norm-aware deliberation to contribute to the exploitation of run-time norm compliance. Run-time norm execution specifically focuses on the operationalisation of new and unknown (event-based) norms not stated in the agent program at run-time. By judging the executability of them, N-Jason agents executes those norms following an extended model of norm awareness consisting of: (i) event reconsideration, to find out what the norm is intended to achieve or to reach, and (ii) option reconsideration, to identify which plan is the most appropriate in response to the norm. The selection of norm compliant behaviour is achieved in the norm-aware deliberation process by intention scheduling with deadlines, priorities and prohibitions which confirms the decision about which behaviour agent would prefer between goals, norms and sanctions. It brings about a preference maximal set of intentions in order to realise the norm compliance. N-Jason is implemented in Jason/AgentSpeak(L) and extends its syntax and semantics to create N-Jason.

We believe that run-time norm compliance model is beneficial for the enhancement of both a norm compliance capability and agent autonomy from the agent’s perspective. However, we note that the behaviour triggered by run-time norm execution may look like unpredictable/unwanted behaviour from the agent programmer’s perspective.

Although this paper particularly considers the execution of event-based norms at run-time in conjunction with the institutional model, the extension to support state-based norms and its normative systems can easily be incorporated into N-Jason agents and will be as future work. We also plan to detect violations which are generated in the norm aware deliberation, particularly when the normative goals are dropped during scheduling. This offers a potentially useful link for enforcement in the context of normative system implementation. In addition, both empirical and analytical evaluation of the performance of N-Jason requires proper investigation.

References


Typing Multi-Agent Systems via Commitments

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati

Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{matteo.baldoni,cristina.baroglio,federico.capuzzimati}@unito.it

Abstract. This work presents an agent typing system, that differently than most of other proposals relies on notions that are typical of agent systems instead of relying on a functional approach. Specifically, we use commitments to define types. The proposed typing includes a notion of compatibility, based on subtyping, which allows for the safe substitution of agents to roles along an interaction that is ruled by a commitment-based protocol. Type checking can be done dynamically when an agent enacts a role. The proposal is implemented in the 2COMM framework and exploits Java annotations. 2COMM is based on the Agent & Artifact meta-model, exploit JADE and CArtAgO, by using CArtAgO artifacts in order to reify commitment protocols.

Keywords: Commitments, Static and dynamic type checking, Agents and Artifacts, JADE, Implementation

1 Introduction

Software infrastructures are quickly changing, becoming more and more global, pervasive and autonomic. Computing is becoming ubiquitous, with embedded and distributed devices interacting with each other. Multi-Agent Systems (MAS) have been recognized to be a promising paradigm for this kind of scenarios, however, as the complexity of programming these systems increases, the need for effective tools for reasoning on properties of programs becomes stronger and stronger. This is particularly true in the case of open systems, where heterogeneous and autonomously developed agents may need to interact. MAS usually rely on interaction protocols (or other kinds of “contract”) to specify the interacting behavior that is expected of the agents. How can, then, an agent, a designer, the system verify that the agent has the the means for carrying on the encoded interaction? How to decide whether the agent is capable of behaving in a certain way or whether it shows specific skills/properties?

One way is to rely on some typing of agents, in a way that is similar to the typing of objects. Typing provides abstractions to perform sophisticated forms of program analysis and verifications: it helps performing compile-time/run-time error checking, modeling, documentation, verification of conformance and of compliance, reasoning about programs and components. It also allows a simple form of (a priori/runtime) verification. To the best of our knowledge, Zapf
and Geihs [34] were the first to propose the use of a type system for (mobile) agents, and they also introduced the idea of using sub-typing for the substitution of more specific subclasses in places where more general classes are expected, thus supporting safe extension and program re-use. More recent examples include [18,19,1,26]. In particular, [26] describes an agent-oriented programming language with a type checking that is inspired by mainstream object-oriented languages, and [1] uses global session types for realizing monitors of the interaction.

Differently than [18,19,26], we believe that, since types are abstraction tools for easily programming and modeling, for typing MAS it is necessary to rely on concepts that are typical abstractions of MAS, rather than relying on abstractions from other programming paradigms. Similarly to [1], our proposal is centered around interaction, which we believe to be one essential aspect of MAS. Differently than [1], we rely on commitments rather than on global session types. Commitments [13,28] are one of the fundamental abstractions for ruling agent interaction while preserving agent autonomy. For this reason, we discuss how commitments can be used for typing MAS and why it is interesting to rely on them. Specifically, we report the first steps towards a definition of a behavioral-based typing system for autonomous agents. The proposal is not bound to a specific agent programming language but, rather, it can be implemented in different frameworks. In the paper we describe an implementation in 2COMM [2].

The paper is organized as follows. Section 2 reports and comments the relevant literature motivating our proposal. Section 3 describes the 2COMM system that we used for the implementation. Section 4 introduces the type system, while Section 5 describes its implementation. Conclusions end the paper.

2 Background and Motivation

The notion of “typing an agent” requires a precise, crisp definition. In programming languages, type systems are used to help designers and developers in avoiding code errors, bugs, that can entail unpredictable results. Type systems can be weak or strong, static or dynamic, but at the end they all share the same goal: support the development of error-free and human-readable code.

Most of agent system implementations (JADE [9], Jack [20], A-Globe [29]) are based on programming languages like Java and do not supply agent type support but rather rely on the typing system of the language used for developing the system. Zapf and Geihs [34] underlined the importance of using a type system which allows dynamic type checking and proposed to base agent typing (1) on the externally visible actions of the agents, that they identify as being the messages agents accept and send, (2) on the meaning of the messages agents can exchange which includes, through the special symbol self, a characterization of the agent itself, (3) on the used communication protocol. They structure an agent type as a triple. The first component is the syntactic type, which is stateless and consists of the set of the input messages and of the set of output messages. The second is a transition type, i.e. a finite state automaton capturing a communication
protocol similarly to regular types [22]. The third and last component is the semantic type, an annotation aimed at checking behavior-compatibility, based on J. F. Sowa's conceptual graphs.

We agree on the importance of dynamic type checking for verifying that an agent fits the requirements for interacting in an open MAS in the moment the agent decides to enter the interaction, because it may have the required properties only when it enters the system; on the importance of relying only on externally visible actions, because the agents’ internal states are not inspectable; on the importance of accounting for the interaction protocol, because it captures the rules of encounter of the agents, ruling their interaction. What we disagree with is the solution adopted by the authors of relying on finite state automata for describing the interaction as well as for describing the agents’ behavior. This hinders the agent’s autonomy in two ways. The first reason is that agents must supply a description of their behavior. Secondly, this description concerns how to do things, rather than what to do: it is prescriptive. An agent may have the possibility (and the capability) of doing something in different ways. We think that the typing system should be capable of featuring a more flexible representation of the behavior, with the possibility of leaving the choice of how to act up to the agent.

The main claim of [1] is the importance of using interaction protocols for representing the functioning of a system. To this aim, they use global session types as an abstraction tool, which allows automatically generating monitors that are aimed at verifying the correctness of on-going, multi-party interactions. In particular, the global session type is used to automatically generate a monitor agent, which intercepts all the exchanged messages and verifies whether the protocol is respected. This proposal is implemented in Jason [12]: a global session type is represented by a cyclic Prolog term, which is consumed as messages are sniffed. Along the line of the previous proposal, [1] focuses on externally visible actions (message exchanges) and on the use of interaction protocols. It differs from the previous one in that there is no actual type system, but rather global session types are used for specifying the interaction of a system from a global perspective. Since agents are not typed, when they enter a system, it is not possible to verify whether their behavior is compatible with the protocol nor it is possible to search for agents showing characteristics which allow them to successfully take part to the system. It is up to the monitor agent to check the exchanged messages. This is surely an important functionality but it is not type checking. In other words, the representation does not clearly express what an agent can do nor what is expected of an agent. Moreover, we disagree with the choice of realizing the monitor as an agent. In order for the system to be transparent, the monitor should be inspectable by the interacting agents, and the infrastructure should guarantee that the monitor is notified of all the exchanged messages. We believe that the environment should supply proper monitoring services, or an artifact, but not another autonomous agent.

Ricci and Santi [25,26] defined the SimpAL language, where types are seen as useful for realizing integrated development environments, and they implemented
an Eclipse plugin \[27\]. The approach to typing is a classic one, grounded on interfaces. This is the way in which most of the programming languages assure coherence, and prevent (statically) or detect (dynamically) logical errors. SimpAL extends the notion of interface to the agent abstraction level, introducing the notion of role as a collection of tasks, that an agent is capable to perform. A role will be implemented by an agent script, containing the behavioural logic of the agent. Specifically, a SimpAL role is an interface, while a role task is a method signature, which includes a list of formal parameters needed for its completion, that are expressed as pairs \langle \text{name} : \text{Type} \rangle. SimpAL provides environment typing and organizational typing too, used for programming coordination, resources and interactions between agents.

A typing of agents merely based on syntactic interfaces is criticized in \[34\], where the authors explain how conventional typing does not suffice the context of agent systems. The critic bases upon work by Nierstrasz \[22\] on active objects, that showed how the enumeration of the possible input and output messages is not sufficient to guarantee the interoperability. It is advisable to rely, instead, on some sort of behavioral type, including semantic information. Moreover, in SimpAL agent type checking is static. This is not a major concern in a homogeneous, single application environment. However, in an open MAS, where agents may be composed dynamically, static type checking is not enough; instead, it is necessary to rely on dynamic type checking and on monitoring. In this setting, agents themselves may verify their conformance to a role in order to decide whether to enter an interaction as well as to decide whether adopting new behaviors. As a consequence, the notion of type not only is a tool that supports the programmer’s work but it becomes an programming element, that is used by agents in order to take decisions.

The proposal that we present in this paper concerns an agent typing system, which is characterized by (1) being based on typical agent society abstractions (social relationships), (2) being based on the agents’ observable behavior, (3) dynamically checking if agents satisfy role requirements, (4) supplying a run-time monitoring environment. The implementation is provided in 2COMM, a middleware for developing open MAS whose interaction is commitment-based \[2\], which combines the well-known JADE \[9\] and CArtAgO \[24\] platforms. JADE agents interact based on commitment protocols. Each interaction protocol is realized as a CArtAgO artifact. Such an artifact provides social relationships as environmental resources. Dynamic checks are realized based on Java annotations.

3 Reference Framework

This proposal relies on the 2COMM middleware \[2,3\] for developing Multi-Agent Systems. In 2COMM, the MAS is specified as a set of social relationships, that govern the behavior of the agents taking part into the system. In a system made of autonomous and heterogeneous actors, social relationships cannot but concern the observable behavior \[17\]: for this reason, and in order to give them
that normative value which allows them to create social expectations, we realize social relationships by means of commitments [28].

On the other hand, we need social relationships to be accepted explicitly by the participants to the interaction, and possibly to be inspected by the agents, in order to decide whether conforming to them. To this aim, we need to explicitly model social relationships as resources, that are available to the interacting peers. Given that agents and social relationships are both first-class entities, that interact in a bi-directional manner, we adopt the Agents and Artifacts (A&A) meta-model [32,23], that extends the agent paradigm with another primitive abstraction, the artifact. A&A provides abstractions for environments and artifacts, that can be acted upon, observed, perceived, notified, and so on. When embodied inside artifacts, social relationships can be examined by the agents (to take decisions about their behavior), as advised in [14], used (which entails that agents accept the corresponding regulations), constructed, e.g., by negotiation, specialized, composed, and so forth.

2COMM\(^1\) [2] provides a middleware for programming social relationships, by exploiting a declarative, interaction-centric approach. It is based on a combination of JADE [9] and CArtAgO [24]. JADE provides the agent platform, characterized by a FIPA compliant communication framework, and an agent-developing middleware. CArtAgO is a framework based on the A&A meta-model which extends the agent programming paradigm with the first-class entity of artifact: a resource that an agent can use. CArtAgO provides a way to define and organize workspaces, that are logical groups of artifacts, and that can be joined by agents at runtime. The environment is itself programmable and encapsulates services and functionalities. CArtAgO provides an API to program artifacts that agents can use, regardless of the agent programming language or the agent framework used. CArtAgO artifacts reify communication and interaction, represented in terms of commitment-based protocols. From an organizational perspective, a protocol is structured into a set of roles. A role represents a way of manipulating the social state and belongs to the artifact which reifies a protocol. Roles and agents are different entities, and we assume that roles cannot live autonomously: they exist in the system in view of the interaction, because agents, for interacting, use artifacts and execute actions on them [8]. Agents will use an interaction artifact to establish a channel of normed, mediated communication. The roles of such an artifact specify how agents can manipulate it: by enacting a role, an agent receives social powers by the artifact. Social powers have different and public social consequences, that we express in terms of commitments.

In 2COMM interaction is ruled by commitment-based protocols. A commitment \(C(x,y,r,p)\) represents a directed obligation between a debtor \(x\) and a creditor \(y\) to bring about the consequent condition \(p\) when the antecedent condition \(r\) holds. A commitment may be manipulated by means of a set of primitives: delegate, assign, release [30]. They represent contractual relationships between agents, thus agents have the social expectation that an agent involved

\(^1\) The source files of the system and examples are available at the URL http://di.unito.it/2COMM.
in a commitment as a debtor will realize the consequent condition; the debtor is responsible for the violation of a commitment. A commitment protocol defines a collection of actions, whose social effects are expressed in terms of commitment primitives, e.g., adding a new commitment, releasing another agent from some commitment, satisfying a commitment, see [33].

Figure 1 shows an excerpt of the 2COMM UML diagram. Overall the middleware is organized as follows: JADE supplies standard agent services (message passing, distributed containers, naming and yellow pages services, agent mobil-
ity); when needed, an agent can enact a protocol role, thus using a communication artifact – implemented by exploiting CArtAgO, which provides a set of operations by means of which agents participate in a mediated interaction session. Each communication artifact corresponds to a specific protocol enactment and maintains an own social state and an own communication state.

Class CommunicationArtifact (CA for short) provides the basic communication operations in and out for allowing mediated communication, by means of which agents respectively ask to play or to give up playing a role. CA extends an abstract version of the TupleSpace CArtAgO artifact: briefly, a blackboard that agents use as a tuple-based coordination means. In and out are, then, operations on the tuple space. CA also traces who is playing which role by using the property enactedRoles.

Class Role extends the CArtAgO class Agent, and contains the basic manipulation logic of CArtAgO artifacts. Thus, any specific role, extending this super-type, will be able to perform operations on artifacts, whenever its player will decide to do so. Role provides static methods for creating artifacts and for enacting/deacting roles. This is done by passing a reference to the JADE agent behavior that will actually play the role. The class CARole is an inner class of CA and extends the Role class. It provides the send and receive primitives, by which agents can exchange messages. Send and receive are implemented based on the in and out primitives provided by CA.

ProtocolArtifact (PA for short) extends CA and allows modeling the social layer with the help of commitments. It maintains the state of the on-going protocol interaction, via the property socialState, a store of social facts and commitments, that is managed only by its container artifact. This artifact implements the operations needed to manage commitments (create, discharge, cancel, release, assign, delegate). PA realizes the commitment life-cycle and for the assertion/retraction of facts. Operations on commitments are realized as internal operations, that is, they are not invokable directly: the protocol social actions will use them as primitives to modify the social state. Being an extension of CA, PA maintains two levels of interaction: the social one (based on commitments), and the communication one (based on message exchange). The class PARole is an inner class of PA and extends the CARole class. It provides the primitives for querying the social state, e.g. for asking the commitments in which a certain agent is involved, and the primitives that allow an agent to become, through its role, an observer of the events occurring in the social state. For example, an agent can query the social state to verify if it contains a commitment with a specific condition as consequent, via the method existsCommitmentWithConsequent(InteractionStateElement el). Alternatively, an agent can be notified about the occurrence of a social event, provided that it implements the inner interface ProtocolObserver. Afterwards, it can start observing the social state. PARole also inherits the communication primitives defined in CARole.

In order to specify a commitment-based interaction protocol, it is necessary to extend PA by defining the proper social and communicative actions as operations
on the artifact itself. Actions can have guards that correspond to context preconditions: each such condition specifies the context in which the respective action produces the described social effect. Since we want agents to act on artifacts only through their respective roles, when defining a protocol it is also necessary to create the roles. We do so by creating as many extensions of PARole as protocol roles. These extensions are realized as inner classes of the protocol: each such class will specify, as methods, the powers of a role. Powers allow agents who play roles to actually execute artifact operations. The reification of commitment protocols by way of artifacts has many advantages: by exploiting the distributed nature of artifacts it is possible to naturally rely on a modularization that helps the re-use of software, it is possible to implement run-time monitoring functionalities, and it is possible to provide a normative characterization of interaction thanks to commitments.

4 Typing MAS

To the aim of defining an agent typing system, we assume each agent $a$ to be characterized by a set of behaviors $\{b_1, \ldots, b_m\}$, enabling $a$ to perform various activities. Along the lines of [22], we view types as partial specifications of behavior, which support in using agents to play protocol roles safely. A type $\tau$ is a set of commitments $\{c_1, c_2, \ldots, c_n\}$, defined inside a collection of definitions of artifacts, that represents the environmental setting. The debtor, creditor, conditions of each commitment are defined as roles and actions inside some artifact, i.e. artifact definitions provide name spaces. Commitments, by having a normative value, can be seen as specifications of behavior because the debtor agents are expected to behave so as to satisfy them. A behavior $b$ has type $\tau$, denoted as $b : \tau$, if it is capable of satisfying the commitments in the type. This means that it allows to make the consequent conditions in the commitments become true.

We assume that for every event (action) involved in the consequent condition, there is at least a corresponding artifact operation. For example, having a commitment $C1 = C(x, y, r, p \land q)$, a protocol artifact needs to supply an operation that makes $p$ true and an operation that makes $q$ become true. Such operations are to be associated to role $x$.

Definition 1 (Type). Given an agent $a$, with a set of behaviors $b_1 : \tau_1, \ldots, b_m : \tau_m$, we say that $a$ has type $\tau = \bigcup_{i=1}^m \tau_i$, denoted as $a : \tau$.

Let $P = r_1 \circ \ldots \circ r_n$ be an interaction protocol, where $r_i$ are all the protocol roles. Let $p$ be a protocol action, whose execution creates the commitments $c_1, \ldots, c_n$, (conditionally) binding the executor to achieve some conditions. This represents the fact that $p$ requires the executor to have the capability of satisfying (directly or indirectly – i.e. by way of other agents) $c_1, \ldots, c_n$. So, we say that $p$ has type $\tau = \{c_1, \ldots, c_n\}$, denoted as $p : \tau$. 

348
Definition 2 (Role and Protocol Types). Let \( p_1 : \tau_1, \ldots, p_m : \tau_m \) be the actions of \( P \) that the role \( r_j \) allows to execute together with their respective types. The type of role \( r_j \) is \( \tau_j = \bigcup_{i=1}^{m} \tau_i \). Finally, the type of \( P \) is \( \{ r_1 : \tau_1, \ldots, r_n : \tau_n \} \).

We, now, introduce a notion of subtype, that is inspired to the width subtyping used for records. Given two types \( \tau_1 \) and \( \tau_2 \), we say that \( \tau_1 \) is a subtype of \( \tau_2 \), denoted by \( \tau_1 \leq \tau_2 \), when the set of commitments of \( \tau_2 \) is included in the one of \( \tau_1 \), i.e. \( \tau_2 \subseteq \tau_1 \). A subtype is a stronger specification which guarantees that the set of values satisfying it is a subset of the set of values of the supertype.

What kinds of properties should types specify? According to the principle of substitutability \([31]\) an instance of a subtype can always be used in any context in which an instance of the supertype is expected. A subtype at least guarantees the “promises” of the supertype, at least the same commitments, and possibly more, are satisfiable.

Since our subtyping relationship is defined based on subset inclusion, it is easy to see that subtyping is a partial order, and thus shows the properties of reflexivity, antisymmetry, and transitivity. More interestingly, the subsumption property also holds: consider an agent \( a : \tau \) and suppose \( \tau \leq \tau' \), then \( a : \tau' \).

The rationale of the proposed subtyping relationship is that we mean to support the substitution of an actual agent and its behaviors to the specification of requirements that is given by a role: any behavior which is capable of achieving a superset of the required commitments will fit our case. Any operation feasible on the supertype will be supported by the subtype. This definition makes it possible to introduce a notion of compatibility of agents with roles.

Definition 3 (Compatibility). An agent \( a : \tau \) is compatible with a protocol role \( r : \tau' \) if \( \tau \leq \tau' \).

In fact, since \( a : \tau \) and \( \tau \leq \tau' \), by subsumption \( a : \tau' \). So, \( a \) has all the capabilities necessary to achieve the commitments it could get engaged into, when playing \( r \). Generally, \( a \) will have a more specialized behavior w.r.t. what the role demands.

We, now, show that subtyping guarantees substitutability: namely, that substituting a role by an agent that is compatible with it preserves the type of the protocol. Such a verification should be performed dynamically during the enactment of the protocol role.

Property 1 (Substitutability). Let \( P = r_1 \circ \ldots \circ r_n \) be an interaction protocol of type \( \tau \). The system obtained by the enactment of the protocol, performed by the set of agents \( a_1, \ldots, a_n \), each compatible with its respective \( P \) role, preserves the type \( \tau \).

The proof is trivially obtained by considering the above definitions.

Besides the behavioural-oriented notion of typing described above, we rely on Java to perform event (action) type checking. In fact, since they are implemented as artifact operations, when an agent uses an operation, through a role, the Java compiler checks the correctness of the parameters.

By adopting classical depth and width subtyping rules for records, i.e. \( \{ r_1 : \tau_1, \ldots, r_m : \tau_m \} \leq \{ r_1' : \tau_1', \ldots, r_m' : \tau_m' \} \) if \( m \leq n \) and \( \tau_i \leq \tau_i' \) for all \( i \) from 1 to \( m \), it is possible to introduce also a notion of protocol specialization.
Definition 4 (Specialization). Let \( P : \tau \) and \( P' : \tau' \) be two interaction protocols with their respective types. We say that \( P' \) is a specialization of \( P \) if \( \tau' \leq \tau \).

5 Implementing the typing in 2COMM

Let us, now, introduce the way in which we implemented the proposed typing system in 2COMM. The implementation relies on Java annotations\(^2\). These are commonly used to provide meta-data about a program which can be used by the compiler, or be used at deploy time or, as in our case, at run-time.

With reference to Figure 2, we introduced two annotations, one for interaction protocol roles, the other for agent behaviors. They are respectively @RoleType and @BehaviourType. They both represent commitment sets. The former via the annotation property requirements, the latter via the annotation property capabilities. @RoleType also contains a property interactionCardinality, specifying

\(^2\) More information about Java annotations can be retrieved at http://docs.oracle.com/javase/tutorial/java/annotations/
whether a role can be concurrently played by many agents – as it is, for instance, the case of the Contract Net Protocol role Participant.

In our implementation, a type (Definition 1) is specified as an object of sort Type, which is an abstract class which contains the field definition (an array of commitments).

```java
public abstract class Type {
    final private ArrayList<Commitment> definition;
    protected Type(Commitment[] commitsDefinition) {
        definition = new ArrayList<Commitment>();
        for (Commitment c : commitsDefinition) {
            definition.add(c);
        }
    }
    public boolean isIncluded(Type includerType) {
        boolean included = true;
        for (Commitment c : this.definition) {
            if (included) {
                included = false;
                for (Commitment d : includerType.definition) {
                    if (c.equals(d)) {
                        included = true;
                        break;
                    }
                }
            } else break;
        }
        return included;
    }
    public boolean equals(Type t) {
        return this.isIncluded(t) && t.isIncluded(this);
    }
    public static Type merge(ArrayList<Type> typesToMerge) {
        ...
    }
}
```

Type must be subclassed by actual types, whose constructors will invoke the superconstructor and specify proper arrays of commitments. Moreover, Type specifies two methods, equals and isIncluded (that we report hereafter) which respectively verify if a type (set of commitments) is identical to another and if a type is subtype of another. A static, utility method merge is provided too, that creates a new Type object from the union of commitments of types passed as parameters.

The equals method considers two commitments equal if all their components are respectively equal.

```java
public boolean equals(SocialStateElement el) {
    if (el.getElType() != SocialStateElementType.COMMITMENT)
        return false;
    Commitment c = (Commitment) el;
    return (this.getCreditor().equals(c.getCreditor()) &&
            this.getDebtor().equals(c.getDebtor()) &&
            this.getAntecedent().equals(c.getAntecedent()) &&
            this.getConsequent().equals(c.getConsequent()));
}
```

Antecedent and consequent formulas have to match exactly, while the identities of creditors and debtors are checked as follows:
The implementation can compare commitments that are instantiated and involve specific agents or that are “generic”, in that they involve protocol roles. To separate the two cases, in the former the debtor and creditor of a commitment are associated to the case \textit{PARTICULAR\_ROLE} while in the latter they are associated to the case \textit{GENERIC\_ROLE}. This information is used by the method \textit{equals}: A debtor/creditor identity is considered equal to that of another in two cases: (1) when the two refer to the very same enactment of a certain role (i.e. they refer to the same agent); (2) when one or both identities refer to a role type (e.g. the initiator) and the respective role names are equal.

With reference to Figure 3, type checking amounts to verifying if the commitments specified in the \textit{capabilities} property of annotation \textit{@BehaviourType} include the commitments specified in the \textit{requirements} of the annotation \textit{@RoleType}. The check is performed by the method \textit{checkRoleRequirements} which is included in the class \textit{CommunicationArtifact}. This method, which is executed in the context of \textit{enactRole}, uses the set of behaviors of an agent and the role this means to play, and computes an answer by extracting at run-time the information contained in the involved annotations. An agent can successfully enact a role only if it is compatible with it (Definition 3), i.e. only if its type is a subtype of that of the role. For the property of substitutability, the enactment preserves the type of the protocol, thereby assuring safety.
When an agent tries to enact a role, the artifact, whose role is being enacted, is in charge of checking the compliance between the agent’s behaviour and the role requirements. The method `checkRoleRequirements` of the class `CommitmentArtifact` performs these controls. This implementation realizes the principle of compatibility: an agent can enact a role provided it has a (set of) behaviour(s) that are compatible with the type of the role.

The `Type` abstract class, together with the `@RoleAnnotation` and `@BehaviourType` annotation classes, allows constructing types as Java structures, an approach similar to the one proposed in [34], where each agent carries an object representing its type.

Let us, now, show an example of annotation added on top of an implementation of the Contract Net Protocol presented in [3]. We will focus on the role `Initiator` and on an agent willing to play that role.
The role Initiator is tagged by the `@RoleType` annotation, whose value for the property `requirements` is set to `InitiatorRequirements.class`, a class that builds the set of commitments that defines the type of the role. `InitiatorRequirements` is specified in this way:

```java
public class InitiatorRequirements extends Type {
    public InitiatorRequirements() throws MissingOperandException, WrongOperandsNumberException {
        super(new Commitment[] {
            new Commitment(CNPArtifact.INITIATOR_ROLE, CNPArtifact.PARTICIPANT_ROLE, "propose", new CompositeExpression(InternalOperatorType.OR, new Fact("accept"), new Fact("reject"))),
            new Commitment(CNPArtifact.BUYER_ROLE, TradeArtifact.SELLER_ROLE, "pay", "deliver")
        });
    }
}
```

Specifically, this class contains the commitment $C(CNPArtifact.INITIATOR_ROLE, CNPArtifact.PARTICIPANT_ROLE, propose, accept \lor reject)$, where `CNPArtifact` is the CommitmentArtifact which realizes the Contract Net Protocol.

On the agent's side, an agent willing to play the role Initiator must offer a set of behaviors that are typed accordingly. In our case, we suppose that the agent offers the following behavior:

```java
public class InitiatorBehaviour extends OneShotBehaviour implements CNPInitiatorObserver {
    ...
}
```

where the class `TypeInitiator` specifies the `capabilities` shown by the agent through the behavior. Once again, this is a set of commitments the behavior can satisfy. `TypeInitiator` is a subclass the `Type`:

```java
public TypeInitiator() throws MissingOperandException, WrongOperandsNumberException {
    super(new Commitment(CNPArtifact.INITIATOR_ROLE, CNPArtifact.PARTICIPANT_ROLE, "propose", new CompositeExpression(InternalOperatorType.OR, new Fact("accept"), new Fact("reject"))));
}
```

It is easy to see that the commitment perfectly matches the requirements, and so the enactment will succeed. Notice that the presented implementation is slightly
different w.r.t. the definition of compatibility with a role (Definition 3): it uses a collection of behaviours instead of an agent because in JADE there is no reference to the agents that we could exploit. The result is a more restrictive test, which does not necessarily account for the whole agent but considers only the set of behaviors the agent displays.

6 Discussion and Future Work

2COMM aims at providing adequate support for programming social relationships, by exploiting a declarative, interaction-centric approach and by relying on existing technologies as far as possible. In 2COMM, commitments are first-class objects. They capture social relationships between agents, that arise from their interaction. We used them to define requirements for role enactment. The use of commitments gives a normative characterization to coordination [13,28], whose public acceptance of the regulation allows reasoning about agents’ behavior [15].

Our aim is to provide static, compile time coding support and dynamic, runtime type checking. The first aspect would be the basis for the development of IDE coding support [27], like smart code completion or type warning or error; the second checks compliance between the agent’s logics and the role requirements at runtime, signalling the occurrence of wrong enactments. The aim is to guarantee the substitutability property, which guarantees the safe replacement of agents to roles, when they have the same type or the agent has a subtype of the role. In this proposal such a verification is performed as a syntactic inclusion of commitment sets. This is limiting because it does not consider logical expressions inside commitment antecedent and consequent conditions. We mean to study the issue which may involve the use of complex typing systems, relying on union and intersection types [16].

The typing system we sketched relies on the social capabilities of the agents, rather than on which tasks they can perform, and it relies on notions that are typical of agents rather than on a functional approach. This is a novelty w.r.t. previous work on agent typing, which apply the functional type theory [18,19,26]. Although the functional approach benefits of the results of a vast literature, types should provide the right abstraction (modeling) features in order to help the programmer. By relying on a functional approach, the typing system discards the typicalities of agents and, thus, it does not accomplish its aims. We believe that a deeper exploitation of the advantages of using the MAS paradigm calls for a different approach to the definition of a type system.

We agree with [22,34] that the typing system should include a representation of the behavior but, differently than in those works which deal with objects, we need a representation of behavior which does not hinder the agents’ autonomy. For this reason, a prescriptive representation, based on finite state automata – as the one introduced in those works, would not be adequate. By relying on commitments it is possible to specify the expected behavior of agents in a minimally prescriptive way. In case a more expressive language for specifying constraints is needed, it is possible to rely either on proposals like [21], where
conditions inside commitments can express temporal regulations, or on proposals like 2CL [6], where commitment protocols are enriched with explicit temporal constraints on the evolution of the social state. This kind of extensions will be one of our next goals.

Clearly, a type system allows only a light check of the behavior of the involved agents, being more concerned with a safe usage rather than a full behavioral compatibility. It does not imply that an agent which has the same type of another agent will display the same behavior. This does not exclude the possibility to integrate deeper checks, for instance based on model checking such as [10].

Type checking as a light verification adopts notions, e.g. substitutability, that are used also for facing the issues of interoperability and conformance discussed in [7,5]. The conformance verification aims at guaranteeing that when an agent plays a role, or substitutes another agent in an on-going interaction, the interoperability of the system is preserved. In the present paper, when an agent plays a role the protocol type is preserved. In the cited works, protocol representation relies on formal languages (a sort of finite state automata), thus, suffering of the drawbacks due to a prescriptive description that does not suit the autonomy of the agents (as described in Section 2 for the approach in [1]). We mean to explore how commitment-based types can be adapted to solve the issue of conformance in MAS.

In [4], we presented an extension of JaCaMo [11] that, analogously to 2COMM, allows reasoning about social relationships in Jason agents. We aim to introduce the use of the proposed typing system also in that setting. This would allow an even deeper comparison to SimpAL, which is built on top of the same platform.

Acknowledgements

We thank the anonymous reviewers for their helpful comments, which gave us important suggestions for future developments.

References


Robust Collaboration:
Enriching Decisions with Abstract Preferences

Loïs Vanhée\textsuperscript{1,2}, Frank Dignum\textsuperscript{2}, Jacques Ferber\textsuperscript{1}

\textsuperscript{1} LIRMM, University of Montpellier II, France
\textsuperscript{2}Utrecht Universiteit, The Netherlands

Abstract. Preferences have been used for modeling agent decision processes, allowing agents to make abstract decisions about their future goals and plans. While being beneficial for making decisions, preferences are only used for making individually-centred decisions. This article investigates on the use of preferences for promoting abstract collaboration. This form of collaboration, by abstracting away from the environment, allows to create different types of assumptions than those feasible by concrete collaboration. In particular, abstract assumptions promote a more environmentally-free form of collaboration, which is more robust. This article investigates solutions for combining these two forms of collaboration. By combining them, collaboration design can be split in two simpler parts: a top-down abstract representation and a bottom-up concrete representation. This separation allows to design each type of collaboration independently, using appropriate representation tools, lowering overall design complexity.

Keywords: Agent Oriented Software Engineering, Methodology

1 Introduction

“The firefighter agent is about to enter the burning house to extinguish the fire and rescue victims. Should it immediately enter the house or first check that colleagues are ready for supporting? No need to waste precious seconds, the agent knows that others are very concerned about timeliness. If someone was not ready, the agent would have been warned.”

Such an assumption is hard to find as a designer when representing agents from a rather concrete level (e.g. through protocols or BDI agents), which correspond to the mainstream for designing of MAS. In traditional design, collaboration is obtained by adequately matching behaviors such that each agent’s action is “streamlined” with regard to system goals. Metaphorically, this step is similar to trying to assemble cogs such that teeth of the cog match well with the next: while participating in collaboration, each cog is blind to its impact on it; the collaboration is apparent only in the observer’s eye. This perspective carries with it some drawbacks, for instance, mismatching details of the collaboration leads to system failures.
What if, instead, agents were aware about how they are expected to interact? Even more, what if agents were collectively driven towards some form of collaboration? In other words, they would prefer to act while promoting this collaboration and share this preference with others. With such an assumption, they can create expectations about other individuals, as well as collective and environmental properties (such as the one from the first paragraph), without having to specify about how to the last detail. So, these abstract properties can be integrated in agent decisions, which can improve system performance, as shown in the example.

In addition, collaboration is not only achieved by “streamlining” concrete agent behaviors. Collaboration is a part of a common goal: agents prefer some property to be achieved (e.g. being in time). Then, they can use this abstract reasoning for triggering concrete behavior for achieving this collaboration (e.g. asking for help if an unexpected event leads to delay). This article aims at investigating how integrating collective preferences into agents, in particular if these preferences concern collaboration, can benefit to MAS.

While we do not aim at replicating human behavior, our idea of integrating collaboration through shared preferences is originally inspired by human culture studies. Human cultures are said to collectively influence individual values, that is to say, abstract individual preferences. This influence impacts in turn on collective behavior. Consider the example, imagine that firefighters had, a culture which promotes respect towards statuses (e.g. China) instead of current one which promotes timeliness (e.g. Germany). In such a culture, agents would give more importance to fire commander decisions because he would be culturally expected (and thus driven) to tightly manage the team. So, in the example the agent would have asked the fire commander that it was ready to get in. These two cultures promote different coordination patterns, which both have advantages and drawbacks on collective performance. To that extent, we inspired from human cultures to metaphorically design “artificial cultures”.

But, sharing abstract preferences is not sufficient to design concrete agent behavior. Abstract preferences provide abstract drives, which can then be used to drive more concrete agent behavior. This behavior being designed using appropriate tools for modelling concrete action, such as BDI agents or protocols. These two aspects combined forming hybrid agents to allow designing both abstract and concrete collaboration.

A running example illustrating our concepts throughout the article is described in Section 2. Then Section 3 describes related work. In Section, 4, we show how to use preferences for modelling abstract decisions and how they can be integrated into an hybrid agent model. The use of shared preferences for designing collaboration is described in Section 5. Examples of human cultures that can be used as inspiration for designing shared preferences is described in Section 6.
2 Running Example

A running example is used in order to better illustrate concepts and methods described throughout this article. Consider a MAS supporting a team of fire fighters. Each fire fighter has his own agent. Each agent keeps track of the information of the fire fighter’s situation and can confer with the other agents about which information or action advise to give to its fire fighter. In the following we identify the agents with the persons they support for ease of reference. The mission (or goal) of the agents consists in extinguishing fires and rescuing people who got injured due to the crisis. In addition to fire fighters, a special agent called the “fire commander” (represented by $a_{fc}$) located in the firetruck can communicate with the fire fighters using point to point communication.

In this setting, an agent (indicated by $a_1$) is about to make a decision. The situation of $a_1$ is as follows: Before the mission, the fire commander prepared a plan for each agent. $a_1$ is assigned the mission to be at the fire at time $T$. From there, $a_1$ has to support agent $a_2$ while $a_2$ extinguishes the fire. $a_1$ is moving to the fire. $a_1$ just spotted a person nearby. $a_1$ has to chose between three available options:

1. **Rescue**: $a_1$ delays its action to move towards the fire and rescues the victim instead. The time required to rescue the victim is unpredictable: if the victim is healthy, the action can be very quick (ask the victim to leave), if the victim is injured this action can take much longer (the agent has to stabilize and to carry the victim out of the danger zone). The fire fighter regulations state that $a_1$ is forbidden to leave a victim if a victim is injured.
2. **Report**: $a_1$ delays its action to move towards the fire and warns the fire commander about the presence of a person. This action takes some time but is quicker than helping.
3. **Ignore**: the agent stores the information that a person has been spotted and keeps moving towards the fire.

If $a_1$ has some available time before $T$, the situation is referred to as $d_t$. Otherwise ($a_1$ is short in time), the situation is referred to as $d_{\bar{t}}$.

3 Previous Work

3.1 Collaboration

Former research in MAS extensively investigated the design of multi-agent solutions that are capable of reaching system goals via the collective action of individual agents [6, 7, 9] [21, p. 189-224]. Numerous methods are proposed for promoting collaboration in MAS. These methods consist in restricting agent’s range of possible behaviors. These restrictions can then be used for directly enforcing some form of collaboration (e.g. everyone must drive on the right) or be used by agents in order to create expectations about other agent’s behavior or about the environment (e.g. expecting the water hose will be ready at some pre-defined time).
**Optimal Solutions** A first category of methods proposes solutions for automatically finding individual agent’s behavior leading to the most efficient collective outcome (e.g. Game Theory [3], DEC-POMDP [1]). These theories have for major drawback that they require an extensive knowledge of environmental dynamics. This drawback leads to other limitations: agents perfectly fit one environment but are inflexible to any unexpected environment. In addition, these methods are operational only for a small number of agents and require rather simple environments. Thus, these solutions are inopportun for the focus of this article (complex, dynamic and unexpected environment, with large number of agents).

**Approximations** Since finding optimally matching behavior is intractable and inflexible, another trend of collaboration design focused instead on finding “well-working” collaboration patterns. These patterns being inspired by natural sciences (e.g. ant systems) or by human societies (e.g. norms, organizations, protocols). Norms [2] provide some rules about how individuals should behave, which promote collaboration by forbidding some counter-productive behaviors. Organizations [14] allocates roles and creates obligations between individuals, promoting collaboration in forcing informations to evolve in some efficient way. Protocols [8] determines some interaction patterns, which promotes collaboration by precisely defining how to behave and thus what can be expected from others.

**Abstract versus Concrete Collaboration Specification** There exist two trends of models for building such approximations. The first trend consists in concretely defining restrictions on agent’s behavior. For instance, norms [18] define very clear rules that cannot be violated, organizations [12] specify how agents should behave and who to communicate with. This trend, while allowing to very objectively determine whether a behavior is conform with some collaboration specification, becomes very environmentally dependent and thus inflexible. For instance, “driving on the right” rule fails if a lane gets closed: with such a rule agents which are compliant with the collaboration cannot do anything but crashing into each other.

Another trend consists in abstractly representing constraints. For instance, for norms and organizations [17] this step consists in defining abstract roles with abstract goals and abstract rules (e.g. be sure that the water hose is ready before the fire extinguisher uses it instead of before time $T$). Such a representation is less sensitive to some particular environment and thus more flexible. But, objectively deciding whether some behavior is compliant with some behavior may become impossible. In that case, one may say that this behavior is streamlined with some collaborations constraints. In this article, we aim for flexible and robust collaboration, thus, we pursue that track.
3.2 Preferences

Preferences can be used for representing the relative satisfaction by similar objects. In our specific setting, these objects are system outcomes. Preferences are thus used to represent goals pursued by individuals and the relative satisfaction of situations with regard to these goals. They are ranked by a partial order.

Some research already used preferences for designing single-agent decision systems. For instance, [16] proposes an automatic planner, capable of designing several plans which can be instantiated depending on the selected preference function. [19] investigates the influence of values on agent deliberation using argumentation. These two frameworks differ in the time-span considered for making decisions. [16] estimates the preference of each action by considering the expectations of its long-term consequences, from a planning perspective. Conversely, [19] only estimates immediate consequences (even if immediate consequences can describe expectations about the future, for instance the consequence of the rescue option is “probably being late”). Both approaches are applicable for designing preference systems for MAS. Nonetheless, the one from [19] appears to be more suitable for that purpose, because collective action and uncertainty tend to make long-term planning inefficient. The value based approach of [19] is on a more strategic level and fits better the collaboration situation where things like the expectation of cooperation is more important than the exact protocol that is used in the interaction at hand.

Preferences have also been used in MAS. For instance, [13] uses preferences for modeling desires of negotiating agents. In [13], preferences are used to evaluate the acceptability of bids. However, the assumptions in the negotiation context are very specific and do not hold for most collaboration contexts. In our article, we want to expand the use of preferences for directing agents towards collaboration.

4 Integrating Preferences In Decisions

Including preferences influences how agents make their decisions. Instead of directly being driven towards reaching a concrete goal, agents are driven towards reaching the most satisfactory (or preferred) situation according to their preferences. Based on their preferences they can select a concrete goal that satisfies their preferences best or, in simpler cases, directly select an optimal action according to their preferences. In this paper we concentrate on the simple case and leave the more complex case where goals are chosen and changed based on the preferences for future work. In order to select actions which are more likely to reach the preferred situation, agents require two capabilities. They have to know which situation is most preferable, which is determined by their preference function and they have to be capable of estimating the effects of their actions. These two capabilities can be combined in order to select the action which leads to preferred outcomes.
4.1 Estimating Outcomes

**Representing Outcomes** Action outcomes model the estimated effects of performing an action in a given situation. We talk about estimated effects, because we assume actions might fail, the environment or other agents can interfere, etc. There are several ways to represent and reason with the effects of actions under these circumstances.

For instance, [16] combines a planning approach with preferences, where preferences order the expected final situations. In this approach, which is conceptually similar to MDPs, the outcome of action \( a \) corresponds to the expected satisfaction given by the final situations which would be reached assuming that the agent selects the most satisfactory actions in the future. This representation implies some assumptions for the model. First, there is only one agent acting at the time (or the actions of other agents are seen as integral part of the environment) and that expectations over preferences can be combined, which is possible in their case because preferences are one dimensional, represented by real numbers.

[19, p. 109-136] uses another approach, which consists in estimating “by hand” the effect of performing an action. In the running example, the expected effect of performing the “rescue” action can be that the victim will be rescued, the agent will probably be late and the agent will be near the victim. An important remark is that such representation of effects can describe consequences that can be indirect, only potential or which can arise under specific (additional) conditions. For instance, hiding in the fire truck appears to be “safe” while considering only immediate consequences. But, agents should also consider that this action also leads to an “unsafe” situation if the fire is expected to spread. Thus, agents need to reason about both the expected direct and indirect effects given their situation in order to choose an action.

**Designing Outcome Estimators** Outcome estimators are not easy to model for making concrete decisions, due to possible incompleteness of the environmental model, partial information and collaborative action. If all these elements would be modeled through some uncertainty factors, one would soon reach a point where the effects of actions are completely unknown.

The representation of [19] allows to cut short the search for all possible effects (which is intractable) by heuristically estimating consequences of actions. However, this approach is more suitable for making strategic actions (e.g. whether to collaborate with a specific agent) rather than low-level decisions (e.g. whether to send a request or an order to the other agent). The reason being that the range of outcomes and of possible situations explodes when being concrete, thus making the design of the heuristic intractable. In the example, the expected outcome in the perspective of punctuality for the action “rescue” is “be late for the mission”.

365
4.2 Ordering Preferences

Preferences are used to order effects. For instance, agents can prefer safe situations over unsafe ones, but we also need to order different types of criteria. E.g., preferring being on time versus being safe.

Preferences can capture a large variety of aspects, which correspond to attributes of effects of actions or situations pursued by agents. These aspects can be individual (e.g., preferring to be on time, saving energy), environmental (e.g., number of casualties, number of damaged agents, amount of money lost due to fire) as well as collective (e.g., time spent in order to complete the goal, risks taken by the group). Preference functions, which order these various aspects, can be extremely cumbersome. Thus, efficient representations of user preferences is an important issue and as such has been investigated by [4].

Designing preference functions can be a complex issue due to the combinatorially large space of orders over aspects. Nonetheless, some restrictions on the space of representation allows to represent much more concise preference functions.

One of those restriction, referred to as preference independence, appears to be at least partly applicable in our setting, allowing to reduce design complexity. Most of the preference aspects investigated for designing multi-agent systems can be completely ordered (e.g., lowering the time to completion, all other things being equal, is always better). When this property holds, representing the whole preference function consists of determining how each aspect can be combined with others. In other words, how much of one aspect can be traded off for another. Several solutions are possible, such as prioritizing on one aspect (the safest, the best), “linearly” combining the two (a bit late and safe is as good as being on time and run a bit of trouble) or even preferring situations avoiding an aspect to be too low.

[19] proposes a method to design preference functions using preferentially independent variables, by looking at the issue the other way around. A global preference function can be decomposed into conceptually uncorrelated sub-preferences. For instance, when considering the overall preference function, the system designer can see that safety and efficiency can be estimated independently. Given two situations with the same efficiency, the one with the highest safety is the best; and vice versa with efficiency.

From a design perspective, preference independence has several non-negligible benefits. First, this approach allows to abstract away from ordering low-level outcomes: designers consider how to combine sub-preferences instead of outcomes. Second, sub-preferences may also be decomposable in more detailed (simpler) preference functions (e.g., safety can be represented as the combination of avoiding being burned and avoiding being hit by falling items). Consequently, the preference function can be represented by combining simpler preferences. Third, preferences (and thus sub-preferences) are conceptually independent in design. Consequently, any designed preference can be used as a sub-preference without any additional cost, allowing extensive re-usability of preferences. For instance, “safety” can be used both for evaluating “operation costs” and “system relia-
Fig. 1: Preference decomposition of firefighter agents. Each level of the tree is more concrete than the former. Line thickness represent the relative importance given in the decomposition of sub-preferences. Gray boxed represent cultural preferences.

There exist numerous solutions for representing preferences (e.g. a single node or a deep preference tree). A solution of particular interest for the Section 4.3 consists in decomposing preferences as much as possible such that each sub-preference is conceptually more concrete than its parent. An example of such a decomposition is provided in Figure 1.

4.3 Preferences and Hybrid Agents

Preferences are adequate for driving important decisions which require abstract reasoning, but they are impractical for handling concrete behavior. This concrete behavior is more suitably represented by traditional design solutions (like plain code, BDI, planning, protocols), which are less adapted for integrating abstract drives.

Dividing agent decision processes in different layers of abstraction has received some attention in the past. For instance inteRRaP [15] proposes different reasoning layers with different internal logics (reacting to the environment, planning from a single agent point of view, planning from a group point of view).

In our article, we are interested in a similar approach. We want agents to be capable of making long-term abstract decisions, referred as strategical decisions, from an abstract level. These abstract decisions being highly sensitive to an agent’s culture. Then, this direction can then used to drive more concrete behavior, referred to as the tactical level.

The nature of the connection between the two aspects depends on each implementation and up to our knowledge, no implementation generically cover both aspects. For instance, strategical decisions can be represented at the tactical level by changing of agent goals, activating a module [5], executing some protocol, or
Fig. 2: Decision process for an agent with preference function $p_1$ in situation $d_t$. Bold lines and text highlight choices made by the agent. If $a_1$ uses preference $p_2$, then the selected action is “ignore”

Performing some transition in them. The influence of tactical to strategical decisions can occur during belief updates, goal fulfilment, or a specific procedure state is reached. In addition, the strategical level may also store some memory if required (e.g. by some desired dynamics).

4.4 Running Example

Perspectives In this example, four preferences are considered: safety $p_1$, punctuality $p_2$ and combinations of these two. The whole decision process is illustrated in Figure 2.

Estimating Outcomes If “rescue” tactical action is performed, $a_1$ expects $o_{r1}$: late($a_1$), at(injured.person). If “report” is performed in $d_t$, $a_1$ expects $o_{r2}$: in.time($a_1$), reported($a_1$, person), at(unknown_position). If “report” is performed in $d_t$, $a_1$ expects $o_{r3}$: late($a_1$), at(unknown_position). If “ignore” is performed, $a_1$ expects $o_{i}$: in.time($a_1$), unreported(person), at(fire).

Preference Functions Safety is represented by the following order: situations with the property safe are better than those with the property tiny_risk which are better than those with the property unsafe. safe is true if the agent is far from fire (e.g. rescuing the injured person, thus at(injured.person) is true), tiny_risk is true the agent may have to move to the fire (e.g. when at(unknown_position) is true, for instance when the agent waits for leader instructions) and unsafe if the agent is near a fire (thus at(fire) is true). Punctuality is represented by the following order: situations with in_time($a_1$) are better than those with late($a_1$).

$p_1$ and $p_2$ are combinations of safety and punctuality. $p_1$ compromises punctuality and safety. $p_2$ drastically favors punctuality over safety; for two situations $s_1$ and $s_2$: $s_1$ is better than $s_2$ if, for timeliness $s_1$ is better than $s_2$ or they are incomparable with regard to timeliness and for safety $s_1$ is better than $s_2$.
5 Culture-Based Design

5.1 Inspiration

While our aim is completely unrelated to modeling human features, our idea is inspired by descriptions from social sciences about human cultures [10, 11]. In these studies, cultures are described as collectively shared values (representing what individuals consider important, such as being normal or being rational) and practices (e.g. greeting by bowing or shaking hands). These shared drives represent abstract individual motivations, that are correlated to preferences in our hybrid model. But, apart for practices, that we disregard in this article, cultures do not further influence how individuals make concrete decisions.

These cultural studies also highlight that cultures, through their influence on individual preferences, promote collaboration. See Figure 3 for an illustration of correlations between some cultural features (power distance and uncertainty avoidance) and preferred organizational patterns. Each pattern is known to have a very specific collective performance profile (e.g. bureaucracies are fitter for simple and static environments while adhocracies are fitter for more complex and dynamic environments [14]). In this article, we aim at producing agent models which are capable, as human cultures do, of harmonizing individual drives to improve collective robustness.

In this article, cultures are represented by shared abstract preference functions. In some sense, cultures are some shared strategical preferences. For instance, gray nodes of Figure 1 illustrate the culture of this agent. This preference function has the particular property of being expected to be shared with other agents.

Fig. 3: Culture and preferred organizational pattern, from [11]
5.2 Comparing Cultures and Concrete Approaches

When designing collaboration agents, system designers create some restrictions on possible agent’s behavior. These assumptions restrict the space of admissible agent behaviors but allow in return system designers to make expectation about the system. Nevertheless, introducing restrictions leading to expectations implies to lower designer (and agent) freedom. Thus, these two aspects have to be carefully investigated.

In this article, cultures are such a type of restrictions: agent behavior must be inlined with some assumed culture. But, culture is also used to create assumptions about other agent’s behavior.

Abstract cultures and concrete approaches attack to the same problem from two different angles: abstraction can be achieved by concrete approaches and concreteness can be achieved with culture, both at a very large design cost. In the following, approaches are compared assuming a feasible design: concrete approaches cannot be too abstract while abstract cultures cannot be too concrete.

Restrictions Using culture as a collaboration mechanism implies to fix a part of agent preferences. Agent behavior has to be inlined with their culture, reducing the amount of freedom given to agent designers.

The perspective of abstract cultures creates assumptions (and thus restrictions) on agent abstract drives (e.g. informing the leader about victims is important). This approach differs from concrete approaches, such as protocols, which puts limitations on concrete behavior (e.g. how firefighters should contact fire commander). Compared to BDI agents, goals are concrete instances (e.g. call fire commander) while cultural preferences are abstract (e.g. preferably informing fire commander, relatively to other drives). In other words, cultures put assumptions on what is pursued, while concrete perspectives put assumptions how agents behave.

Expectations One of the consequences of adding assumptions, thus restraining the set of admissible agent behaviors, is that predictions can be made about the system. These predictions, or expectations, can be integrated into agent design for improving collective performance.

Both approaches create different types of assumptions, leading to different types of expectations. Concrete approaches create assumptions about agent behavior. So, possible expectations have to directly result from the direct consequences of these assumptions, allowing some concrete expectations about the behavior of other agents (e.g. if a firefighter do not reply before the timeout, then this firefighter can be assumed to be damaged). They can be used to make local expectations about the environment (e.g. fire commander expects fire to be extinguish if being told so). Nonetheless, expectations about emerging behavior is difficult to find\(^1\), at least without requiring a large amount of assumptions.

\(^1\) This is the topic of complex systems, which are not called “complex” for no reason
Conversely with cultures, they create assumptions on collective drives. These assumptions are more adequate to make expectations about emergent behavior. For instance, consider the assumption that sharing informations about victims is important. Then, from this assumption, it is rather easy to expect that fire commanders have complete information about victims, even if cultures do not describe in detail how agents will communicate with the fire commander.

A similar reasoning is applicable when designing a system the other way around. That is to say, by creating system assumptions in order to drive some individual behavior and emerging patterns.

5.3 Comparing Cultures and Abstract Approaches

The difference for specifying collaboration using culture differs and other abstract approaches studied in the past (e.g. abstract norms, abstract organizations) may be confusing. Conceptually speaking, other abstract approaches are only related to the agent’s environment. Abstract norms provide rules about what to do or not to do. Same abstract organizations provide some structure about who to interact with and about what. Nevertheless, these approaches do not drive agents towards some particular solutions, which is what culture does.

One of the properties of abstract design is that numerous acceptable solutions are possible. Consequently more freedom is left on agent’s freedom on agent’s side, making expectations about which alternative to prefer and thus which alternatives is likely to be preferred by others is difficult. Culture helps in solving this issue in helping the selection of solutions that are inlined with a shared culture. Thus, they are also inlined with each other. For instance, if the system user wants to avoid losing agents, he or she may create a lot of rules for determining when to inform others about unsafe situations and backing up each others. With culture, the designer would just have to state “group safety is very important for agents”. In that case, designers would consider as an important aspect to be planned upon while being free about how to enforce safety (e.g. letting agents informing each other when some dangerous situation may occur and how, giving capabilities to process warning messages about danger, agreeing on what “dangerous” means and so on).

From a design perspective, this method avoids the introduction of unnecessary environment-related information into collaboration specification, increasing the flexibility of the system. In addition, this method avoids to remove freedom for agent designers on challenges faced by their agents, aspects for which designers are the most likely to be expert on (e.g. knowing which situation is dangerous, who to warn and about what).

In general, culture appears to be too abstract for being the only specification for driving collaboration. Nevertheless, culture can used in order to drive some more concrete form of collaboration towards desirable outcome. For instance, a culture where safety is important can easily drive the creation of safety-promoting rules and roles. If the environment changes, rules and roles will change, but not the common drive of avoiding casualties.
5.4 Complementary Collaborations

**Appropriate Use of Complementary Perspectives** The underlying argument conveyed by Section 5.2 is that the both approaches consider the same problem from two opposed perspectives: from abstract to concrete (top-down) and from concrete to abstract (bottom-up). To that extent, these techniques can be combined for efficiently designing hybrid agents: strategical level encompassing cultural assumptions, while tactical level encompassing concrete assumptions. Each of these perspective allowing to create assumptions that are relevant with regard to its level of abstraction. Then, expectations resulting from these assumptions can be used in order to improve the other (e.g. knowing about timeliness can be usable in concrete plans to set sharp deadlines). Another benefit of using two representation is that no representation is misused: concrete approach are not used for making abstract assumptions and vice versa. This appropriate use allow to keep each representation concise and tend to reduce design complexity.

**Different Impact on Performance** These two approach are also differently impacted by technical constraints: concrete approaches are more related to the environment than abstract ones. Consequently, they are more likely to fail due to unexpected events than abstract approaches, which abstract away from technical details. Consequently, abstract cultures are more robust and flexible than concrete approaches. Nonetheless, they are more limited than concrete approaches to tackle low-level interactions, which are crucial to maximize efficiency.

6 Artificial Cultures: Socio-Inspired Preferences

Artificial cultures are inspired by concrete culture in the sense that both capture collective preferences. Former social science discovered the presence of such shared collective preferences amongst human cultures referred as **cultural dimensions**. While we do not claim to build faithful model of those, some inspiration can be drawn from them for designing artificial cultures.

Each dimension evaluates cultural response to some dilemma, like “What is more important, rules or relationships?” [10]. Such a dilemma if relevant for modelling problem-solving MAS\(^2\) can be integrated as a preference aspect. On the track of linking culture and collective behavior, [20] conceptualizes links between cultural dimensions, individual behavior, emerging collective behavior and performance.

In the rest of this section, we briefly introduces cultural dimensions from [10, 11] that can be considered when designing preference functions for collaborative MAS.

\(^2\) Several cultural dimensions focuses on aspects without immediate relevance for computer systems, such as indulgence vs. restraint; or neutral vs. emotional.
6.1 Power Distance (PDI)

[11] describes the cultural importance given to individual statuses. In high PDI, subordinates prefer to give information and decision power to leaders. Reciprocally, leaders are expected to assign missions to subordinates. As a result, such preferences tend to bring about the property that leaders (and mostly them) have the most accurate strategical information and can thus make the best-informed decisions. In addition, due to subordinate higher obedience, leaders can further optimize subordinate schedules, increasing efficiency. High PDI lowers system robustness: leaders are bottlenecks (in particular in information-rich environments) and missing leaders collapse the whole communication and decision structures. In the running example, high PDI agents are likely to “report” to make sure that leaders have informations or to skip, if the leader is assumed to have this informations.

In low PDI, individuals consider themselves as independent and of equal value with regard to information and decisions. They are likely to take more initiative and carry their own tasks. As a result, individuals have locally more information. Such a culture is likely to increase system robustness, since no agent is critical to the system but efficiency is expected to be lower, because of more difficulties for obtaining information or synchronizing groups of agents.

6.2 Uncertainty Avoidance (UAI)


In high UAI, individuals prefer situations where their beliefs are expected to be coherent with reality (uncertainty). To this extent, they try to lower this uncertainty either by getting more information or by making assumptions about it (e.g. someone will support me when I will enter the burning house). As a result, individuals prefer to behave according to standards, further reducing uncertainties for itself as well as for others. Thus, as a an emerging property, individuals can expect less variability in other individual actions and environmental states, further increasing the interest of making assumptions. High UAI is very efficient for static environment because a lot of assumptions can be made about the environment as well as optimizing collective action. Nonetheless, this preference is not flexible: if the environment is dynamic, either agents constantly update their procedures or they may try to apply mis-adapted procedures. In the example, the decision depends on uncertainties generated by each option: ignoring the victim can lead to casualty; while rescuing may prevent the agent to be at the fireplace while being expected to be there, creating uncertainty for others.

In low UAI, individuals are less sensitive to uncertainty. Their behavior is more directed by goals than by procedures. To that extent, behaviors are likely to be more adaptive, leading to more variability in environmental situation. Task resolution variability is not a collective issue since other agents expect this uncertainty and thus to adopt adaptive solutions. This adaptability tends to raise collective flexibility at the expand of possible standardization which leads to lower efficiency.
6.3 Sequential versus Synchronous time

[10] describes two paradigms to consider time management: sequential and synchronous.

In sequential time, individuals consider time as a sequence of events. Respecting deadlines is very important to not delay this time line. As consequence, timeliness is expected from other individuals. From a collective perspective deadlines and schedules are expected to be more reliable. This preferences are likely to improve efficiency and lower time to completion in allowing accurate planning of tight schedules. But, this approach fails when time considerations cannot be estimated accurately (lower flexibility) and is sensitive to failures, missing agents and congestion (lower robustness). In the example sequential agents can choose between “report” and “ignore” in \( d_t \) and always “ignore” in \( d_{\bar{t}} \).

In synchronous time, time is considered as a resource to be planned against. To that extent, individuals prefer to locally maximize their efficiency, for instance by taking opportunities. With this consideration of time, timeliness is less important, so individuals tend to be late. Other individuals can expect delays and thus can, for instance, prepare activities for filling waiting time. This form of time management can also lead to high efficiency, if the environment is suitable for “filling in” waiting time. A negative point concerns the unpredictability of time to completion: an agent can continuously delay a task because of getting opportunities to perform other tasks more efficiently. In the example, synchronous agents select between the three options, comparing the time cost incurred by selecting one of the other option (time for extinguishing a wider fire if “help” and time for getting back and rescuing for “ignore”, estimated cost for sending someone else rescuing for “report”).

7 Conclusion

This article proposes a solution for combining two levels of collaboration into agent decisions: abstract and concrete collaboration. For this purpose, we propose a model of hybrid agents, capable of reasoning both at an abstract (strategic) and concrete (tactic) level. This strategical level, represented by preferences, provides abstract directions to a tactical level. The tactical level, represented by traditional agent decision process (protocol, BDI), turns these directions into concrete action. In return, the tactical level provides the strategical level with relevant information to revise strategical action.

Collaboration is integrated in these two levels, by creating assumptions for each level. Assumptions of the tactical level describe in detail how individuals should behave (e.g. some interaction protocols). Assumptions of the strategical level abstractly describe which drives are pursued by agents, what they want to achieve, referred as an “artificial culture” as a metaphor to human cultures. These two types of assumptions form two types of collaboration: a concrete one and an abstract one.

Each level of collaboration is adapted to describe different aspects of the system. For instance, individual precise behavior is captured by concrete col-
laboration while collective patterns to be expected are captured by abstract cultures. In addition, each level of collaboration has different influence on collective performance: abstract collaboration promotes robustness and flexibility while concrete collaboration, which is more related to the environment, promotes efficiency.

Culture proposes some new form of collaboration which consists in collectively driving agents towards some very abstract course of action. This collaboration is on purpose the least correlated possible from the environment, providing with maximal robustness and flexibility. Nevertheless, by being too abstract, this form of correlation is not suitable in itself to actually provide the very concrete elements required by agents to interact on a concrete base. But, cultures do provide some directions for streamlining agent actions towards similar situations. By giving a similar emphasis on what is important, agents can easily find collaborative concrete solutions without having to debate or better, making exact expectations about what is important for others (e.g. an agent in a culture promoting safety can expect support from the others). Thus, culture is appropriate for designing MAS for wide or evolving environments (e.g. exploration, building networks of sensors). In addition, culture allows the independent design of agents, making this approach appropriate for open environments.

So, cultures are appropriate for three categories of applications, in particular if agents are designed independently, like because of required expertise for designing them:

- unknown environments (e.g. exploration, building dynamic sensor networks). In such an environment, agent designers cannot easily determine beforehand some patterns of collaboration. Agents should rather do it on the fly, depending on the situation. Nevertheless, they should not discarding preferences of designers in order to provide appropriate results (e.g. group preservation is relatively more important than efficiency).

- adversarial environments (e.g. military applications, game-oriented applications). In such an environment, other approaches are risky because they tend to force some (collaborative) behavior, which puts the system at risk of being exploited (e.g. raise an emergency call to attract all the drones around). Instead, versatile and adaptive behavior is preferable but promoting collaboration remains also important. Cultures provide collective drives which can be used as a basis for making decisions without explicitly forbidding some behavior (e.g. rescuing damaged drones is important but is not a “must do” rules. If rescuing drones appear to actually lead to drone loss, the decision to rescue can be changed).

- human-machine interactions (e.g. health-care robots, serious gaming). Making accurate predictions about human behavior is a complex topic, there are always exceptions to be found. The current standard solution to cope with this uncertainty consists in restricting human users freedom (e.g. standardized forms, limited number of functions of a system). Instead, a culture allows to promote some type of collective objectives (e.g. high-quality care is relatively more important than costs) without restricting agent’s behaviors.
Thus, agents can keep the global direction while being capable of adapting in numerous settings (e.g. a robot-nurse which would work both in a doctor’s house in a large hospital. Each institution local procedures can be learned on the spot).

For future work, we plan to implement a prototype of the hybrid agent, combining a BDI representation like 2APL for the tactical layer, while the representation of the strategical layer is still under consideration. We expect this prototype to be ready to be shown for the workshop. Then, we plan to use this hybrid agent on a concrete problem, allowing to confront technical issues raised by reality. This confrontation will allow us to get knowledge about methodologies relevant for the design of such agent and tools that can support the promotion of collaboration, both at concrete and abstract levels.

References

The Interaction as an Integration Component for the JaCaMo Platform

Maicon R. Zatelli, Jomi F. Hübner

Department of Automation and Systems Engineering
Federal University of Santa Catarina (UFSC) – Florianópolis, SC – Brazil
xsplyter@gmail.com, jomi.hubner@ufsc.br

Abstract. Interaction is a subject widely investigated in multi-agent systems (MASs), but there are still some open issues. While most of current approaches of interaction in MAS just consider the interaction between agents, some problems are better modeled when the MAS is composed of agents, environment, interaction, and organization. In our approach, we integrate the interaction with the other MAS components, like the organization and the environment, keeping it as a first class abstraction. In this paper we present a conceptual model for the interaction component, a programming language to specify the interaction, and how our approach was integrated in an MAS platform. The main result of this paper is the conception of the interaction also as a first class abstraction considering an MAS composed of agents, environment, interaction, and organization.

1 Introduction

It is quite common in MAS that the agents need to interact to achieve their goals. Sometimes an MAS can be composed of Agent, Environment, Interaction, and Organization as introduced in [15, 23]. In this kind of MAS, the interaction does not concern only the agents, it is strongly related to the environment and the organization of the system. For instance, besides interacting directly with other agents, agents also interact (act and sense) with objects in its environment.

It already exists many works about agents, organization, and environment. There are tools to specify, develop, and execute each of these components. For example, an MAS developer is able to build the environment by means of CARtAgO [39], the organization by means of AGR [21], ISLANDER [20], Moise [28], and so forth, and finally, the agents by means of GOAL [24], JADE [11], 2APL [13], Jason [10], and so on. There are also tools to link these components to work together, such as EIS [5] and JaCaMo [9]. This separation of concerns can improve the maintenance, modularity, organization, reuse of code, etc. It is also easy to see that each of these components can be programmed by different developers, which also facilitates the division of tasks.

In addition, there are several approaches that defend the idea of keeping the interaction as a first class abstraction [14, 31–33, 42, 43]. However, none of the current works provide us features to specify and execute the interaction considering the existence of the other MAS components, that is, to allow the specification, development, and execution of the interaction not only considering agents, but also considering the environment and the organization.
We already introduced a conceptual model and a programming language for the interaction considering the other MAS components in previous works [48, 49]. In this paper we focus on the integration of the interaction with the JaCaMo platform. JaCaMo is a project that allows the developer to consider each one of the MAS components as first class abstractions. Although the agent, environment, and organization components are already considered by this platform, the interaction component was not properly integrated. In this platform, the interaction is not a first class abstraction, it is simply reduced to messages coded inside the agents program. For instance, it is not easy to find in the system code how the interaction is programmed (it is indeed spread in several agent programs).

The aim of our whole work about interaction (conceptual model, programming language, and integration with JaCaMo) is to provide a mechanism to institutionalize how the agents may interact with the different elements in an MAS to achieve the organizational goals. We are linking the organization (e.g. its goals) to the agents (that should fulfill them) and to the environment (by defining interaction protocols that could be used as guidelines for the achievement of the goals). By considering the interaction with the environment, we can formalize more general situations in a protocol, where the agents should interact with the environment by means of performing actions and perceiving changes. We are looking for an interaction component that is able to deal with the other three MAS components. It means that we are considering a more complex MAS, composed of Agent, Environment, Interaction, and Organization.

The paper is organized as follows. Section 2 briefly presents the state-of-art about interaction when more MAS components are considered. Section 3 presents our conceptual model of interaction. Section 4 presents the programming language to specify interaction protocols following the interaction model. Section 5 presents the integration of the interaction model into the JaCaMo platform. Finally, before conclusion (section 7), we discuss some results (section 6).

2 Related Work

In this section, we present the interaction problematic and some related work. We start with the works focused on interaction between agents, followed by those that consider the interaction with the environment, and in the following, the works that regard the interaction with the organization. Finishing this section, we mention some works that have already introduced the interaction problematic considering the integration with the three other components.

2.1 Interaction - Agent

There are several drawbacks of specifying the interaction inside of the agents code [19, 31, 43]. One of them is related to the maintainability of the system. If the interaction specification is modified, it is necessary to update the code of each agent involved. Another one is related to the protocol composition. The protocols could not be composed at run-time in order to allow more complex interactions.
As pointed by some approaches, it is unnecessary to keep the interaction control inside the agents code [30–32, 34, 43]. The separation of the two issues simplifies the development of applications, leading to a modular approach [22]. Consequently, protocols can be used to compose more complex protocols [12, 16, 17, 26, 36, 37]. In [26, 37, 43], it is presented other advantages of a modular approach such as the specification of reusable protocols, the improvements in the validation process, and the capacity to share protocols between agents at run-time.

2.2 Interaction - Environment

One of the main limitation in most of works is to regard the interaction only by means of message exchange between agents, not considering the agent interaction with the environment [2, 3, 6]. Some examples that justify this kind of interaction are presented in [2, 3]. One of these examples refers to the election in the human world. When people have to do an election, they do not say the candidate name. They use their hands to interact with the electronic ballot box or simply raise them without saying any word. On the one hand, the electronic ballot box is responsible for computing the votes and notify the winner. On the other hand, by raising their hands, people also may discover the winner of some election only by counting the upper hands. In both cases, the interaction occurs by actions and percepts in the environment and not by speech acts.

There are some works that consider the relation between interaction and environment. In [38] and [41], it is presented a model that allows some different kinds of interaction, called overhearing, or eavesdropping. In this interaction kind, the agent intercept messages of others by using the environment. The environment is a way to send and receive messages. In [29], the aim is to conceive an environment as a way to allow indirect interaction. Their focus is on interactions like stigmergy, which are interactions used by several natural systems such as amoebae and ants. Finally, in [4, 40] the authors use artifacts to handle the interaction between the agents. In [40], the aim is to provide a communication infrastructure based on artifacts. The implementation of such infrastructure is done in JaCaMo platform [9] and the authors provide the representation of two kinds of artifacts. The former has the aim to represent the interaction protocol itself and allows the specification of a sequence of messages. The latter defines each speech act individually. In [4], the authors use CArTAgO artifacts to embed commitment-protocols following the model introduced in [45–47]. Their work also enrich the JADE [7] with mechanisms to exploit the use of commitments and protocols based on commitments. Each artifact keeps a social state, which is composed of social facts and commitments. Thus, the agents are able to reason about the interaction by means of observing the social state evolution. In both cases [4, 40], instead of the agents exchange messages directly, they use the operations provided by the artifacts. For example, in the contract-net protocol, the operations of the artifact can be cfp, propose, refuse, accept, reject, done, and failure. Moreover, the communication artifacts have the aim to notify the receiver about the messages.
2.3 Interaction - Organization

The relation between interaction and organization is also important. The GAIA methodology [44], for instance, has already defined a role as a composition of four main attributes: responsibilities, permissions, activities, and protocols. The protocols are responsible for specifying the interaction between the agents that are playing the organizational roles.

Some works about organization already relate the interaction with the organization by means of a dialogical dimension [8, 18, 20, 21, 27]. In this case, they use several organization concepts, like goals, roles, and obligations. Each of these concepts are strongly connected with the interaction concepts.

2.4 Integration with the Three Components

Some works already have an initial integration between interaction and the other three components [2, 3, 14, 30, 34, 42], but their aim is different than ours. In [34], although the environment is considered, it is a simple mediator between agents and not a proper first class abstraction. Another existing limitation in this work is that it does not have an integration with an organization model. A role, for example, while existing inside a protocol, may not exist in the organization. As a consequence, the specification may lack coherence since different role conceptualization may exist in different components. In [30], the environment is considered by another perspective: the agents could recognize other agents by the concept of neighborhood. The agents are only able to communicate with others depending on how far they are from each other. As [34], it does not consider the actions or percepts performed by the agents in the environment, and the organization component is rather simple (only role names are considered).

The MERCURIO framework [2, 3], a very similar work to ours, focus on integration of the interaction model regarding agents and environment. The environment considers the actions performed by the agents and the percepts that the agents may sense. However, since the main aim of MERCURIO is to deploy the interaction with the environment, the interaction is not strongly connected with the organization. The roles in the interaction, for example, are not the same roles as in the organization. The existence of the other organizational concepts is not considered either.

In contrast to the previous works, MAS-ML [42] and O-MaSE [14] are a modeling language and methodology, respectively, which consider the interaction integration with the three other components. However, both approaches are conceived for the specification phase, not regarding the implementation and execution phases. In addition, even providing tools to generate code, they do not generate the interaction code.

We noticed the lack of proposals that regard the interaction integration with the three other components. Moreover, in some of them, the interaction specification is conceived to be handled by humans during the MAS design and does not allow the agents to read it (or eventually to change it) at run-time. Although some authors are concerned with the interaction between agents and some of the other components, none of them integrate the interaction with the three components in a unified perspective.
3 Conceptual Model

This section briefly presents how the several MAS components are conceptually integrated with the interaction. Only the core ideas of the model are described here. More details can be found in [48].

Fig. 1 shows the four MAS components and the relations between the interaction and the others. In order to keep the figure clear and clean, we only show the concepts that were directly related to the interaction. The most important concept in our model is the interaction protocol, which is basically composed of a set of participants, transitions, states, and goals. Each transition links two states (one source state and one target state) and it can be fired by an event, a message, or an action. When some transition is fired, a new state is achieved and the protocol execution makes progress. In order to separate the protocol specification and the protocol execution, we call scene an instance of a protocol. It is possible for a protocol to have several scenes executing at the same time.

The organizational concepts used in our model (top of Fig. 1) are based on the organizational models presented in [18, 28]. The interaction is related to organization in four points. Firstly, the protocols are related to organizational goals. A protocol specifies a possible interaction scheme to achieve them. When a protocol finishes successfully, the organizational goal is considered achieved. For example, if there is an organizational goal for an agent to contract a company to build a house, such goal can be achieved by the use of a contract-net protocol. The protocol is just one (and not the only or
even a *mandatory*) way for the agents to achieve the organizational goals. It can exists several protocols to achieve the same goal and the agents could also achieve a goal using other means. We could also imagine the existence of protocols without a relation to organizational goals, however, in this work, our main objective with the use of protocols is to help the agents to achieve the organizational goals. Thus, we are not interested in the representation of protocols that do not drive the agent to accomplish organizational goals and neither about what the agents do for achieving their own (not organizational) goals.

The second organizational concept used in our model is obligations. The transitions of a protocol are related to organizational obligations. Obligations are created for the agents to perform the action that fires some enabled transition of the scene and thus evolve its execution. For example, if there is a transition in a protocol that specifies that some agent needs to tell the price of a product to another agent, an obligation with this information will be created as soon as the transition is enabled. Thirdly, the participants of a protocol are related to organizational roles. To be a participant in a protocol, an agent must previously play a role in the organization (e.g. the role baker, manager). Since the organization constraints the role adoption based on the agent skills, the agent will be able to perform the activities required as a participant in the protocol. Finally, the organization also provides operations, which are the actions that some agent can perform in the organization such as adopt or leave some role, commit to some mission or goal, and achieve some goal.

The environment concepts used in our model (bottom of Fig. 1) are based on the A&A meta-model introduced in [35]. We map the concept of artifact onto a participant in the interaction component, which constrains the participation of artifacts in the protocol; the operations, which represent the actions that the agents can perform in the environment (for example, the agent can execute actions to regulate the temperature of an oven, such as turn the oven on or off); and finally, the observable events, which agents can perceive in the environment, such as an alarm indicating that the temperature of an oven is too high, the color of something, the sound of a machine, etc. It is important to notice that the artifacts are not an autonomous entity and, in our approach, we are not trying to define what the artifacts should do. Rather, the protocol defines which actions the agents should do on them. Besides the actions, the use of protocols is a way to handle the observable events that are being produced by the artifacts.

The agent component (right side of Fig. 1) provides the concepts of action, which can be some action performed in the environment or in the organization, and the message exchange, which represents the use of communicative acts (e.g. tell/inform, achieve) in order to interact with the other agents. The actions that the agents perform in the environment or in the organization are mapped onto their respective concepts in their respective components. An action performed by the agent in the organization is mapped onto the concept of action in the organization component while an action performed by the agent in the environment is mapped onto the concept of action in the environment component. Finally, the concept of message exchange is directly mapped onto the concept of message in the interaction component.

The conceptual model introduced in this section is a generic solution for the integration of the organizations, environments, and agents based on the concepts depicted
in Fig. 1. For example, if the organization provides concepts like goals, roles, and obligations, it can fit very well with the proposed model. Moreover, the model can also be adapted to other organizations, environments, or agents. One of the core ideas of this paper is to take advantage of using a formal representation of the interaction considering the environment and the organization. A well-detailed protocol (specified by means of messages, actions, and events) can help the development of open systems or help the agents that do not know how to achieve some goal. Thus, the protocols are used to define a more general behavior for a system and not simply to define the behavior of the agents using message exchange.

4 A Language to Specify Interaction Protocols

In this section, we map the concepts presented in Fig. 1 onto a programming language used to specify interaction protocols\(^1\). The language is mostly presented by means of two examples. The aim of the first example is to provide a typical sequence of steps to write a protocol in our approach. For this first example, we consider a simplified situation where an agent must make a cake. The protocol shows especially how an agent interact with the environment by means of actions and percepts. The second example provides more features of the language, such as the specification of message exchanges and timeouts. In both examples, we present very simple situations, however the real advantages of the proposed interaction protocols are better noticed in large MAS, where the system is composed of hundreds of agents with complex tasks and interactions.

The first step to build a protocol with the proposed language is to decide which organizational goals the protocol must achieve. For example, to make a cake for a bakery organization, we can conceive a protocol as a way to achieve that goal (to make a cake). When the cake is done, the goal “to make a cake” can be set as achieved too.

In the following, we need to decide who will be the participants of the protocol. Using the example of the cake, we can assume that in the bakery organization there is a role baker that is the responsible for the cake production, therefore we can define the baker as a participant of the protocol. In addition, we need to include some environment elements that will participate of this scenario. For example, we will need an oven, a blender, a clock, etc.

Then we need to specify the states of the protocol and the order that they should be achieved. The states of a protocol can be achieved by means of transitions that can be fired by actions that the agents perform in the environment, events that the agents can perceive, and messages that the agents can exchange. Back to the making a cake scenario we can see some transitions. In summarize, we can define as a first transition the agent with the role baker needs to mix the ingredients using the blender. In the second transition, the baker needs to put the cake into the oven and finally it needs to set the clock with the required time. After the time elapsed, the clock emits a sound, which can notify the baker to take the cake out of the oven.

Finally, we can define a name, some description, the initial state, and the final states. Notice that we can have several final states, however we can have just one initial state.

\(^1\) We will only briefly present the most important parts of the language, since more details can be found in [49].
Code 1 Making a cake protocol.

```plaintext
1. protocol making_a_cake {
2.   description: "Tell the agent how to make a cake";
3.   goals: "to_make_a_cake";
4.   participants:
5.     agBaker agent "baker";
6.     artBlender artifact "artifacts.Blender";
7.     artOven artifact "artifacts.Oven";
8.     artClock artifact "artifacts.Clock";
9.   states:
10. n1 initial; n2; n3; n4; n5; n6 final;
11. transitions:
12. n1 - n2 # agBaker -- action "mixIngredients" -> artBlender;
13. n2 - n3 # agBaker -- action "putCake" -> artOven;
14. n3 - n4 # agBaker -- action "setTimer" -> artClock;
15. n4 - n5 # artClock -- event "alarm" -> agBaker;
16. n5 - n6 # agBaker -- action "takeCake" -> artOven;
17. }
```

In the making a cake scenario, we can set as the initial state, the first state when the agent starts the protocol, when there is "nothing" of the cake. As a final state, we can set the state after the agent takes the cake out of the oven. Therefore, when this final state is achieved, the goal to make a cake is achieved in the organization. A possible implementation of this protocol is presented in Code 1.

The advantage of using protocols in the case of the making a cake scenario is the openness. A new agent, which has never made a cake before, can adopt the role baker and follow the protocol specification. The protocol is a way to guide the agent to make the cake. Therefore, we can replace the agents and if they know how to follow protocols, they can make a cake easily. Another aspect of this example is the fact that we just used actions and events, such as put the cake into the oven, take the cake out of the oven, set the time in the clock, and the sound emitted by the clock. Both actions and events are related to environmental concepts. Although the transitions in our example represent macro-tasks, we could detail the protocol as much as we need. For example, the transition n5 - n6 could be detailed using other actions. Instead of simply taking the cake out of the oven, we could specify that the agent should turn the oven off, open the oven door, take the cake out of the oven, and close the oven door.

Code 2 presents another example of protocol, where the aim is to serve a customer in a store and the sellers do an election in order to decide which one will serve the customer. The participation of the agents is defined in line 5 and 6, which state that they must play the role customer (line 5) or the role seller (line 6) in the organization. The protocol also includes the participation of a ballot box artifact to help the agents to vote in an anonymous approach (line 7).

The protocol is composed of four states (line 8): k1, k2, k3, and k4, where k1 is the initial state and k4 is the final state. On the one hand, the available transition from state k1 is defined in line 10. It defines that the agent who is playing the participant playerCustomer must send a message to the agents who are playing the participant playerSeller informing them that it needs some seller. On the other hand, the available transitions from state k2 are those defined in lines 11 and 12. The former can
Code 2 Attending protocol.

1. protocol attending {
2.  description: "Serve a customer";
3.  goals: "chooseSeller";
4.  participants:
5.  playerCustomer agent "customer";
6.  playerSeller agent "seller" all;
7.  artBallotBox artifact "artifacts.BallotBox";
8.  states: k1 initial; k2; k3; k4 final;
9.  transitions:
10.  k1 - k2 # playerCustomer -- message[tell] "needSeller" -> playerSeller;
11.  k2 - k3 # playerSeller -- action "vote(X)" -> artBallotBox :
12.  k2 - k3 # timeout 30000;
13.  k3 - k4 # artBallotBox -- event "winner(Y)" -> playerSeller;
14. }

be triggered only by agents participating as playerSeller in the protocol by doing
the action vote(X) on the artifact artBallotBox (the ballot box). Moreover, when
the protocol is at the state k2 an obligation to perform the action vote(X) is created
for the agents playing playerSeller. Although created from a fact in the interaction
component, this obligation exists in the organizational component of the MAS.

An important mechanism used in the language is the unification, which is equiva-
 lent with the traditional unification mechanism of several agent languages and also
Prolog. When an agent performs the action vote or the environment produces the
event winner, it must unify with their respective expressions vote(X) and winner(Y),
where X and Y are variables. Notice that in transition k2 - k3 we have specified the
test ".string(X) & .is_agent(X)" that means when the agent performs the action
vote(X), the X must be a string and an agent in the MAS. Moreover, it is important to
notice that this expression is a string, which means we can have many ways to evaluate
some action. More details about this mechanism is explained afterwards.

Notice that a transition between k2 and k3 is defined with a timeout (line 12). The
timeout is important in situations where temporal constraints are fundamental, such as
the time that an agent must wait for the proposals of the others in an auction. Finally, the
last transition (line 13) of the protocol defines that the participant artBallotBox count
the votes and emit an observable event named winner(Y), where Y is the winner name.
With the successful termination of the protocol, the goal chooseSeller is achieved in
the organization (line 3).

It is also possible to specify different ways to fire transitions. Fig. 2 presents the
language grammar with its non-terminal symbols. The non-terminal duty defines what
must happen to fire the transitions and each transition may have several different ver-
ifications (represented by the non-terminal trigger) to make sure whether the occurrence
is valid to fire it. For example, in Code 3, we specified part of the contract-net
protocol. In this part, the agents playing the participant seller must answer the call-
for-proposals (replyCFP(CNPId)) sent by the agent playing the participant client.
The triggers define the two possible answers that the agents could use to fire the transition
no2 - no3. The former indicates that the seller could refuse to make a proposal
(refuse(CNPId)), while the latter indicates that the seller could send a proposal
Fig. 2: Language grammar [49].

Code 3 Reply to call-for-proposals in the contract-net protocol.

\begin{verbatim}
1. no2 - no3 # seller -- message[tell] "replyCFP(CNPId)" -> client
   trigger "refuse(CNPId)" : ".number(CNPId)"
   trigger "propose(CNPId,Offer)" : ".number(CNPId) & .number(Offer)"
\end{verbatim}

(propose(CNPId,Offer)). In the previous protocols, presented in Code 1 and Code 2, we do not have such kind of situation because for each transition there was only one way to fire it. However, as presented in Code 3, we can represent transitions that could be fired using other ways.

The non-terminal \texttt{trigger} is composed of an expression to evaluate the occurrence pattern (represented by the non-terminal \texttt{pattern}) and an expression to evaluate the occurrence content (represented by the non-terminal \texttt{content}). For example, in Code 3, the pattern is represented by "\texttt{refuse(CNPId)}" and "\texttt{propose(CNPId,Offer)}", while the evaluation of the content is represented by ".number(CNPId)" and ".number(CNPId) & .number(Offer)", respectively. If the occurrence satisfies the pattern, then we can evaluate the content of the variables (if there are variables in the pattern).

If the pattern is omitted, the expression defined in the non-terminal \texttt{duty} will be considered as the pattern. For example, the pattern is omitted in the case of the
protocols presented in Code 1 and Code 2. Consider the action \(\text{vote}(X)\) presented in Code 2. The agent receives this obligation and it has to perform the action \(\text{vote}\). As the pattern is omitted, the expression specified in the duty \(\text{vote}(X)\) is used as the pattern. Next to the symbol : (line 11), it is defined the expression to evaluate the content of the action. Suppose the agent tries to execute something like \(\text{vote}("Ana", 22)\). This action is not valid because it does not unify with the pattern \(\text{vote}(X)\), then the action is discarded. However, suppose that the agent performs the action \(\text{vote}(22)\). This action follows the pattern because it unifies the pattern (with \(X = 22\)), however the action is invalid because 22 is not a String, as required by the content. Finally, suppose the agent tries to execute the action \(\text{vote}("Ana")\). We have \(X = "Ana"\) and "Ana" is a String. In the case where Ana is also an agent, the action is valid to fire the transition.

Other features of the language are the composition of protocols and the cardinality. The composition is made by using the \texttt{import} directive. The \texttt{import} directive needs the information about the address of the sub-protocol and a mapping between the participants of the protocol and the sub-protocol. The mapping is necessary because, sometimes, the protocols may not have the same participants. An example of composition is presented in Code 4. In this case, the transition \(y2 - y3\) will be fired after the \\

\begin{verbatim}
1. y2 - y3 # import "election.ptl"
   mapping {
     employee elector;
   };
\end{verbatim}

\end{verbatim}


election protocol be accomplished. The mapping in this protocol is made by defining that the participant \texttt{employee} will be the participant \texttt{elector} in the election protocol. Although the election protocol needs a goal related to it, during the composition its goal will be ignored. Only the goals related to the main protocol will be used at run-time. The goal in the election protocol is necessary to avoid the agents to instantiate the election protocol itself to achieve no goal. The flexibility to allow protocols without goals and completely disconnected from the organization remains as future works.

The language also provides two different kinds of cardinality: the participant cardinality and the transition cardinality. The former is related to the number of necessary entities to play some participant in the protocol. The latter is related to the number of entities that are necessary to perform the duty specified in some transition. For example, we can have several attendants in a call-center, however we just need one to answer the phone. In an election, we have electors and it is necessary that all of them participate. Therefore, with cardinality mechanisms we can define these situations. Such features are presented in more details in [49].

5 Integrating with JaCaMo

The main aim of the integration of our interaction approach with JaCaMo is to provide an MAS programming platform supporting concerns separation also considering
the interaction\textsuperscript{2}. Fig. 3 shows a general idea of the integration. In JaCaMo platform, the MAS developer can already program each of its three components separately and each component can be programmed with specific tools/languages. The organization can be programmed by using Moise, the agents can be programmed by using Jason, and the environment can be programmed by using CArtAgO. In our work, we also enriched the JaCaMo platform with the interaction component, which also has its proper tool/language. The next two sections detail how the integration was made.

5.1 Mapping the Conceptual Model onto JaCaMo Platform

In order to integrate our approach into JaCaMo platform, we mapped the model presented in Fig. 1 onto the JaCaMo platform. Since the components of agent, organization, and environment in JaCaMo already use the same concepts, we need to integrate the relations between the interaction component and the other ones. As part of the integration, we introduce an interaction artifact (\texttt{SceneArtifact}), which allows the agents to work with the interaction component. A similar integration was already done with the organization by means of ORA4MAS artifacts [25].

Basically, when the agent receives an organizational obligation to achieve some organizational goal, it can verify which protocol it can use to help it accomplish the goal. The agent can instantiate the protocol, informing its specification. Each instance of a protocol is executed in a different instance of the \texttt{SceneArtifact}, which allows the agent to follow the execution of each scene individually. The \texttt{SceneArtifact} reads

\textsuperscript{2} The full implementation of our approach can be found at https://sourceforge.net/projects/intmas/.
Handling the organizational obligations created by the scene artifact.

Code 5

1. +obligation(MyName, _Scene, transition(_CurrentState, _GotoState, _TriggerType, _Target, Duty), _Deadline):
2. .my_name(MyName)
3. <- !Duty.

The relation between the protocol and the organizational goal (Fig. 1) is reified by using a link between the artifact SceneArtifact and the artifact SchemeBoard of the organization. The artifact SchemeBoard is the responsible to deal with the organization goals in the organizational component of JaCaMo. Therefore, when the SceneArtifact achieves the final state of a protocol, it changes the state of the goals related to the protocol in the organization by means of that link. The agent does not need to update the goal state in the organization after accomplishing the protocol execution.

An important part of our approach is the use of obligations, represented by the relation between transition and obligation (Fig. 1). Everytime the scene achieves a new state, new obligations are created to help the agents to accomplish the protocol. For example, suppose the protocol presented in Code 2. After the scene starts the execution, the enabled state is k1 and an obligation related to the transition k1 - k2 is created. This obligation defines that the agent playing the participant playerCustomer should send a message needSeller, using the performative tell, to the agents playing the participant playerSeller. After the agent sends the messages, the obligation is accomplished and the scene moves from state k1 to k2. As a consequence, new obligations will be created. In this case, it will be created an obligation related to the transition k2 - k3 for the agents playing the participant playerSeller to perform the action vote(X) on the artifact that is playing the participant artBallotBox. In addition, this new obligation will have a timeout of 30000 milliseconds, as defined in line 12.

The agents in JaCaMo already knows how to handle organizational obligations because it is a concept already used in Moise. Thus, it is not necessary to build any new specific mechanism for the agents to work with the obligations created by the interaction component. The main advantage of using obligations is that they are created at run-time, which also means that the protocols can be updated at run-time. For example, if the order of the transitions is modified in the protocol, the next obligations will be created respecting the new order of the transitions. It is possible because the obligations are only created as soon as the scene moves from one state to another. Therefore, the agents code usually does not need to be modified if the protocol is modified, since the agents simply follow the obligations.

The Jason code presented in Code 5 illustrates how the agents can deal with the obligations created by the interaction component. In line 1, it is indicated that the agents perceive an obligation to do a duty in such moment of the scene execution. That duty must be done in order to fire the enabled transition. As soon as the agents perceive that obligation, they create a new goal to accomplish that duty (line 4). Notice that it is just the protocol specification and convert it in several observable properties to guide the agent during the scene.
necessary to add the code presented in Code 5 in the agents program to make the agents able to create their own goals to accomplish the duties of the protocol. If the protocol is modified, other obligations for the agents are created and the agents will be able to continue following the protocol in the same way.

Fig. 4 shows the interface of the SceneArtifact, such as its operations and observable properties. Its operations allow the agents to play some participant of the scene (joinScene), to leave the scene (leaveScene), add or remove artifacts of the scene (addArtifact and removeArtifact, respectively), and to start (start), stop (stop), or continue (goOn) the scene execution. Moreover, by means of observable properties, the agents can get some information about the scene. For example, they can see the current state of the scene (Current State), the enabled transitions (by means of the Current State property), their obligations (Obligations), the entities that are playing the participants (Entities), the protocol specification (Specification), etc.

Since there is the concept of links in CArtAgO, which allows the representation of “operations” that can be accessed by other artifacts, we specify some links to allow the development of tools to monitor the scene execution. In that sense, there are links to add and remove some listener (addListener and removeListener, respectively). The general idea of the links is to allow other artifacts to receive information about the scene evolution. For example, it is possible to get information about the enabled states and transitions, the fired transitions and the actions, messages, and events that were responsible to fire each transition.

The last link (updateRolePlayers) is necessary because the interaction mechanism needs to know which are the agents playing each role in the organization. This information is used to handle the cardinalities and to make sure that certain agent is really playing determinate role. The Moise GroupBoard artifact already provides a link to add listeners and then get such information. In the same way, we need to handle the cardinalities of artifacts and verify if certain artifact is of determinate kind. Therefore,
we created a link (getArtifactList) into the WorkspaceArtifact in CArtAgO. This link has the aim to return the list of all artifacts and their kinds in some workspace. Such mechanisms to handle the roles in the organization and the artifacts in the environment were introduced to reify the relations between interaction participant with organizational role and environmental artifact, as presented in the conceptual model (Fig. 1).

5.2 Getting Messages, Actions, and Events

All the messages, actions, and events must be intercepted and sent to the scenes. Fig. 5 shows the interception model. It shows messages, actions, and events being intercepted during their occurrences. The agents do not need to notify the interaction about what they are doing explicitly, since they could try to cheat the interaction mechanism. For example, they could notify the interaction about things that they have never done.

Some related works use a mediator agent to get the necessary information [1], however the mediator agent is an autonomous entity and then it is possibly malicious. Our approach to get messages, actions, and events is similar to the approach presented in [3, 34], where the authors define a layer that behaves like a filter to consider only the correct messages to change the interaction state. In order to do that in JaCaMo platform, in a first moment, we modified the agent architecture. The new agent architecture allows to intercept the messages exchanged between the agents, the events that occurs in the environment, and the actions that the agents perform in the environment. Notice
that the agents interact with the organization in JaCaMo by means of organizational artifacts in the environment, therefore it is not necessary to create a specific mechanism to deal with the actions performed in the organization. In the end, the messages, actions, and events that were intercepted are delivered to the scenes that the agents are attending. Then, they will be processed/evaluated in order to fire the enabled transitions.

6 Results and Discussion

Our main contribution in this paper is the integration of the interaction component into the JaCaMo platform. With this integration we have an MAS platform to program the agents, the environment, the organization, and the interaction, all of them as first class abstractions. We can now specify the interaction in a separated component, avoiding specifying the interaction inside the code of agents or other components.

As another result, we can also specify the agents more independent of the application. Before the integration of our approach into JaCaMo, it was necessary to specify how the agents interact with the other MAS components in their own code. With the interaction integrated into JaCaMo by means of artifacts and assuming the fact that agents already know how to deal with artifacts and organization, the agents do not need any specific mechanism to deal with the interaction. Even in the case of open and heterogeneous MAS, a global behavior can be defined for the overall system by means of the interaction. It is possible because the interaction allows the definition of the desired sequence of steps to achieve the organizational goals. Moreover, while the organizational goals provide information about what the agents need to do, the interaction protocols provide a more detailed description about how to behave to achieve them.

The integration with the JaCaMo platform allowed us to evaluate our interaction proposal and also to provide an example of how to integrate it into an MAS platform composed of agents, environment, and organization. In our experiments, we saw several advantages considering the interaction as a first class abstraction. For example, we can update the interaction, most of times, without changing the code of the other MAS components. We also got some positive results with the relations that we made between the interaction and the other MAS components. For example, the obligations facilitate the agent programming and allow the agents to reason about them, specially whether the agents already can handle with organizational obligations, as in the case of JaCaMo platform. We can change the steps sequence of the protocols and, since the obligations are created in execution time regarding to these steps, we do not need to update the agents code. Moreover, in future works, norms and obligations will allow us to create punishment and reward mechanisms to prevent malicious behavior and reward the agents with good performances. The relation between participant in the interaction and role in the organization allows the agents to search for partners to cooperate because the protocols specify which roles they must interact with. The relation between interaction and environment by means of artifacts permits the specification of how the agents must proceed to interact with the artifacts by means of actions and observable events.

As some drawbacks of the integration with JaCaMo platform, we noticed a decrease in performance and some negative impact related to scalability. In fact, it was an expected impact because we did not focused on performance and scalability issues in a
first moment. The main reason for this negative impact is the interception and manage-
ment of messages, actions, and events that happen in the MAS execution. Since most
of them could be relevant to the scenes, after the interception mechanism catch such
ocurrences we need to send them to the scenes and process them. So far, we built a
centralized solution to process such occurrences in each scene, however it seems not
the best solution for an MAS where there are many messages exchanges, actions, and
events. The improvement of these issues remains as future work.

Another questionable point of our approach is related to the number of different
languages that the developer should learn in order to implement an MAS using JaCaMo
platform. With the integration of the interaction component into JaCaMo platform, there
will have four different languages, each one dedicated to specify one of its components
(agents, organization, environment, and interaction). Indeed, learning four languages
would require more time and investments from the MAS developers. However, all the
four languages are more suitable to implement their own concerns. For example, in
order to specify the environment, it is better to use a specific environmental language
than to specify the environment by means of an agent language. Naturally, when it
is necessary to implement a simple MAS, most of times, the agents themselves are
enough to solve the problems. The organization, environment, and interaction are better
suitable to implement large and complex systems, where the separation of concerns is
underlying.

Finally, our approach is not the only one to deal with interaction and some of the
other components. As we presented in section 2, there are several approaches of inter-
action, however, none of them integrate the interaction with all the other three MAS
components in a unified way. Some of them handle the interaction between agents,
others deal with the interaction and the environment or organization. Furthermore, our
proposal is focused on more complex MAS, composed of agents, environment, and or-
ganization. Our aim is to integrate these components by means of the interaction and
explore the advantages of this kind of MAS. It means that in platforms that are only
composed of agents our approach could not be so suitable.

7 Conclusions and Future Works

In this paper we presented the integration of an approach of interaction considering
agents, environment, and organization into the JaCaMo platform. Although we present
the integration with the JaCaMo platform, our approach can also be integrated with
other MAS platforms. We also highlighted the interaction model and the programming
language. As future works, we intend to evaluate the use of this proposal in the devel-
opment of large systems and also to verify protocols that are created by some agent,
since the agents could create protocols at run-time and execute it. Other interesting sub-
jects to explore are how the agents could reason about a protocol in order to optimize
its execution, and a proposal of a mechanism to specify and handle exceptions. Finally,
mechanisms of punishment and reward should be studied for the purpose of evaluating
the performance of the agents when they are participating of some scene.
Acknowledgments. The authors are grateful for the support given by CNPq, grants 140261/2013-3 and 306301/2012-1. We would also like to thank the reviewers for the useful comments and questions, which helped us to improve this paper.

References


396


## Author Index

**A**  
Ahlbrecht, Tobias 72  
Albayrak, Sahin 145  
Alechina, Natasha 320  
Alexander, Rob 269  
Ancona, Davide 212  

**B**  
Baldoni, Matteo 338  
Baroglio, Cristina 338  
Bordini, Rafael 304  
Brandão, Anarosa A. F. 128  
Briola, Daniela 212  
Brown, Matthew 91  

**C**  
Capuzzimati, Federico 338  
Casare, Sara 128  
Caval, Costin 1  
Clark, John 269  

**D**  
Dastani, Mehdi 178  
Delle Fave, Francesco 91  
Dignum, Frank 357  
Dignum, Virginia 111  
Dix, Jürgen 72  
Dunin-Keplicz, Barbara 285  
Dybalova, Daniela 320  

**E**  
El Fallah Seghrouchni, Amal 1, 212  

**F**  
Ferber, Jacques 357  
Freitas, Artur 304  

**G**  
Gleizes, Marie-Pierre 40  
Graja, Zeineb 40  

**H**  
Hadj Kacem, Ahmed 40  
Hessler, Axel 145  
Huang, Zhan 269  
Huber, Marc 196  
Hubner, Jomi Fred 161, 375  

**J**  
Jensen, Andreas Schmidt 111
<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jiang, Albert</td>
<td>91</td>
</tr>
<tr>
<td>Jones, Randolph</td>
<td>196</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>Kmiec, Slawomir</td>
<td>232</td>
</tr>
<tr>
<td>Kraus, Philipp</td>
<td>72</td>
</tr>
<tr>
<td>Köster, Michael</td>
<td>72</td>
</tr>
<tr>
<td>Küster, Tobias</td>
<td>145</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Lee, Jeehang</td>
<td>320</td>
</tr>
<tr>
<td>Lespérance, Yves</td>
<td>232</td>
</tr>
<tr>
<td>Logan, Brian</td>
<td>320</td>
</tr>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Mascardi, Viviana</td>
<td>212</td>
</tr>
<tr>
<td>Maurel, Christine</td>
<td>40</td>
</tr>
<tr>
<td>Meneguzzi, Felipe</td>
<td>304</td>
</tr>
<tr>
<td>Mermet, Bruno</td>
<td>250</td>
</tr>
<tr>
<td>Migeon, Frederic</td>
<td>40</td>
</tr>
<tr>
<td>Müller, Jörg P.</td>
<td>72</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Nunes, Ingrid</td>
<td>55</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>Padget, Julian</td>
<td>320</td>
</tr>
<tr>
<td>Panisson, Alison</td>
<td>304</td>
</tr>
<tr>
<td>Parunak, H. Van Dyke</td>
<td>196</td>
</tr>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>Quist, Michael</td>
<td>196</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Regayeg, Amira</td>
<td>40</td>
</tr>
<tr>
<td>Rocha Costa, Antônio Carlos</td>
<td>20</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Schmidt, Daniela</td>
<td>304</td>
</tr>
<tr>
<td>Shieh, Eric</td>
<td>91</td>
</tr>
<tr>
<td>Sichman, Jaime S.</td>
<td>128</td>
</tr>
<tr>
<td>Simon, Gaëlle</td>
<td>250</td>
</tr>
<tr>
<td>Szalas, Andrzej</td>
<td>285</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Taillibert, Patrick</td>
<td>1, 212</td>
</tr>
<tr>
<td>Tambe, Milind</td>
<td>91</td>
</tr>
<tr>
<td>Testerink, Bas</td>
<td>178</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>Uez, Daniela Maria</td>
<td>161</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Vanhée, Loïs</td>
<td>357</td>
</tr>
<tr>
<td>Verbrugge, Rineke</td>
<td>285</td>
</tr>
<tr>
<td>Vieira, Renata</td>
<td>304</td>
</tr>
</tbody>
</table>
Villadsen, Jørgen 111
Z
Zaientz, Jack 196
Zatelli, Maicon Rafael 375
Zhang, Chao 91