Downloaded from orbit.dtu.dk on: Apr 18, 2024

DTU Library

Formal Reasoning on Natural Language Descriptions of Processes

S'ànchez-Ferreres, Josep; Burattin, Andrea; Carmona, Josep; Montali, Marco

Published in:
Business Process Management

Link to article, DOI:
10.1007/978-3-030-26619-6_8

Publication date:
2019

Document Version
Early version, also known as pre-print

Link back to DTU Orbit

Citation (APA):
S'ànchez-Ferreres, J., Burattin, A., Carmona, J., & Montali, M. (2019). Formal Reasoning on Natural Language Descriptions of Processes. In *Business Process Management* (pp. 86-101). Springer. https://doi.org/10.1007/978-3-030-26619-6_8

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Formal Reasoning on Natural Language Descriptions of Processes

Josep Sànchez-Ferreres[1], Andrea Burattin[2], Josep Carmona[1], Marco Montali[3], and Lluís Padró[1]

[1] Universitat Politècnica de Catalunya
{jsanchezf,jcarmona,padro}@cs.upc.edu
[2] Technical University of Denmark
andbur@dtu.dk
[3] Free University of Bolzano
montali@inf.unibz.it

Abstract. The existence of unstructured information that describes processes represents a challenge in organizations, mainly because this data cannot be directly referred into process-aware ecosystems due to ambiguities. Still, this information is important, since it encompasses aspects of a process that are left out when formalizing it on a particular modelling notation. This paper picks up this challenge and faces the problem of ambiguities by acknowledging its existence and mitigating it. Specifically, we propose a framework to partially automate the elicitation of a formal representation of a textual process description, via text annotation techniques on top of natural language processing. The result is the `ATDP` language, whose syntax and semantics are described in this paper. `ATDP` allows to explicitly cope with several interpretations of the same textual description of a process model. Moreover, we link the `ATDP` language to a formal reasoning engine and show several use cases. A prototype tool enabling the complete methodology has been implemented, and several examples using the tool are provided.

1 Introduction

Organizing business processes in an efficient and effective manner is the overarching objective of *Business Process Management* (BPM). Classically, BPM has been mainly concerned with the quantitative analysis of key performance dimensions such as time, cost, quality, and flexibility [10] without considering in depth the analysis of textual data that talks about processes.

Hence, textual descriptions of processes in organizations are a vast and rather unexploited resource. Not neglecting the information that is present in natural language texts in a organization brings opportunities to complement or correct process information in conceptual models. In spite of this, only very recently *Natural Language Processing* (NLP)-based analysis has been proposed in the BPM context, as reported in [12, 15, 14, 22].

This paper is a first step towards the challenge of unleashing formal reasoning on top of textual descriptions of processes. By relying on *textual annotations*, we

propose ATDP, a multi-perspective language that can be connected to a reasoner so that a formal analysis is possible. From a raw textual description, annotations can be introduced manually, or selected from those inferred by NLP analysis (e.g., from libraries like [16]), thus alleviating considerably the annotation effort. Remarkably, our perspective differs from the usual trend in conceptual modelling, i.e., ATDP specifications can contain several interpretations, so ambiguity is not forced to be ruled out when modelling, for those cases when the process is under-specified, or when several interpretations are equally valid.

We formalize ATDP, and describe its semantics using linear temporal logic (LTL), with relations defined at two different levels, thanks to the notion of *scopes*. Then we show how to cast reasoning on such a specification as a model checking instance, and provide use cases for BPM, such as model consistency, compliance checking and conformance checking. Notably, such reasoning tasks can be carried out by adopting the standard infinite-trace semantics of LTL, or by considering instead finite traces only, in line with the semantics adopted in declarative process modeling notations like Declare [17]. Finally, a tool to convert ATDP specifications into a model checking instance is reported.

The paper is organized as follows: in the next section we provide the work related to the contributions of this paper. Then Section 3 contains the preliminaries needed for the understanding of the paper content. Section 4 describes a methodology to use ATDP in organizations. In Section 5 we provide intuition, syntax and semantics behind the ATDP language. Then in Section 6 it is shown how reasoning on ATDP specification can be done through model checking and finally Section 7 concludes the paper.

2 Related Work

In order to automatically reason over a natural language process description, it is necessary to construct a formal representation of the actual process. Such generation of a formal process model starting from a natural language description of a process has been investigated from several angles in the literature. We can project these techniques into a spectrum of support possibilities to automation: from fully manual to automatic.

The first available option consists in converting a textual description into a process model by manually modeling the process. This approach, widely discussed (e.g., [10, 9]), has been thoroughly studied also from a psychological point of view, in order to understand which are the challenges involved in such process of process modeling [18, 4]. These techniques, however, do not provide any automatic support and the possibility for automatic reasoning is completely depending on the result of the manual modeling. Therefore, ambiguities in the textual description are subjectively resolved.

On the opposite side of the spectrum, there are approaches that autonomously convert a textual description of a process model into a formal representation [13]. Such representation can be a final process model (e.g., as BPMN) [7] and, in this case, it might be possible to automatically extract in-

formation. The limit of these techniques, however, is that they need to resolve ambiguities in the textual description, resulting in "hard-coded" interpretations.

In the middle of the spectrum, we have approaches that automatically process the natural language text but they generate an intermediate artifact, useful to support the manual modeling by providing intermediate diagnostics [8, 20]. The problem of having a single interpretation for ambiguities is a bit mitigated in this case since a human modeler is still in charge of the actual modeling. However, it is important to note that the system is biasing the modeler towards a single interpretation.

The approach presented in this paper drops the assumption of resolving all ambiguities in natural language texts. Therefore, if the text is clear and no ambiguities are manifested, then the precise process can be modeled. However, if this is not the case, instead of selecting one possible ambiguity resolution, our solution copes with the presence of several interpretations for the same textual description.

3 A Recap on Linear Temporal Logics

In this paper, we use Linear Temporal Logic (LTL) [19] to define the semantics of the `ATDP` language. In particular, we use the standard interpretation of temporal logic formulae over *infinite traces*.

LTL formulae are built from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\bigcirc$ (*next-time*) and the binary temporal operator U (*until*):

$$\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \, U \, \varphi_2 \quad \text{with } a \in \mathcal{P}$$

Intuitively, $\bigcirc\varphi$ says that φ holds at the *next* instant, $\varphi_1 \, U \, \varphi_2$ says that at some future instant φ_2 will hold and *until* that point φ_1 always holds. Common abbreviations used in LTL include the ones listed below:

- Standard boolean abbreviations, such as $\top$, $\bot$, $\vee$, $\rightarrow$.
- $\Diamond\varphi = \top \, U \, \varphi$ says that φ will *eventually* hold at some future instant.
- $\Box\varphi = \neg\Diamond\neg\varphi$ says that from the current instant φ will *always* hold.
- $\varphi_1 \, W \, \varphi_2 = (\varphi_1 \, U \, \varphi_2 \vee \Box\varphi_1)$ is interpreted as a *weak until*, and means that either φ_1 holds until φ_2 or forever.

Recall that the same syntax can also be used to construct formulae of LTL interpreted over *finite traces* [6]. Later on in the paper we show how our approach can also accommodate this interpretation. Recall however that the intended meaning of an LTL formula may radically change when moving from infinite to finite traces [5].

4 A Framework for Semantic Reasoning of Natural Language Descriptions of Processes

We briefly describe our envisioned framework for process modelling and management based on natural language. Figure 1 overviews the framework. Given a

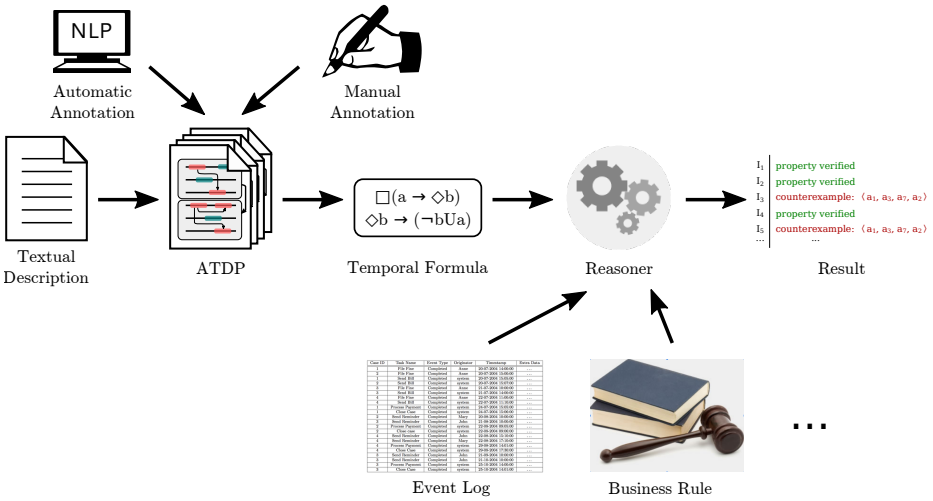

Fig. 1. Annotation framework overview

textual description of a process, automatic or manual annotation (or a combination of both) is used to obtain an Annotated Textual Description of a Process (`ATDP`), which contains all the interpretations of the original text. This specification can then be automatically transformed into temporal formula that encompasses the semantics of the process. The temporal formula can then be queried with the help of a reasoner (e.g., a model checker). Typical use cases may require the encoding of additional inputs, e.g., traces of an event log, compliance rules, among others. The result of the reasoner is the satisfaction or rebuttal (with the corresponding counterexample) of the query. Notice that query results may not hold in all possible interpretations of the text.

5 Processes as Annotated Textual Descriptions

We now propose `ATDP`, a language for annotated textual descriptions of processes starting with a gentle introduction relying on a real-world example. Specifically, we use the textual description of the examination process of a Hospital extracted from [21]. Figure 2 shows the full text, while Figure 3 contains a a fragment of the visualization for an `ATDP` specification of the description.

One of the key features of the `ATDP` approach is the ability to capture *ambiguity*. In our example, we can see this at the topmost level: the text is associated to three different interpretations I_1, I_2 and I_3, providing three different process-oriented semantic views on the text. Each interpretation is a completely unambiguous specification of the process, which fixes a specific way for understanding ambiguous/unclear parts. Such parts could be understood differently in another interpretation. A specification in `ATDP` then consists of the union of

all the valid interpretations of the process, which may partially overlap but also contradict each other.

Each interpretation consists of a hierarchy of *scopes*, providing a recursive mechanism to isolate parts of text that correspond to "phases" in the process. Each scope is thus a conceptual block inside the process, which is in turn decomposed as a set of lower-level scopes. Each scope dictates how its inner scopes are linked via control-flow relations expressing the allowed orderings of execution of such inner scopes. In our example, I_1 contains two scopes. A sequential relation indicates that the second scope is always executed when the first is completed, thus reconstructing the classical flow relation of conventional process modeling notation. All in all, the scope hierarchy resembles that of a process tree, following the variant used in [1].

Inside leaf scopes, *text fragments* are highlighted. There are different types of fragments, distinguished by color in our visual front-end. Some fragments (shown in red) describe the atomic units of behavior in the text, that is, activities and events, while others (shown in blue) provide additional perspectives beyond control flow. For example, `outpatient physician` is labelled as a *role* at the beginning of the text, while `informs` is labelled as an *activity*. Depending on their types, fragments can be linked by means of *fragment relations*. Among such relations, we find:

- Fragment relations that capture background knowledge induced from the text, such as for example the fact that the `outpatient physician` is the role responsible for performing (i.e., is the `Agent` of) the `informs` activity.
- Temporal constraints linking activities so as to declaratively capture the acceptable courses of execution in the resulting process, such as for example the fact that `informs` and `signs an informed consent` are in `succession` (i.e., `informs` is executed if and only if `signs an informed consent` is executed afterwards).

As for temporal relations, we consider a relevant subset of the well-known patterns supported by the Declare declarative process modeling language [17]. In this light, `ATDP` can be seen as a multi-perspective variant of a process tree where the control-flow of leaf scopes is specified using declarative constraints over the activities and events contained therein. Depending on the adopted constraints, this allows the modeler to cope with a variety of texts, ranging from loosely specified to more procedural ones. At one extreme, the modeler can choose to nest scopes in a fine-grained way, so that each leaf scope just contains a single activity fragment; with this approach, a pure process tree is obtained. At the other extreme, the modeler can choose to introduce a single scope containing all activity fragments of the text, and then add temporal constraints relating arbitrary activity fragments from all the text; with this approach, a pure declarative process model is obtained.

5.1 `ATDP` Models

`ATDP` models are defined starting from an input text, which is separated into *typed text fragments*. We now go step by step through the different components

The process starts when the female patient is examined by an outpatient physician, who decides whether she is healthy or needs to undertake an additional examination. In the former case, the physician fills out the examination form and the patient can leave. In the latter case, an examination and follow-up treatment order is placed by the physician, who additionally fills out a request form. Furthermore, the outpatient physician informs the patient about potential risks. If the patient signs an informed consent and agrees to continue with the procedure, a delegate of the physician arranges an appointment of the patient with one of the wards. Before the appointment, the required examination and sampling is prepared by a nurse of the ward based on the information provided by the outpatient section. Then, a ward physician takes the sample requested. He further sends it to the lab indicated in the request form and conducts the follow-up treatment of the patient. After receiving the sample, a physician of the lab validates its state and decides whether the sample can be used for analysis or whether it is contaminated and a new sample is required. After the analysis is performed by a medical technical assistant of the lab, a lab physician validates the results. Finally, a physician from the outpatient department makes the diagnosis and prescribes the therapy for the patient.

Fig. 2. Textual description of a patient examination process.

of our approach, finally combining them into a coherent model. We then move into the semantics of the model, focusing on its temporal/dynamic parts and formalizing them using LTL.

Fragment types. Fragments have no formal semantics associated by themselves. They are used as basic building blocks for defining `ATDP` models. We distinguish fragments through the following types.

Activity. This fragment type is used to represent the atomic units of work within the business process described by the text. Usually, these fragments are associated with verbs. An example activity fragment would be `validates` (from `validates the sample state`). Activity fragments may also be used to annotate other occurrences in the process that are relevant from the point of view of the control flow, but are exogenous to the organization responsible for the execution of the process. For instance, `(the sample) is contaminated` is also an activity fragment in our running example.

Role. The role fragment type is used to represent types of autonomous actors involved in the process, and consequently responsible for the execution of activities contained therein. An example is `outpatient physician`.

Business Object. This type is used to mark all the relevant elements of the process that do not take an active part in it, but that are used/manipulated by activities contained in the process. An example is the (medical) `sample` obtained and analyzed by physicians within the patient examination process.

When the distinction is not relevant, we may refer to fragments as the entities they represent (e.g. *activity* instead of *activity fragment*).

Given a set F of text fragments, we assume that the set is partitioned into three subsets that reflect the types defined above. We also use the following dot

notation to refer to such subsets: *(i)* $F.\mathsf{activities}$ for activities; *(ii)* $F.\mathsf{roles}$ for roles; *(iii)* $F.\mathsf{objects}$ for business objects.

Fragment relations. Text fragments can be related to each other by means of different non-temporal relations, used to express multi-perspective properties of the process emerging from the text. We consider the following relations over a set F of fragments.

Agent. An *agent relation* over F is a partial function

$$agent_F : F.\mathsf{activities} \to F.\mathsf{roles}$$

indicating the role responsible for the execution of an activity. For instance, in our running example we have $agent(\texttt{informs}) = \texttt{physician}$, witnessing that informing someone is under the responsibility of a physician.

Patient. A *patient relation* over F is a partial function

$$patient_F : F.\mathsf{activities} \to F.\mathsf{roles} \cup F.\mathsf{objects}$$

indicating the role or business object constituting the main recipient of an activity. For instance, in our running example we have $patient(\texttt{prepare}) = \texttt{sample}$, witnessing that the `prepare` activity operates over a `sample`.

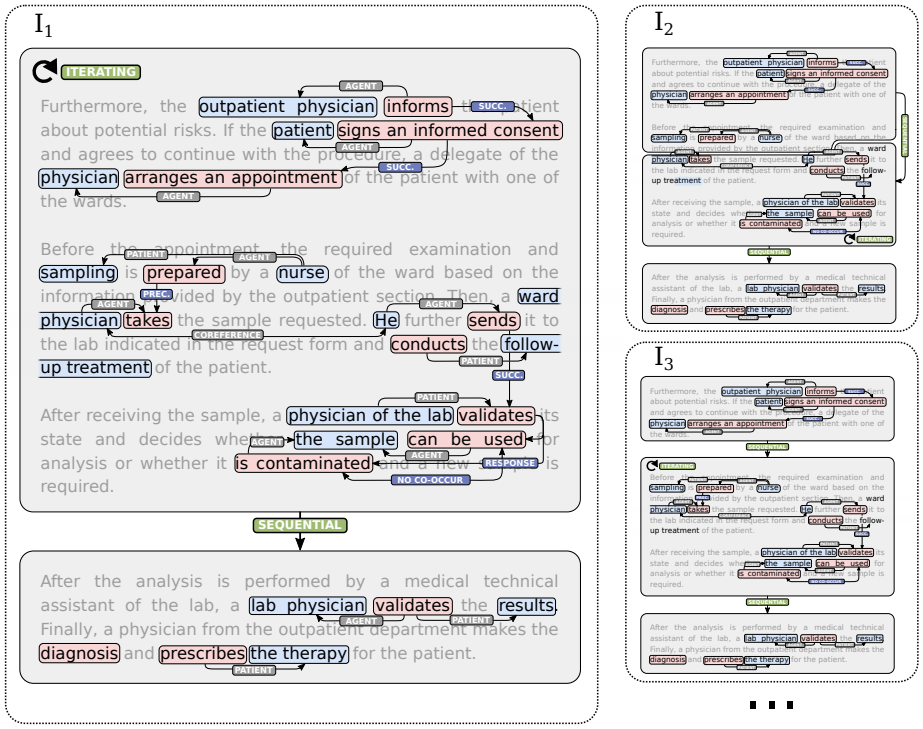

Fig. 3. Example annotation of a textual process description with multiple ambiguous interpretations. Some relations are omitted for brevity.

Coreference. A *coreference relation* over F is a (symmetric) relation

$$coref_F \subseteq F.\mathsf{roles} \times F.\mathsf{roles} \cup F.\mathsf{objects} \times F.\mathsf{objects}$$

that connects pairs of roles and pairs of business objects when they represent different ways to refer to the same entity. It consequently induces a coreference graph where each connected component denotes a distinct process entity. In our running example, all text fragments pointing to the `patient` role corefer to the same entity, whereas there are three different physicians involved in the text: the `outpatient physician`, the `ward physician` and the `physician of the lab`. These form disconnected coreference subgraphs.

Text scopes. To map the text into a process structure, we suitably adjust the notion of process tree used in [1]. In our approach, the blocks of the process tree are actually *text scopes*, where each scope is either a *leaf scope*, or a branching scope containing a one or an ordered pair[4] of (leaf or branching) sub-scopes.

Each activity is associated to one and only one leaf scope, whereas each leaf scope contains one or more activities, so as to non-ambiguously link activities to their corresponding process phases.

Branching scopes, instead, are associated to a corresponding control-flow operator, which dictates how the sub-scopes are composed when executing the process. At execution time, each scope is enacted possibly multiple times, each time taking a certain amount of time (marked by a punctual scope start, and a later completion). We consider in particular the following scope relation types:

Sequential **($\rightarrow$)** A sequential branching scope s with children $\langle s_1, s_2 \rangle$ indicates that each execution of s amounts to the sequential execution of its sub-scopes, in the order they appear in the tuple. Specifically: *(i)* when s is started then s_1 starts; *(ii)* whenever s_1 completes, s_2 starts; *(iii)* the completion of s_2 induces the completion of s.

Conflicting **($\times$)** A conflicting branching scope s with children $\langle s_1, s_2 \rangle$ indicates that each execution of s amounts to the execution of one and only one of its children, thus capturing a choice. Specifically: *(i)* when s is started, then one among s_1 and s_2 starts; *(ii)* the completion of the selected sub-scope induces the completion of s.

Inclusive **($\vee$)** An inclusive branching scope s with children $\langle s_1, s_2 \rangle$ indicates that each execution of s amounts to the execution of at least one of s_1 and s_2, but possibly both.

Interleaving **($\wedge$)** An interleaving branching scope s with children $\langle s_1, s_2 \rangle$ indicates that each execution of s amounts to the interleaved, parallel execution of its sub-scopes, without ordering constraints among them. Specifically: *(i)* when s is started, then s_1 and s_2 start; *(ii)* the latest, consequent completion of s_1 and s_2 induces the completion of s.

Iterating **($\circlearrowleft$)** An iterating branching scope s with child s_1 indicates that each execution of s amounts to the iterative execution of s_1, with one or more

[4] We keep a pair for simplicity of presentation, but all definitions carry over to n-ary tuples of sub-blocks.

iterations. Specifically: *(i)* when s is started, then s_1 starts; *(ii)* upon the consequent completion of s_1, then there is a non-deterministic choice on whether s completes, or s_1 is started again.

All in all, a *scope tree* T_F over the set F of fragments is a binary tree whose leaf nodes S_l are called *leaf scopes* and whose intermediate/root nodes S_b are called *branch nodes*, and which comes with two functions:

- a total *scope assignment* function $\mathit{parent} : F.\mathsf{activities} \to S_l$ mapping each activity in F to a corresponding leaf scope, such that each leaf scope in S_l has at least one activity associated to it;
- a total *branching type* function $\mathit{btype} : S_b \to \{\rightarrow, \times, \vee, \wedge, \circlearrowleft\}$ mapping each branching scope in S_b to its control-flow operator.

Temporal constraints among activities. Activities belonging to the same leaf scope can be linked to each other by means of temporal relations, inspired by the Declare notation [17]. These can be used to declaratively specify constraints on the execution of different activities within the same leaf scope. Due to the interaction between scopes and such constraints, we follow here the approach in [11], where, differently from [17], constraints are in fact scoped.[5]

We consider in particular the following constraints:

Scoped Precedence Given activities $a_1, \ldots, a_n, b$, $\mathsf{Precedence}(\{a_1, \ldots, a_n\}, b)$ indicates that b can be executed only if, within the same instance of its parent scope, at least one among $a_1, \ldots, a_n$ have been executed *before*.

Scoped Response Given activities $a, b_1, \ldots, b_n$, $\mathsf{Response}(a, \{b_1, \ldots, b_n\})$ indicates that whenever a is executed within an instance of its parent scope, then at least one among $b_1, \ldots, b_n$ has to be executed *afterwards*, within the same scope instance.

Scoped Non-Co-Occurrence Given activities a, b, $\mathsf{NonCoOccurrence}(a, b)$ indicates that whenever a is executed within an instance of its parent scope, then b *cannot* be executed within the same scope instance (and vice-versa).

Scoped Alternate Response Given activities $a, b_1, \ldots, b_n$, $\mathsf{AlternateResponse}(a, \{b_1, \ldots, b_n\})$ indicates that whenever a is executed within an instance of its parent scope, then a *cannot be executed again* until, within the same scope, at least one among $b_1, \ldots, b_n$ is *eventually* executed.

Terminating Given activity a, $\mathsf{Terminating}(a)$ indicates that the execution of a within an instance of its parent scope terminates that instance.

Mandatory Given activity a, $\mathsf{Mandatory}(a)$ indicates that the execution of a must occur at least once for each execution of its scope.

Interpretations and models. We are now ready to combine the components defined before into an integrated notion of text interpretation. An *ATDP interpretation* I_X over text X is a tuple $\langle F, \mathit{agent}_F, \mathit{patient}_F, \mathit{coref}_F, T_F, \mathcal{C}_F, \rangle$, where: *(i)* F is a set of *text fragments* over X; *(ii)* agent_F is an *agent function* over F; *(iii)* $\mathit{patient}_F$ is a *patient function* over F; *(iv)* coref_F is a *coreference relation* over F; *(v)* T_F is a *scope tree* over the activities in F; *(vi)* $\mathcal{C}_F$ is a set of *temporal*

[5] It is interesting to notice that Declare itself was defined by relying on the patterns originally introduced in [11].

constraints over the activities in F, such that if two activities are related by a constraint in, then they have to belong to the same leaf scope according to T_F.

An *ATDP model* M_X over text X is then simply a finite set of ATDP interpretations over X.

5.2 ATDP Semantics

We now describe the execution semantics of ATDP interpretations, in particular formalizing the three key notions of scopes, scope types (depending on their corresponding control-flow operators), and temporal constraints over activities. This is done by using LTL, consequently declaratively characterizing those execution traces that conform to what is prescribed by an ATDP interpretation. We consider execution traces as finite sequences of atomic activity executions over interleaving semantics.

Scope Semantics.

To define the notion of scope execution, for each scope s, we introduce a pair of artificial activities st_s and en_s which do not belong to $F.\mathsf{activities}$. The execution of s starts with the execution of st_s , and ends with the execution of en_s. The next three axioms define the semantics of scopes:

A1. An activity a inside a scope s can only be executed between st_s and en_s:

$$\neg a \, W \, st_s \wedge \Box(en_s \rightarrow \neg a \, W \, st_s)$$

A2. A scope s can only be started and ended inside of its parent s':

$$\neg(st_s \vee en_s) \, W \, st_{s'} \wedge \Box(en_{s'} \rightarrow \neg(st_s \vee en_s) \, W \, st_{s'})$$

A3. Executions of the same scope cannot overlap in time. That is, for each execution of a scope s's start there is a unique corresponding end:

$$\begin{aligned} \Diamond en_s \rightarrow (\neg en_s \, U \, st_s) \;\wedge\; \Box(st_s \rightarrow \Diamond en_s) \wedge \\ \Box(st_s \rightarrow \mathsf{O}(\Diamond st_s \rightarrow (\neg st_s \, U \, en_s))) \wedge \\ \Box(en_s \rightarrow \mathsf{O}(\Diamond en_s \rightarrow (\neg en_s \, U \, st_s))) \end{aligned}$$

Temporal Constraint Semantics. In this section, we define the semantics of temporal constraints between activities. Note that, in all definitions we will use

the subindex s to refer to the scope of the constraint.

$$\mathsf{Precedence}_s(\{a_1,..,a_K\},b) := \bigvee_{i=1}^{N} \Box(st_s \rightarrow (\neg b\, U (a_i \vee en_s)))$$

$$\mathsf{Response}_s(a,\{b_1,..,b_N\}) := \bigvee_{i=1}^{N} \Box(st_s \rightarrow (a \rightarrow (\neg en_s\, U\, b_i))\, U\, en_s)$$

$$\begin{aligned}\mathsf{NonCoOccurrence}_p(a,b) := {} & \Box(st_s \rightarrow (a \rightarrow (\neg b\, U\, en_s))\, U\, en_s) \wedge \\ & \Box(st_s \rightarrow (b \rightarrow (\neg a\, U\, en_s))\, U\, en_s)\end{aligned}$$

$$\begin{aligned}\mathsf{AlternateResponse}_p(a,b) := {} & \mathsf{Response}_p(a,b) \ \wedge \\ & \Box(st_s \rightarrow (a \rightarrow \mathsf{O}(\neg a\, U (b \vee en_s)))\, U\, en_s)\end{aligned}$$

$$\mathsf{Terminating}_p(a) := \Box(a \rightarrow \mathsf{O}\, en_s)$$

$$\mathsf{Mandatory}_p(a) := \Box(st_s \rightarrow (\neg en_s\, U\, a))$$

Scope Relation Semantics. In all our definitions, let $\langle s_1, s_2 \rangle$ denote the children of a branching scope s, associated to the control-flow operator being defined. Note that by $\mathsf{Sequence}(a,b)$ we refer to the formula $\mathsf{Precedence}(\{a\},b) \wedge \mathsf{Response}(a,\{b\})$.

Sequential $(\rightarrow)$: $\mathsf{Sequence}_s(en_{s_1}, st_{s_2}) \wedge \mathsf{Mandatory}_s(st_{s_1}) \wedge \mathsf{Mandatory}_s(st_{s_2})$

Conflicting $(\times)$: $\mathsf{Mandatory_s}(st_{\mathsf{s_1}}) \oplus \mathsf{Mandatory_s}(st_{\mathsf{s_2}})$

Inclusive $(\vee)$: $\mathsf{Mandatory_s}(st_{\mathsf{s_1}}) \vee \mathsf{Mandatory_s}(st_{\mathsf{s_2}})$

Interleaving $(\wedge)$: $\mathsf{Mandatory_s}(st_{\mathsf{s_1}}) \wedge \mathsf{Mandatory_s}(st_{\mathsf{s_2}})$

Iterating $(\circlearrowleft)$: This relation is defined by negation, with any non-iterating scope s, child of s', fulfilling the property:

$$(st_{s'} \rightarrow (\neg en_{s'}\, U\, st_s \ \wedge \ (st_s \rightarrow \mathsf{O}(\neg st_s\, U\, en_{s'}))\, U\, en_{s'}))$$

Additionally, iterating scopes may be affected by the presence of terminating activities, as defined by the following property: A terminating activity a_t inside an iterating scope s, child of s', stops the iteration. That is, its execution cannot be repeated anymore inside its parent:

A4. $\Box(st_{s'} \rightarrow ((a_t \rightarrow (\neg st_s\, U\, en_{s'}))\, U\, en_{s'}))$

6 Reasoning on ATDP Specifications

A specification in ATDP is the starting point for reasoning over the described process. This section shows how to encode the reasoning as a model checking instance, so that a formal analysis can be applied on the set of interpretations of the model. Furthermore, we present three use cases in the scope of business process management: *checking model consistency*, *compliance checking* and *conformance checking*.

6.1 Casting Reasoning as Model Checking

Reasoning on ATDP specifications can be encoded as an instance of model checking, which allows performing arbitrary queries on the model. The overall system can be defined by the following formula

$$(A \wedge \mathcal{C}_F \wedge \mathcal{C}_{T_F}) \implies Q \tag{1}$$

where A is the conjunction of all LTL formulas defined by the axioms, $\mathcal{C}_F$ is the conjunction of the activity temporal constraints, $\mathcal{C}_{T_F}$ is the conjunction of all LTL formulas defined by the semantics of the process tree, and Q is an arbitrary query expressed in LTL (cf. Section 5.2).

In this paper, we present an encoding of the ATDP's semantics into NuSMV, a well-known software for model-checking [3]. First, the notion of process execution is defined using an *activity* variable, with a domain of all the activities in the ATDP. At any given step, the system may choose a single value for this variable, meaning that this activity has been executed. This ensures that simultaneous execution of activities will not happen.

The system definition for an ATDP is then split into two parts: a transition system and an LTL property specification. The transition system is a graph defining the next possible values for the *activity* variable given its current value. In our proposed encoding the transition system is specified as a complete graph, since all the behavioural constraints are specified in A, $\mathcal{C}_F$ and $\mathcal{C}_{T_F}$ as parts of the property specification, as seen in Eq. (1). This property specification is directly encoded as a single LTL formula.

We can adapt NuSMV, which performs model checking on infinite traces, to check properties on finite traces when necessary. In order to do that, we add a special activity value STOP. In the transition system, an edge is added from any possible activity to STOP. Additionally, the constraint $\Diamond$STOP is added to the antecedent of the LTL property specification. This enforces that all traces accepted by the model end in an infinite loop repeating the (only) execution of the STOP activity, which is equivalent to terminating execution.

Non-temporal information can be introduced in the queries without increasing the problem complexity, since the information is statically defined. For example, when the text mentions that several activities are performed by a certain role, this information remains invariant during the whole model-checking phase. Thus, queries concerning roles can be translated directly into queries about the

set of activities performed by that role. A possible encoding of this into a model checker consists of adding additional variables during the system definition.

When dealing with multiple interpretations, the above framework is extended with two types of queries:

Existential: Is the proposition true in any interpretation of the process?
$\exists I \in \texttt{ATDP} : (A_I \wedge \mathcal{C}_{F_I} \wedge \mathcal{C}_{T_{F_I}}) \implies Q$
Complete: Is the proposition true in all interpretations of the process?
$\forall I \in \texttt{ATDP} : (A_I \wedge \mathcal{C}_{F_I} \wedge \mathcal{C}_{T_{F_I}}) \implies Q$

Existential and complete queries can be used to reason in uncertain or incomplete specifications of processes.

An application of complete queries would be finding invariant properties of the process. That is, a property that holds in all possible process interpretations. Existential queries, in turn, fulfill a similar role when the proposition being checked is an undesired property of the process. By proving invariant properties, it is possible to extract information from processes even if these are not completely specified or in case of contradictions. A negative result for this type of query would also contain the non-compliant interpretations of the process, which can help the process owner in gaining some insights about which are the assumptions needed to comply with some business rule.

Tool Support.

The encoding technique described in Section 6.1 has been implemented in a prototype tool, `ATDP2NuSMV`. The tool can be used to convert an ATDP specification into a `NuSMV`instance.

`ATDP2NuSMV` is distributed as a standalone tool, that can be used in any system with a modern Java installation, and without further dependencies. A compiled version as well as the source code can be found in the following repository: https://github.com/setzer22/atdp2nusmv.

In the next subsection, we present use cases that have been tested with `ATDP2NuSMV` and `NuSMV`. The `ATDP` specifications as well as the exact query encodings can be found in the repository. The use case examples are based on a full version of the specification presented in Figure 3.

Use Case 1: Model Consistency. An `ATDP` specification can be checked for consistency using proof by contradiction. Specifically, if we set $Q = \bot$, the reasoner will try to prove that $A \wedge \mathcal{C}_F \wedge \mathcal{C}_{T_F} \rightarrow \bot$, that is, whether a false conclusion can be derived from the axioms and constraints describing our model. Since this implication only holds in the case $\bot \rightarrow \bot$, if the proof succeeds we will have proven that $A \wedge \mathcal{C}_F \wedge \mathcal{C}_{T_F} \equiv \bot$, i.e. that our model is not consistent. On the contrary, if the proof fails we can be sure that our model does not contain any contradiction.

To illustrate this use case, we use interpretations `hosp-1` and `hosp-1-bad`, available in our repository. The first interpretation consists of a complete version of the specification in Figure 3, where $F.\mathsf{activities}$ includes a_1 = `takes (the sample)` and a_2 = `validates (sample state)`, and constraints in $\mathcal{C}_F$ include: $\mathsf{Mandatory}(a_1)$, $\mathsf{Precedence}(\{a_1\}, a_2)$ and $\mathsf{Response}(a_1, \{a_2\})$.

NuSMVfalsifies the query in interpretation `hosp-1` with a counter-example. When the model is consistent, the property is false, and the resulting counter example can be any valid trace in the model.

The second specification, `hosp-1-bad` adds $\mathsf{Precedence}(\{a_2\}, a_1)$ to the set of relations R. This relation contradicts the previously existing $\mathsf{Precedence}(\{a_1\}, a_2)$, thus resulting in an inconsistent model. Consequently, NuSMVcannot find a counter-example for the query in interpretation `hosp-1-bad`. This result can be interpreted as the model being impossible to fulfill by any possible trace, and thus inconsistent.

Use Case 2: Compliance Checking.

Business rules, as those arising from regulations or SLAs, impose further restrictions that any process model may need to satisfy. On this regard, compliance checking methods assess the adherence of a process specification to a particular set of predefined rules.

The presented reasoning framework can be used to perform compliance checking on `ATDP` specifications. An example rule for our running example might be: "An invalid sample can never be used for diagnosis". The relevant activities for this property are annotated in the text: a_3 = `(the sample) can be used`, a_4 = `(the sample) is contaminated`, a_5 = `makes the diagnosis`, and the property can be written in LTL as: $Q = \Box(a_4 \rightarrow (\neg a_5 \, U \, a_3))$.

In the examples from our repository, interpretations `hosp-2-i`, with `i={1,2,3}`, correspond to the three interpretations of the process shown in Figure 3. Particularly, the ambiguity between the three interpretations is the scope of the repetition when the taken sample is contaminated. The three returning points correspond to: `sign an informed consent`, `sampling is prepared` and `take the sample`. NuSMVfinds the property true for all three interpretations, meaning that we can prove the property $\Box(a_4 \rightarrow (\neg a_5 \, U \, a_3))$ without resolving the main ambiguity in the text.

Use Case 3: Conformance Checking. Conformance checking techniques put process specifications next to event data, to detect and visualize deviations between modeled and observed behavior [2]. On its core, conformance checking relies on the ability to find out whereas an observed trace can be reproduced by a process model.

A decisional version of conformance checking can be performed, by encoding traces inside Q as an LTL formulation. Given a trace $t = \langle a_1, a_2, \cdots, a_N \rangle$, we can test conformance against an `ATDP` interpretation with the following query[6]:

$$Q = \neg(a_1 \wedge \mathsf{O}(a_2 \wedge \mathsf{O}(... \wedge \mathsf{O}(a_N \wedge \mathsf{O}\texttt{STOP}))))$$

This query encodes the proposition "Trace t is not possible in this model". This proposition will be false whenever the trace is accepted by the model.

[6] The proposed query does not account for the start and end activities of scopes, which are not present in the original trace. A slightly more complex version can be crafted that accounts for any invisible activity to be present between the visible activities of the trace. We do not show it here for the sake of simplicity.

Other variants of this formulation allow for testing trace patterns: partial traces or projections of a trace to a set of activities. In this case, the counter-example produced will be a complete trace which fits the model and the queried pattern.

As an example of this use-case, we provide the example `ATDP` interpretation `hosp-3` in our repository. We project the set of relevant activities to the set a_6 = `informs (the patient)`, a_7 = `signs (informed consent)` a_8 = `arranges (an appointment)`. Two trace patterns are tested, the first: $t_1 = \langle \cdots a_6, a_7, a_8, \cdots \rangle$ and $t_2 = \langle \cdots a_7, a_6, a_8, \cdots \rangle$. `NuSMV`finds the trace pattern t_1 fitting the model, and produces a full execution trace containing it. On the other hand, t_2 does not fit the model, which is successfully proven by `NuSMV`.

7 Conclusions and Future Work

This paper proposes `ATDP`, a novel multi-perspective language for the representation of processes based on textual annotation. On the control-flow dimension, `ATDP` is a mixture of imperative constructs at general level via scopes, and declarative constructs inside each scope. In a way, the language generalizes process trees, allowing declarative relations instead of atomic activities in the leaf nodes. The paper also shows how to translate `ATDP` specifications into temporal formulas that are amenable for reasoning. Three use cases in the context of BPM are shown, illustrating the potential of the ideas in this paper.

Several avenues for future work are under consideration. First, to explore alternatives or refinements of the encoding in Eq. (1) to make it more suitable in a model-checking context. Second, to validate the proposed language against more examples and use cases, specifically by testing how the `ATDP` primitives accommodate to different document styles. Finally, studying the connection between `ATDP` and other process model notations may serve as a bridge between textual descriptions and their operationalization within an organization.

Acknowledgments This work has been supported by MINECO and FEDER funds under grant TIN2017-86727-C2-1-R and by Innovation Fund Denmark project EcoKnow.org (7050-00034A).

References

1. Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. A genetic algorithm for discovering process trees. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2012.
2. Josep Carmona, Boudewijn F. van Dongen, Andreas Solti, and Matthias Weidlich. *Conformance Checking - Relating Processes and Models.* Springer, 2018.
3. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.
4. Jan Claes, Irene Vanderfeesten, J. Pinggera, H.A. Reijers, B. Weber, and G. Poels. A visual analysis of the process of process modeling. *Information Systems and e-Business Management*, 13(1):147–190, 2015.
5. Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 1027–1033. AAAI Press, 2014.

6. Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 854–860. IJCAI/AAAI, 2013.
7. Luis Delicado, Josep Sanchez-Ferreres, Josep Carmona, and Lluís Padró. NLP4BPM - Natural Language Processing Tools for Business Process Management. In *BPM Demo Track*, 2017.
8. Luis Delicado, Josep Sanchez-Ferreres, Josep Carmona, and Lluis Pardo. The Model Judge - A Tool for Supporting Novices in Learning Process Modeling. In *BPM 2018 Demonstration Track*, 2018.
9. Remco Dijkman, Irene Vanderfeesten, and Hajo A. Reijers. Business process architectures: overview, comparison and framework. *Enterprise Information Systems*, 10(2):129–158, feb 2016.
10. Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.
11. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, pages 411–420. ACM, 1999.
12. Henrik Leopold. *Natural language in business process models*. PhD thesis, Springer, 2013.
13. Bilal Maqbool, Farooque Azam, Muhammad Waseem Anwar, Wasi Haider Butt, Jahan Zeb, Iqra Zafar, Aiman Khan Nazir, and Zuneera Umair. A Comprehensive Investigation of BPMN Models Generation from Textual Requirements - Techniques, Tools and Trends. In *Information Science and Applications 2018 - ICISA 2018, Hong Kong, China, June 25-27th, 2018*, volume 514 of *Lecture Notes in Electrical Engineering*, pages 543–557. Springer, 2019.
14. Jan Mendling, Bart Baesens, Abraham Bernstein, and Michael Fellmann. Challenges of smart business process management: An introduction to the special issue. *Decision Support Systems*, 100:1–5, 2017.
15. Jan Mendling, Henrik Leopold, and Fabian Pittke. 25 challenges of semantic process modeling. *International Journal of Information Systems and Software Engineering for Big Companies*, 1(1):78–94, 2015.
16. Lluís Padró and Evgeny Stanilovsky. Freeling 3.0: Towards wider multilinguality. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*, pages 2473–2479, 2012.
17. M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: Full support for loosely-structured processes. In *Proc. of the Eleventh IEEE Int. Enterprise Distributed Object Computing Conference (EDOC'07)*, pages 287–298. IEEE Computer Society, 2007.
18. J. Pinggera. *The Process of Process Modeling*. PhD thesis, University of Innsbruck, Department of Computer Science, 2014.
19. Amir Pnueli. The temporal logic of programs. pages 46–57. IEEE, 1977.
20. Josep Sànchez-Ferreres, Josep Carmona, and Lluís Padró. Aligning textual and graphical descriptions of processes through ILP techniques. In *Proceedings of the 29th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 413–427, 2017.
21. F Semmelrodt. Modellierung klinischer prozesse und compliance regeln mittels BPMN 2.0 und eCRG. Master's thesis, University of Ulm, 2013.
22. Han van der Aa, Josep Carmona, Henrik Leopold, Jan Mendling, and Lluís Padró. Challenges and opportunities of applying natural language processing in business process management. In *Proceedings of the 27th International Conference on Computational Linguistics (COLING)*, pages 2791–2801, 2018.