DTU Library

A Fixpoint-Based Calculus for Graph-Shaped Computational Fields

Lluch Lafuente, Alberto; Loreti, Michele; Montanari, Ugo

Citation (APA):
Lluch Lafuente, A., Loreti, M., & Montanari, U. (2015). A Fixpoint-Based Calculus for Graph-Shaped Computational Fields. In T. Holvoet, & M. Viroli (Eds.), *Coordination Models and Languages: Proceedings of the 17th IFIP WG 6.1 International Conference (COORDINATION 2015)* (pp. 101-116). Springer. https://doi.org/10.1007/978-3-319-19282-6_7

A Fixpoint-based Calculus for Graph-shaped Computational Fields [⋆]

Alberto Lluch Lafuente[1], Michele Loreti[2], and Ugo Montanari[3]

[1] DTU Compute, Technical University of Denmark, Denmark
[2] University of Florence, Italy
[3] Computer Science Department, University of Pisa, Italy

Abstract. Coordination is essential for dynamic distributed systems exhibiting autonomous behaviors. Spatially distributed, locally interacting, propagating computational fields are particularly appealing for allowing components to join and leave with little or no overhead. In our approach, the space topology is represented by a graph-shaped field, namely a network with attributes on both nodes and arcs, where arcs represent interaction capabilities between nodes. We propose a calculus where computation is strictly synchronous and corresponds to sequential computations of fixpoints in the graph-shaped field. Under some conditions, those fixpoints can be computed by synchronised iterations, where in each iteration the attributes of a node is updated based on the attributes of the neighbours in the previous iteration. Basic constructs are reminiscent of the semiring μ-calculus, a semiring-valued generalisation of the modal μ-calculus, which provides a flexible mechanism to specify the neighbourhood range (according to path formulae) and the way attributes should be combined (through semiring operators). Additional control-flow constructs allow one to conveniently structure the fixpoint computations. We illustrate our approach with a case study based on a disaster recovery scenario, implemented in a prototype simulator that we use to evaluate the performance of a disaster recovery strategy.

1 Introduction

Coordination is essential in all the activities where an ensemble of agents interacts within a distributed system. Particularly interesting is the situation where the ensemble is dynamic, with agents entering and exiting, and when the ensemble must adapt to new situations and must have in general an autonomic behavior. Several models of coordination have been proposed and developed in the last years. Following the classification of [10] we mention (i) direct coordination, (ii) connector-based coordination, (iii) shared data space, (iv) shared deductive knowledge base, and (v) spatially distributed, locally interacting, propagating computational fields. Among them, computational fields are particularly appealing for their ability of allowing new interactions with little or no need of

---

communication protocols for initialization. Computational fields are analogous to fields in physics: classical fields are scalars, vectors or tensors, which are functions defined by partial differential equations with initial and/or boundary conditions. Analogously, computational fields consist of suitable space dependent data structures where interaction is possible only between neighbors.

Computational fields have been proposed as models for several coordination applications, like amorphous computing, routing in mobile ad hoc and sensor networks, situated multi agent ecologies, like swarms, and finally for robotics applications, like coordination of teams of modular robots. Physical fields, though, have the advantage of a regular structure of space, e.g. the one defined by Euclidean geometry, while computational fields are sometimes based on some (logical) network of connections. The topology of such a network may have little to do with Euclidean distance, in the sense that a node can be directly connected with nodes which are far away, e.g. for achieving a logarithmic number of hops in distributed hash tables. However, for several robotics applications, and also for swarms and ad hoc networking, one can reasonably assume that an agent can interact only with peers located within a limited radius. Thus locality of interaction and propagation of effects become reasonable assumptions.

Contributions. The main contribution of the paper is the *Soft Mu-calculus for Computational fields* (SMuC) calculus, where computation is strictly synchronous and corresponds to sequential computations of fixpoints in a graph-shaped field that represents the space topology. Our graph-based fields are essentially networks with attributes on both nodes and arcs, where arcs represent interaction capabilities between nodes. In particular, fixpoints can be computed by synchronised iterations under reasonable conditions, where in each iteration the attribute of a node is updated based on the attributes of the neighbours in the previous iteration. Basic constructs are reminiscent of the semiring μ-calculus [8], a semiring-valued generalisation of the modal μ-calculus, which provides a flexible mechanism to specify the neighbourhood range (according to path formulae) and the way attributes should be combined (through semiring operators). Additional control-flow constructs allow one to conveniently structure the fixpoint computations.

An additional contribution is a novel disaster recovery coordination strategy that we use here as a case study. The goal of the coordination strategy is to direct several rescuers present in the network to help a number of victims, where each victim may need more than one rescuer. While an optimal solution is not required, each victim should be reached by its closest rescuers, so to minimise intervention time. Our proposed approach may need several iterations of a sequence of three propagations: the first to determine the distance of each rescuer from its closest victim, the second to associate to every victim v the list of rescuers having v as their closest victim, so to select the best k of them, if k helpers are needed for v; finally, the third propagation is required for notifying each selected rescuer to reach its specific victim.

We have also developed a prototype tool for our language, equipped with a graphical interface that provides useful visual feedback to users of the language.

We use indeed those visual features to illustrate the application of our approach to the aforementioned case study.

Last, we discuss several aspects related to possible distributed implementation of our calculus. In particular, we sketch a simple endpoint projection that would automatically generate distributed code to be deployed on the agents of the the network and we discuss the possibility of using spanning tree based techniques to efficiently implement some of the global synchronisations involved in such endpoint projection.

Structure of the paper. The rest of the paper is structured as follows. Sect. 2 presents the SMuC calculus. Sect. 3 presents the SMuC specification of our disaster recovery case study, which is illustrated with figures obtained with our prototypical tool. Sect. 4 discusses several performance and synchronisation issues related to distributed implementations of the calculus. Sect. 5 discusses related works. Sect. 6 concludes the paper, describes our current work and identifies opportunities for future research.

2   SMuC: A Soft μ-calculus for Computations fields

Our computational fields are essentially networks of inter-connected agents, where both agents and their connections have attributes. One key point in our proposal is that the domain of attributes and their operations have the algebraic structure of a class of semirings usually known as *absorptive semirings* or *constraint semirings*. Such class of semirings has been shown to be very flexible, expressive and convenient for a wide range of problems, in particular for optimisation and solving in problems with soft constraints and multiple criteria [4].

Definition 1 (semiring). *An* absorptive semiring *is a set A with two operators* $+$, $\times$ *and two constants* $\bot$, $\top$ *such that*

- $+ : 2^A \to A$ *is an associative, commutative, idempotent operator to "choose" among values;*
- $\times : A \times A \to A$ *is an associative, commutative operator to "combine" values;*
- $\times$ *distributes over* $+$*;*
- $\top + a = a$*,* $\bot + a = \bot$*,* $\bot \times a = a$*,* $\top \times a = \top$ *for all* $a \in A$*;*
- $\leq$*, which is defined as* $a \leq b$ *iff* $a + b = b$*, provides a lattice of preferences with top* $\top$ *and bottom* $\bot$*;*

We will use the term *semiring* to refer to absorptive semirings. Typical examples are the Boolean semiring $\langle \{true, false\}, \vee, \wedge, false, true \rangle$, the tropical semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, min, +, +\infty, 0 \rangle$, and the fuzzy semiring $\langle [0,1], max, min, 0, 1 \rangle$. A useful property of semirings is that Cartesian products and power constructions yield semirings, which allows one for instance to lift techniques for single criteria to multiple criteria.

We are now ready to provide our notion of field, which is essentially a graph equipped with semiring-valued node and edge labels. The idea is that nodes play

the role of agents, and (directed) edges play the role of (directional) connections. The node labels will be used as attributes of the agents, while the node labels correspond to functions associated to the connections, e.g. representing how attribute values are transformed when traversing a connection.

Definition 2 (field). *A field is a tuple* $\langle N, E, A, L = L_N \uplus L_E, I = I_N \uplus I_E \rangle$ *formed by*

- *a set N of nodes;*
- *a relation $E \subseteq N \times N$ of edges;*
- *a set L of node labels L_N and edge labels L_E;*
- *a semiring A;*
- *an interpretation function $I_N : L_N \to N \to A$ associating a function from nodes to values to every node label in L_N;*
- *an interpretation function $I_E : L_E \to E \to A \to A$ associating a function from edges to functions from values to values to every edge label in P;*

where node, edge, and label sets are drawn from a corresponding universe, i.e. $N \subseteq \mathcal{N}$, $E \subseteq \mathcal{E}$, $L_N \subseteq \mathcal{L}$, $L_E \subseteq \mathcal{L}'$.

As usual, we may refer to the components of a field F using subscripted symbols (i.e. N_F, E_F, ...). We will denote the set of all fields by $\mathcal{F}$.

It is worth to remark that while standard notions of computational fields tend to be restricted to nodes (labels) and their mapping to values, our notion of field includes the topology of the network and the mapping of edge (labels) to functions. As a matter of fact, the topology plays a fundamental role in our field computations as it defines how agents are connected and how their attributes are combined when communicated. On the other hand, in our approach the role of node and edge labels is different. In fact, some node labels are computed as the result of a fixpoint approximation which corresponds to a propagation procedure. They thus represent the genuine computational fields. Edge labels, instead, are assigned directly in terms of the data of the problem (e.g. distances) or in terms of the results of previous propagations. They thus represent more properly equation coefficients and boundary conditions as one can have in *partial differential equations* in physical fields.

SMuC (*Soft μ-calculus for Computations fields*) is meant to specify global computations on fields. One key aspect of our calculus are atomic computations denoted with expressions reminiscent of the semiring modal μ-calculus proposed in [8], a semiring-valued generalisation of the modal μ-calculus, used to reason about quantitative properties of graph-based structures (e.g. transition systems, network topologies, etc.). In SMuC similar expressions will be used to specify the functions being calculated by global computations, to be recorded by updating the interpretation functions of the nodes. Such atomic computations can be embedded in any language. To ease the presentation we present a global calculus where atomic computations are embedded in a simple imperative language reminiscent of WHILE [12].

(μSTEP)	$$\frac{[\![\Psi]\!]_{\emptyset}^{I_F} = f \quad I'_F = I_F[^f/_i]}{\langle i \leftarrow \Psi, F\rangle \rightarrow \langle \mathsf{skip}, F[^{I'_F}/_{I_F}]\rangle}$$
(SEQ1)	$$\frac{\langle P, F\rangle \rightarrow \langle P', F'\rangle}{\langle P;Q, F\rangle \rightarrow \langle P';Q, F'\rangle}$$
(SEQ2)	$$\frac{\langle P, F\rangle \rightarrow \langle P', F'\rangle}{\langle \mathsf{skip};P, F\rangle \rightarrow \langle P', F'\rangle}$$
(IFT)	$$\frac{[\![\Psi]\!]_{\emptyset}^{F} = \lambda n.a \text{ for some } a \in A_F}{\langle \mathsf{if} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{then} \ P \ \mathsf{else} \ Q, F\rangle \rightarrow \langle P, F\rangle}$$
(IFF)	$$\frac{[\![\Psi]\!]_{\emptyset}^{F} \neq \lambda n.a \text{ for some } a \in A_F}{\langle \mathsf{if} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{then} \ P \ \mathsf{else} \ Q, F\rangle \rightarrow \langle Q, F\rangle}$$
(UNTILF)	$$\frac{[\![\Psi]\!]_{\emptyset}^{F} \neq \lambda n.a \text{ for some } a \in A_F}{\langle \mathsf{until} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{do} \ P, F\rangle \rightarrow \langle (P \ ; \ \mathsf{until} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{do} \ P), F\rangle}$$
(UNTILT)	$$\frac{[\![\Psi]\!]_{\emptyset}^{F} = \lambda n.a \text{ for some } a \in A_F}{\langle \mathsf{until} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{do} \ P, F\rangle \rightarrow \langle \mathsf{skip} , F\rangle}$$

Table 1. Rules of the operational semantics

Definition 3 (SMuC syntax). *The syntax of* SMuC *is given by the following grammar*

$$P, Q ::= \mathsf{skip} \mid i \leftarrow \Psi \mid P \ ; \ P' \mid \mathsf{if} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{then} \ P \ \mathsf{else} \ Q$$
$$\mid \quad \mathsf{until} \cdot \mathsf{agree} \cdot \mathsf{on} \ \Psi \ \mathsf{do} \ P$$

where $i \in \mathcal{L}$, Ψ is a SMuC *formula (cf. Def 4).*

We remark that the main difference with respect to the while language are the $\mathsf{agree} \cdot \mathsf{on}$ variants of the traditional control flow constructs. We explicitly use a different syntax in order to remark the characteristic semantics of those constructs, where the global control flow depends on the existence of agreements among all agents in the field.

The semantics of the calculus is straightforward, along the lines of WHILE [12] with fields (and their interpretation functions) playing the role of memory stores. In addition we have that the right-hand side of assignments are SMuC formulas that we will introduce next.

Given a semiring A, a function $\mathcal{N} \rightarrow A$ is called a *node valuation*. Given a set $\mathcal{Z}$ of variables, a set $\mathcal{M}$ of function symbols, an environment is a function $\rho : \mathcal{Z} \rightarrow \mathcal{N} \rightarrow A$.

Definition 4 (syntax of SMuC formulas). *The syntax of* SMuC *formulas is as follows:*

$$\Psi ::= i \mid z \mid f(\Psi, \ldots, \Psi) \mid [a]\Psi \mid \langle a \rangle \Psi \mid [[a]]\Psi \mid \langle\langle a \rangle\rangle.\Psi \mid \mu z.\Psi \mid \nu z.\Psi$$

with $i \in \mathcal{L}$, $a \in \mathcal{L}'$, $f \in \mathcal{M}$ *and* $z \in \mathcal{Z}$.

We remark that the set of functions symbols may include, among others, the semiring operator symbols $+$ and $\times$ and possibly some additional ones, for which an interpretation on the semiring of interest can be given.

Definition 5 (semantics of SMuC formulas). *Let F be a field. The semantics of* SMuC *formulas is given by the interpretation function* $\llbracket \cdot \rrbracket_\rho^F : \Psi \to N_F \to A_F$ *defined by*

$$\llbracket i \rrbracket_\rho^F = I_F(i)$$
$$\llbracket z \rrbracket_\rho^F = \rho(z)$$
$$\llbracket f(\Psi_1, \ldots, \Psi_n) \rrbracket_\rho^F = \llbracket f \rrbracket_{A_F}(\llbracket \Psi_1 \rrbracket_\rho^F, \ldots, \llbracket \Psi_n \rrbracket_\rho^F) \rrbracket_\rho^F$$
$$\llbracket [a]\Psi \rrbracket_\rho^F = \lambda n. \ \prod_{\{n' \mid (n,n') \in E_F\}} .I_F(a)(n,n')(\llbracket \Psi \rrbracket_\rho^F(n'))$$
$$\llbracket \langle a \rangle.\Psi \rrbracket_\rho^F = \lambda n. \ \sum_{\{n' \mid (n,n') \in E_F\}} .I_F(a)(n,n')(\llbracket \Psi \rrbracket_\rho^F(n'))$$
$$\llbracket [[a]]\Psi \rrbracket_\rho^F = \lambda n. \ \prod_{\{n' \mid (n',n) \in E_F\}} .I_F(a)(n',n)(\llbracket \Psi \rrbracket_\rho^F(n'))$$
$$\llbracket \langle\langle a \rangle\rangle\Psi \rrbracket_\rho^F = \lambda n. \ \sum_{\{n' \mid (n',n) \in E_F\}} .I_F(a)(n',n)(\llbracket \Psi \rrbracket_\rho^F(n'))$$
$$\llbracket \mu z.\Psi \rrbracket_\rho^F = lfp \ \lambda d.\llbracket \Psi \rrbracket_{\rho[d/z]}^F$$
$$\llbracket \nu z.\Psi \rrbracket_\rho^F = gfp \ \lambda d.\llbracket \Psi \rrbracket_{\rho[d/z]}^F$$

where lfp and gfp stand for the least and greatest fixpoint, respectively.

As usual, the semantics is well defined if so are all fixpoints. A sufficient conditions for fixpoints to be well-defined is for functions $\lambda d.\llbracket \Psi \rrbracket_{\rho[d/z]}^F$ to be continuous and monotone (cf. Tarski's theorem). This implies, for instance, that if a negation operation is part of the function symbols f used in a formula, as reasonable with some but not all semiring instances, then we should ensure that all fixpoint variables have positive polarity. Another desirable property is for functions to be computable by iteration. This requires the fixpoint to be equal to Ψ^n for $n \in \mathbb{N}$, where $\Psi^{i+1} = \llbracket \Psi \rrbracket_{\rho[\Psi^i/z]}^F$ and $\Psi^0 = \llbracket \Psi \rrbracket_{\rho[\alpha/z]}^F$, with $\alpha = \bot$ if we are computing a least fixpoint and $\alpha = \top$ if we are computing a greatest fixpoint. The formulae we use in our case study satisfy the above mentioned properties.

The semantics of our calculus is a transition system whose states are pairs of calculus terms and fields and whose transitions $\to \subseteq (P \times \mathcal{F})^2$ are defined by the rules of Table 1. Most rules are standard. Rule IFT and IFT are similar to the usual rules for conditional branching. However, the condition is not a Boolean value but the existence of an agreement on the same value a to be assigned on each agent n in the field F. If such agreement exists, the then branch is taken, otherwise the else branch is followed. Similarly for the until $\cdot$ agree $\cdot$ on operator (cf. rules UntilT and UNTILF). States of the form (skip, I) represent termination. Initial states must have all node and edge labels interpreted.

```
finish ← false;
until · agree · on finish do
      /* 1st Stage: */
      /* Establishing the distance to victims */
      D ← μZ.min₁(source, ⟨dist⟩Z);

      /* 2nd Stage: */
      /* Computing the rescuers paths */
      rescuers ← μZ.init ∪ ⟨⟨grad⟩⟩Z;

      /* 3rd Stage: Engaging rescuers */
      finish ← false;
      /* engaging the rescuers */
      engaged ← μZ.choose ∪ ⟨cograd⟩Z;
      /* updating victims and available rescuers */
      victim' ← victim;
      victim ← victim ∧ ¬saved;
      rescuer ← rescuer ∧ engaged ≠ ∅;
      /* determining termination */
      finish ← (victim' == victim);

/* 4th Stage: Checking success */
if · agree · on ¬victim
      /* ended with success */
else
      /* ended with failure */
```

```
/* Semiring types of labels */

         source, D : N → T ×₁ N≤N
     init, rescuers : N → 2^(T×N*)
  choose, engaged : N → 2^(N*)
             dist : E → T ×₁ N≤N → T ×₁ N≤N
             grad : E → 2^(T×N*) → 2^(T×N*)
           cograd : E → 2^(N*) → 2^(N*)
```

$$\text{source}, \mathsf{D} : N \to T \times_1 N_{\leq N}$$
$$\text{init}, \text{rescuers} : N \to 2^{T \times N^*}$$
$$\text{choose}, \text{engaged} : N \to 2^{N^*}$$
$$\text{dist} : E \to T \times_1 N_{\leq N} \to T \times_1 N_{\leq N}$$
$$\text{grad} : E \to 2^{T \times N^*} \to 2^{T \times N^*}$$
$$\text{cograd} : E \to 2^{N^*} \to 2^{N^*}$$

Fig. 1. Robot Rescue SMuC Program

3 SMuC at Work: Rescuing Victims

The left side of Fig. 2 depicts a simple instance of the considered scenario. There, victims are rendered as black circles while landmarks and rescuers are depicted via grey and black rectangles respectively. The length of an edge in the graph is proportional to the distance between the two connected nodes. The main goal is to assign rescuers to victims, where each victim may need more than one rescuer and we want to minimise the distance that rescuers need to cover to reach their assigned victims. We assume that all relevant information of the victim rescue scenario is suitably represented in field F. More details on this will follow, but for now it suffices to assume that nodes represent rescuers, victims or landmarks and edges represent some sort of direct proximity (e.g. based on visibility w.r.t. to some sensor).

It is worth to remark that in practice it is convenient to define A as a Cartesian product of semirings, e.g. for differently-valued node and edge labels. This is indeed the case of our case study. However, in order to avoid explicitly dealing with these situations (e.g. by resorting to projection functions, etc.) which would introduce a cumbersome notation, we assume that the corresponding semiring is implicit (e.g. by type/semiring inference) and that the interpretation of functions and labels are suitably specialised. For this purpose we decorate the specification in Fig. 1 with the types of all labels.

We now describe the coordination strategy specified in the algorithm of Fig. 1. The algorithm consists of a loop that is repeated until an iteration does not
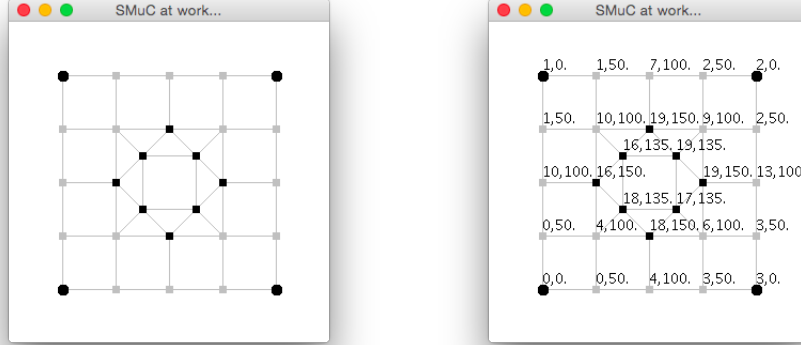
Fig. 2. Execution of Robot Rescue SMuC Program (part 1)

produce any additional matching of rescuers to victims. The body of the loop consist of different stages, each characterised by a fixpoint computation.

1st Stage: Establishing the distance to victims. In the first stage of the algorithm the robots try to establish their closest victim. Such information is saved in to D, which is valued over the total ordering semiring obtained by the lexicographical construction applied to the tropical semiring T and to the semiring $N_{\leq_N}$ given by some total ordering on the nodes N. We denote such construction by $N \to T \times_1 N_{\leq_N}$. In order to compute D some information is needed on nodes and arrows of the field, in particular the decorations are source and dist whose interpretation is defined as follows:

- $I(\mathsf{source})(n) = $ if $n \in victim$ then $(0, n)$ else $(+\infty, n)$, i.e. victims point to themselves with no cost, while the rest of the nodes point to themselves with infinite cost;
- $I(\mathsf{dist})(n, n') = \lambda(v, m).(distance(n, n') + v, n')$ where $distance(n, n')$ is the weight of (n, n'). Intuitively, dist provides a function to add the cost associated to the transition. The second component of the value encodes the direction to go for the shortest path, while the total ordering on nodes is used for solving ties.

The desired information is then computed as $\mathsf{D} \leftarrow \mu Z.\mathsf{min}_1(\mathsf{source}, \langle \mathsf{dist} \rangle Z)$. This fix point calculation is very similar to the standard ones used to calculate reachability or shortest paths. Here min_1 is the additive operation of semiring $N \to T \times_1 N_{\leq_N}$, specifically for a set $B \subseteq (\mathbb{R} \cup \{+\infty\}) \times N$ the function min_1 is defined as $\mathsf{min}_1(B) = (a, n) \in B$ such that $\forall (a', n') \in B : a \leq a'$ and if $a = a'$ then $n \leq n'$.

At the end of this stage, D associates each element with the distance to its closest victim. In the right side of Fig. 2 each node of our example is labeled with the computed distance. We do not include the second component of D (i.e. the identity of the closest neighbour) to provide a readable figure. In any case, the closest victim is easy to infer from the depicted graph: the closest victim of the rescuer in the top-left corner of the inner box formed by the rescuers is the victim at the top-left corner of the figure, and respectively for the top-right, bottom-left and bottom-right corners.

2nd Stage: Computing the rescuers paths to the victims. In this second stage of the algorithm, the robots try to compute, for every victim v, which are the paths from every rescuer u to v — but only for those u for which v is the closest victim — and the corresponding costs, as established by D in the previous stage. Here we use the semiring $2^{T \times N^*}$ with union as additive operator, i.e. $\langle 2^{T \times N^*}, \cup, \cap, T \times N^*, \emptyset \rangle$. We use here decorations init and grad whose interpretation is defined as

- $I(\mathsf{init})n = $ if $n \in$ rescuer then $\{(\mathsf{D}(n), \epsilon)\}$ else $\emptyset$;
- $I(\mathsf{grad})(n, n') = \lambda C.$ if $\mathsf{D}(n) = (u, n')$ then $n; C$ else $\emptyset$, where operation ; is defined as $n; C = \{(cost, n; path) \mid (cost, path) \in C\}$.

The idea of label *rescuers* is to compute, for every node n, the set of rescuers whose path to their closest victim passes through n (typically a landmark). However, the name of a rescuer is meaningless outside its neighbourhood, thus a path leading to it is constructed instead. In addition, each rescuer is decorated with its distance to its closest victim. Function init associates to a rescuer its name and its distance, the empty set to all the other nodes. Function grad checks if an arc (n, n') is on the optimal path out of n. In the positive case, the rescuers in n are considered as rescuers also for n', but with an updated path; in the negative case they are discarded.

In left side of Fig. 3 the result of this stage is presented. There, the edges that are part of path from one rescuer to a victim are now marked. We can notice that some victims can be reached by more than one rescuer.

3rd Stage: Engaging the rescuers. The idea of the third stage of the algorithm is that each victim n, which needs k rescuers, will choose the k closest rescuers, if there are enough, among those that have selected n as target victim. For this computation we use the decorations choose and cograd.

- $I(\mathsf{choose})(n) = $ if $n \in victim$ and $\mathsf{saved}(n)$ then $opt(\mathsf{rescuer}(n), howMany(n))$ else $\emptyset$, where:
  - $\mathsf{saved}(n) = |\mathsf{rescuers}(n)| \leq howMany(n)$ and $howMany(n), n \in victim$ returns the number of rescuers n needs;
  - $opt(C, k) = \{path \mid (cost, path) \in C$ and
    $$|\{(cost', path') \mid (cost', path') < (cost, path)\}| < k\}$$
    where $(cost, path) < (cost', path')$ if $cost < cost'$ or $cost = cost'$ and $path < path'$, and paths are totally ordered lexicographically;
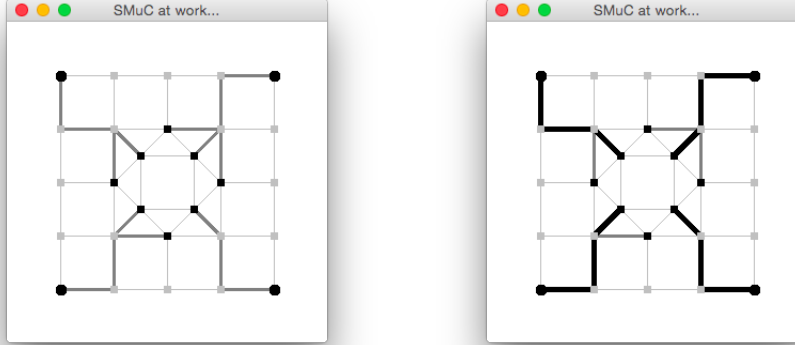
Fig. 3. Execution of Robot Rescue SMUC Program (part 2)

- $I(\mathsf{cograd}(n, n') = \lambda C.\{path \mid n; path \in C\}$.

Intuitively, choose allows a victim n that has enough rescuers to choose and to record the paths leading to them. The annotation cograd associates to each edge (n, n') a function to select in a set C of paths those of the form $n; path$.

The computation in this step is $\mathsf{engaged} \leftarrow \mu Z.\mathsf{choose} \cup \langle \mathsf{cograd} \rangle Z$, which computes the desired information: in each node n we will have the set of rescuer-to-victim paths that pass through n and that have been chosen by a victim.

The result of this stage is presented in the right side of Fig. 3. Each rescuer has a route, that is presented in the figure with black edges, that can be followed to reach the assigned victim. Again, for simplicity we just depict some relevant information to provide an appealing and intuitive representation.

Notice that this phase, and the algorithm, may fail even if there are enough rescuers to save some additional victims. For instance if there are two victims, each requiring two rescuers, and two rescuers, the algorithm fails if each rescuer is closer to a different victim.

These three stages are repeated until there is agreement on whether to finish. The termination criteria is that an iteration did not update the set of victims. In that case the loop terminates and the algorithm proceeds to the last stage.

4th Stage: Checking succes. The algorithm terminates with success when $victim' = \emptyset$ and with failure when $victim'$ is not empty. In Fig. 4 we present the result of the computation of program of Fig. 1 on a randomly generated graph composed by 1000 landmarks, 5 victims and 10 rescuers. We can notice that, each victim can be reached by more than one rescuer and that the closer one is selected.
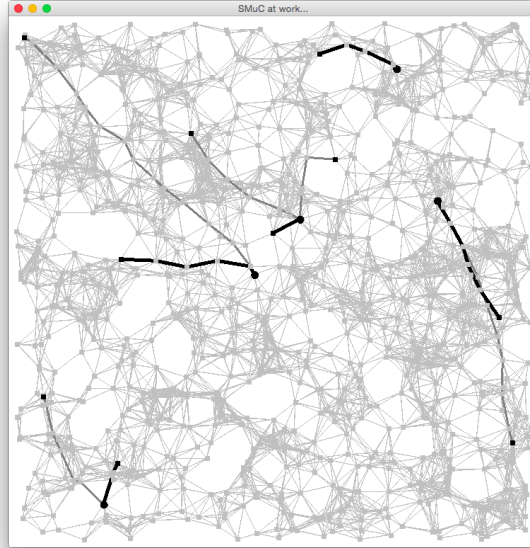
Fig. 4. Execution of Robot Rescue SMuC Program on a random graph

4   On Distributing SMuC Computations

We discuss in this section some aspects of a distributed implementation of SMuC computations. Needless to say, an obvious implementation would be based on a *centralised* algorithm. In particular, the nodes could initially send all their information to a centralised coordinator that would construct the field, compute the SMuC computations, and distribute the results back to the nodes. This solution is easy to realise and could be based on our prototype which indeed performs a centralised, global computation, as a sequential program acting on the field. However, such a solution has several obvious drawbacks: first, it creates a bottleneck in the coordinator. Second, there are many applications in which the idea of constructing the whole field is not feasible and each agent needs to evolve independently. We discuss here some possible alternatives.

A naïve distributed implementation. We start with a naïve distributed implementation based on an endpoint projection of SMuC computations on local programs on the nodes. Such projection is sketched informally in Fig. 5 where a projection function ·⌈ maps SMuC programs and formulas into local code to be executed on agents. We neglect the formal presentation of the local programming language and rely on the intuition of the reader since the main goal is to make explicit the (high) amount of synchronisation points in such an approach. Those synchronisation points are marked by underlining the corresponding statements.

$$
\begin{aligned}
P\!\upharpoonright_F &= \;||_{n \in N_F}\; n : P\!\upharpoonright_n \\
\mathsf{skip}\!\upharpoonright_n &= \;\mathsf{skip} \\
i \leftarrow \Psi\!\upharpoonright_n &= \;\mathsf{self}.i \leftarrow \Psi\!\upharpoonright_n \\
P \; ; \; P'\!\upharpoonright_n &= \;P\!\upharpoonright_n \; ; \; \underline{\mathsf{sync}} \; ; \; P'\!\upharpoonright_n \\
\mathsf{if} \cdot \mathsf{agree} \cdot \mathsf{on}\; \Psi \;\mathsf{then}\; P \;\mathsf{else}\; Q\!\upharpoonright_n &= \;\underline{\nu\;\mathsf{global}\;z;} \\
&\quad\; \underline{\mathsf{self}.z \leftarrow \Psi\!\upharpoonright_n;} \\
&\quad\; \underline{\mathsf{if} \cdot \mathsf{global} \cdot \mathsf{agree} \cdot \mathsf{on}\; z}\; \mathsf{then}\; P\!\upharpoonright_n \;\mathsf{else}\;\; Q\!\upharpoonright_n \\
\mathsf{until} \cdot \mathsf{agree} \cdot \mathsf{on}\; \Psi \;\mathsf{do}\; P\!\upharpoonright_n &= \;\underline{\nu\;\mathsf{global}\;z;} \\
&\quad\; \underline{\mathsf{self}.z \leftarrow \Psi\!\upharpoonright_n;} \\
&\quad\; \underline{\mathsf{until} \cdot \mathsf{global} \cdot \mathsf{agree} \cdot \mathsf{on}\; z}\; \mathsf{do} \\
&\qquad\; P\!\upharpoonright_n; \\
&\qquad\; \mathsf{self}.z \leftarrow \Psi\!\upharpoonright_n; \\
i\!\upharpoonright_n &= \;\mathsf{self}.i \\
z\!\upharpoonright_n &= \;\mathsf{self}.z \\
f(\Psi_1,\ldots,\Psi_m)\!\upharpoonright_n &= \;f(\Psi_1\!\upharpoonright_n,\ldots,\Psi_m\!\upharpoonright_n) \\
\Psi + \Psi'\!\upharpoonright_n &= \;\Psi\!\upharpoonright_n + \Psi'\!\upharpoonright_n \\
\Psi \times \Psi\!\upharpoonright_n &= \;\Psi\!\upharpoonright_n \times \Psi\!\upharpoonright_n \\
[a]\Psi\!\upharpoonright_n &= \;\prod_{\{n'|(n,n')\in E_F\}} .I_F(a)(n,n')(n'.\Psi\!\upharpoonright'_n) \\
\langle a\rangle\Psi\!\upharpoonright_n &= \;\sum_{\{n'|(n,n')\in E_F\}} .I_F(a)(n,n')(n'.\Psi\!\upharpoonright'_n) \\
[[a]]\Psi\!\upharpoonright_n &= \;\prod_{\{n'|(n,n')\in E_F\}} .I_F(a)(n',n)(n'.\Psi\!\upharpoonright'_n) \\
\langle\langle a\rangle\rangle.\Psi\!\upharpoonright_n &= \;\sum_{\{n'|(n',nn)\in E_F\}} .I_F(a)(n,n')(n'.\Psi\!\upharpoonright'_n) \\
\iota z.\Psi\!\upharpoonright_n &= \;\underline{\nu\;\mathsf{global}\;z;} \\
&\quad\; \underline{\mathsf{self}.z \leftarrow \alpha(\iota);} \\
&\quad\; \underline{\mathsf{until} \cdot \mathsf{global} \cdot \mathsf{fixpoint}(z)}\;\; \mathsf{do} \\
&\qquad\; \mathsf{self}.z \leftarrow \Psi\!\upharpoonright_n; \\
&\qquad\; \underline{\mathsf{sync}};
\end{aligned}
$$

where $\iota \in \{\mu,\nu\}$ and $\alpha(\mu) = \top$, $\alpha(\mu) = \bot$.

Fig. 5. Naïve end point projection of SMuC computations

Note that every occurrence of a sequential composition, every control flow construct and every fixpoint iteration involves a global synchronisation like a global barrier (e.g. sync) or a global commit (e.g. $\mathsf{global} \cdot \mathsf{agree} \cdot \mathsf{on}$). Indeed, each agent has to locally check if a step of the computation has been completely computed or if other iterations are needed to compute the correct value. This holds, in particular, when fixpoints formulas are considered. In what follows we discuss opportunities to optimise and relax those synchronisation points.

Spanning-tree based synchronisations. We describe now a technique that, by relying on a specific structure, can be used to perform SMuC computations in an improved way. The corner stone of the proposed algorithm is a *tree-based* infrastructure that spans the complete field. In this infrastructure each node/agent, that is identified by a unique identifier, is responsible for the coordination of the computations occurring in its sub-tree. In the rest of this section we assume that this *spanning tree* is computed in a *set-up* phase executed when the system is

deployed. We also assume that each agent only interacts with its neighbours and that it knows their identities.

It should be clear from the endpoint projection in Figure 5 that when a SMuC program consists of a sequence of assignments $v_0 \leftarrow \Psi_0 \ldots v_k \leftarrow \Psi_k$, a global barrier needs to be used to ensure that all processes proceed synchronously to guarantee that the computation of v_{i+1} is started only when the computation of v_i has been globally completed. We now discuss a technique that uses a tree infrastructure to implement such global barrier in an efficient way. The optimisation regards also the possible local iterations due to the necessity to compute fixpoints.

As sketched in Fig. 5 each agent sends to (and receives from) its neighbours local values computed in Ψ_i (cf. the use of $n' \ldots$ in the projection of modal operators). Since each Ψ_i may contain several fixpoints, these values have to be computed iteratively.

An alternative to the projection in Fig. 5 would be as follows. Each value within an iteration could be sent together with the index k of the computational step and with the actual iteration. Following this approach each agent would be able to compute the values at some iteration when all the values corresponding to the previous iteration have been collected from its neighbours. When a *local fixpoint* is reached (i.e. its value did not change with respect to the previous iteration) an agent would reach a *local stability point*. An agent becomes *stable* when it is *locally stable* and all its children in the spanning tree are stable (for the leaves of the spanning tree, *local stability* and *stability* coincides). Note that an agent can be stable at a given iteration and unstable in the next one. This happens when an update of local values is propagated in the field.

The node devoted to check the global stability in the field is the root of the spanning tree. We can observe that each update in the field is propagated to the root in a number of steps that equates the height of the spanning tree. For this reason, when the root of the spanning tree is *stable* for a number of iterations that is greater than the height of the spanning tree, a global stability can be assured. After that the root informs all the nodes in the spanning tree that computation of step i is terminated and the index of the current step is updated accordingly. Each node starts the computation of step $i + 1$ just after the commit for the step i has been received.

5 Related Works

In recent years, spatial computing has emerged as a promising approach to model and control systems consisting of a large number of cooperating agents that are distributed over a physical or logical space [3]. This computational model starts from the assumption that, when the density of involved computational agents increases, the underlying network topology is strongly related to the geometry of the space through which computational agents are distributed. Goals are generally defined in terms of the system's spatial structure. A main advantage of these approaches is that their computations can be seen both as working on

a single node, and as computations on the distributed data structures emerging in the network (the so-called "computational fields").

One of the main examples in this area is Proto [1,2]. This language aims at providing links between local and global computations and permits the specification of the individual behaviour of a node, typically in a sensor-like network, via specific space-time operators to situate computation in the physical world. In [15] a minimal core calculus has been introduced to capture the key ingredients of languages that make use of computational fields. In [14] a typed variant of the core calculus of [15] is presented. The new proposed calculus is also equipped with a type-system ensuring self-stabilisation of any well-typed program.

The calculus proposed in this paper starts from a different perspective with respect to the ones mentioned above. In these calculi, computational fields result from (recursive) functional composition. These functions are typically used to compute a single field, which may consists of a tuple of different values. In our approach, at each step of a SMuC program a different field can be computed and then used in the rest of the computation. This is possible because in SMuC only monotone continuous functions over the appropriate semirings are considered. This guarantees the existence of fixpoints and the possibility to identify a global stability in the field computation. This is not possible in other approaches. Of course, monotonicity and continuity do not guarantee computability of the fixpoints by iteration. Other methods may be needed. Further investigations are needed to compare the expressive power of SMuC with respect to the languages and calculi previously proposed in literature.

Different middleware/platforms have been proposed to support coordination of distributed agents via computational fields [9,13,11]. In [9] the framework TOTA (*Tuples On The Air*), is introduced to provide spatial abstractions for a novel approach to distributed systems development and management, and is suitable to tackle the complexity of modern distributed computing scenarios, and promotes self-organisation and self-adaptation. In [13] a similar approach has been extended to obtain a chemical-inspired model. This extends tuple spaces with the ability of evolving tuples mimicking chemical systems and provides the machinery enabling agents coordination via spatial computing patterns of competition and gradient-based interaction. Finally, in [11] a framework for distributed agent coordination via *eco-laws* has been proposed. This kind of laws generalise the chemical-inspired ones [13] in a framework where self-organisation can be injected in pervasive service ecosystems in terms of spatial structures and algorithms for supporting the design of context-aware applications. The proposed calculus considers computational fields at a more higher level of abstraction with respect to the above mentioned frameworks. However, these frameworks could provide the means for developing a distributed implementation of SMuC.

6   Conclusion

We have presented a simple calculus, named SMuC, that can be used to program and coordinate the activities of distributed agents via computational fields. In

SMuC a computation consists of a sequence of fixpoints computed in a graph-shaped field that represents the space topology modelling the underlying network. Our graph-based fields have attributes on both nodes and arcs, where the latter represent interaction capabilities between nodes. Under reasonable conditions, fixpoints can computed via synchronised iterations. At each iteration the attributes of a node are updated based according to the values of neighbours in the previous iteration. SMuC is also equipped with a set of control-flow constructs allow one to conveniently structure the fixpoint computations. We have also developed a prototype tool for our language, equipped with a graphical interface that provides useful visual feedback to users of the language. We use indeed those visual features to illustrate the application of our approach to a robot rescue case study, for which we provide a novel rescue coordination strategy, specified in SMuC.

The general aspects related to possible distributed implementation of our calculus have been also discussed. We have sketched a naïve (overly synchronised) distributed implementation and an improvement based on a spanning tree structure aimed at minimising communication and accelerating the detection of fixpoints. We are currently investigating further distribution techniques. The first one is to perform the updates in the fixpoint iterations sequentially but respecting fairness. The stable result should be the same, but efficiency should be significantly improved if causality of iteration updates is traced, e.g. using a queue as in Dijkstra's shortest path algorithm. The second idea is to update variables looking at one neighbour at a time. Under suitable conditions again the result should be the same, but the amount of asynchrony, and thus efficiency, should increase remarkably. Of course, particular instances of the fixpoint iterations (e.g. when considering associative, commutative, idempotent operations) would allow more drastic improvements by allowing agents to proceed asynchronously, synchronising to ensure a barrier between to sequential programs.

References

1. J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, 2006.
2. J. Beal, O. Michel, and U.P. Schultz. Spatial computing: Distributed systems that take advantage of our geometric world. *ACM Transactions on Autonomous and Adaptive Systems*, 6:11:1–11:3, 2011.
3. Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012.
4. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
5. Carlos Canal and Massimo Villari, editors. *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013,*

Revised Selected Papers, volume 393 of *Communications in Computer and Information Science*. Springer, 2013.

6. eva Kühn and Rosario Pugliese, editors. *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*. Springer, 2014.

7. Pietro Liò, Eiko Yoneki, Jon Crowcroft, and Dinesh C. Verma, editors. *Bio-Inspired Computing and Communication, First Workshop on Bio-Inspired Design of Networks, BIOWIRE 2007, Cambridge, UK, April 2-5, 2007, Revised Selected Papers*, volume 5151 of *Lecture Notes in Computer Science*. Springer, 2008.

8. Alberto Lluch-Lafuente and Ugo Montanari. Quantitative mu-calculus and CTL defined over constraint semirings. *Theor. Comput. Sci.*, 346(1):135–160, 2005.

9. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engingeering and Methodology*, 18:15:1–15:56, 2009.

10. Marco Mamei and Franco Zambonelli. Field-based coordination for pervasive computing applications. In Liò et al. [7], pages 376–386.

11. Sara Montagna, Mirko Viroli, Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, and Franco Zambonelli. Injecting self-organisation into pervasive service ecosystems. *MONET*, 18(3):398–412, 2013.

12. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

13. Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *TAAS*, 6(2):14, 2011.

14. Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In eva Kühn and Pugliese [6], pages 163–178.

15. Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In Canal and Villari [5], pages 114–128.