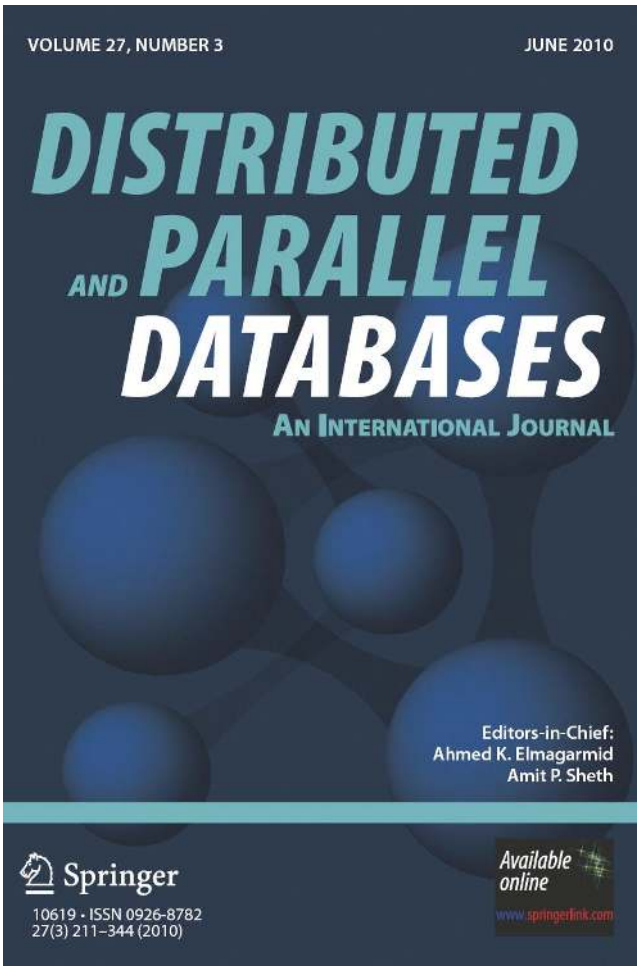
ISSN 0926-8782, Volume 27, Number 3



Distrib Parallel Databases (2010) 2

DOI 10.1007/s10619-010-7060-9

Beyond soundness: on the verification of semantic business process models

Ingo Weber · Jörg Hoffmann · Jan Mendling

Published online: 20 January 2010

Abstract The verification of control-flow soundness is well understood as an important step before deploying business process models. However, the control flow does not capture what the process activities actually do when they are executed. Semantic annotations offer the opportunity to take this into account. Inspired by semantic Web service approaches such as OWL-S and WSMO, we consider process models in which the individual activities are annotated with logical preconditions and effects, specified relative to an ontology that axiomatizes the underlying business domain. Verification then addresses the overall process behavior, arising from the interaction between control-flow and behavior of individual activities. To this end, we combine notions from the workflow community with notions from the AI actions and change literature. We introduce a formal execution semantics for annotated business processes. We point out four verification tasks that arise, concerning precondition/effect conflicts, reachability, and executability. We examine the borderline between classes of processes that can, or cannot, be verified in polynomial time. For precondition/effect conflicts, we show that the borderline is the same as that of the

Communicated by Asuman Dogac.

The major part of this work was conducted while the first and second authors worked for SAP Research, Karlsruhe.

I. Weber
School of Computer Science & Engineering, University of New South Wales, Sydney, NSW 2052, Australia
e-mail: ingo.weber@cse.unsw.edu.au

J. Hoffmann
INRIA, Centre de Recherche Nancy—Grand Est, 54506 Nancy, France
e-mail: joerg.hoffmann@inria.fr

J. Mendling (✉)
Humboldt-Universität zu Berlin, Unter den Linden 6, 10117 Berlin, Germany
e-mail: jan.mendling@wiwi.hu-berlin.de

logic underlying the ontology axioms. For reachability and executability, we identify a class of processes that can be verified in polynomial time by a fixpoint algorithm which we design for that purpose. We show that this class of processes is maximal in the sense that, when generalizing it in any of the most relevant directions, the validation tasks become computationally hard.

Keywords Business process management · Semantic technologies

1 Introduction

The formalism and verification techniques we describe in this paper are rather technical, and the background and intended usage of the proposed technology is not trivial to explain. For these reasons, in what follows we introduce these aspects in some detail. We start with the motivation and background of our work in Sect. 1.1. We give an overview of the approach and its usage in Sect. 1.2. We summarize our formalization and contributions in Sect. 1.3.

1.1 Motivation and background

Nowadays, process-aware information systems utilize process models, which are composed of activities and their associated control flow. This has proven to be an efficient abstraction for rapid implementation of processes [42]. A challenge is to assure correctness. As far as control-flow is concerned, verification of correctness is well understood as an important step before deploying executable business process models, i.e., process models serving as templates for handling individual process instances. The soundness criterion and its derivatives, e.g. [24, 53, 61, 74], are typically used to check whether proper completion is possible or even guaranteed. Tools like Woflan [83] provide the functionality to efficiently verify soundness based on Petri nets theory.

A limitation of soundness verification is that it covers only the control-flow perspective of the process model—in that sense, soundness is a necessary but insufficient condition for correctness. Better support for designing correct models is urgently needed since there are considerable error rates in process models from practice [52]. To assure that a process model indeed behaves as expected, it is necessary to take into account what the individual activities in the process—the activities whose order of execution is governed by the process—actually do when they are executed: What are the prerequisites for the activities to execute successfully? How do they affect the state of the world in case they are executed? Traditional workflow models do not contain any information about this, apart from the naming of the activities.[1] Such activity naming may be sufficient for simple applications in closed domains, where the behavior of the activities is not overly complex and/or known in detail to all persons involved. For more complex applications, however, a more powerful

[1] Indeed, traditional models rather emphasize their black box character to simplify the implementation task.

means of describing the semantics of activities is in order. This is particularly true if the individual activities in the process will be executed by different agents (persons or computers) in a heterogeneous and distributed environment. The first question then is: How should we describe the semantics of activities?

Essentially this same question has been addressed, since several years, in the area of semantic Web services. Approaches such as OWL-S [1, 72] and the Web Service Modeling Ontology (WSMO) [21, 66] are in wide-spread use. At a particular level of abstraction, called "service profile" in OWL-S and "capability level" in WSMO, Web services are perceived as functionalities with a single entry and exit point. This corresponds well to the individual activities in a workflow. The profile/capability of a Web service is described in terms of a precondition—a logical formula capturing the prerequisites of the service—as well as an effect (sometimes also referred to as a "postcondition")—a logical formula capturing how the service affects the state of the world. The formulas are stated relative to the vocabulary of an ontology, which formalizes the underlying domain, i.e., the "world" in which the service executes. The use of ontologies facilitates a precise formulation of the domain structure and its characteristic properties, through ontology axioms. Such formulations are useful for capturing preconditions/effects, and they may reduce ambiguities in the communication among heterogeneous agents.

Following recent work in the area of Semantic Business Process Management (SBPM) [11, 84],[2] we adopt these notions from the semantic Web services area for explicating the behavior of individual process activities. We assume that activities are annotated with logical preconditions and effects. The research question we address is that of verification: Does the control flow interact correctly with the behavior of the individual activities? Precisely, we address the following four verification tasks:

- *Effect conflicts*: Are there activities whose effects are in conflict, but that may be executed in parallel?
- *Precondition conflicts*: Are there activities whose effect and precondition are in conflict, but that may be executed in parallel?
- *Reachability*: Is there an activity that will never be reached by the execution?
- *Executability*: Is there an activity whose precondition may be false at a time when the activity is scheduled for execution?

Note that, in this verification, we go far beyond what is possible based on activity names. We are able to conveniently express and check how particular aspects of the preconditions/effects of some activities affect particular aspects of other activities. For example, activity A may not be reachable because activity B has an effect invalidating the preconditions of activities C and D, if the workflow is such that B must be executed before C and D, and either C or D must be executed before A. In brief: based on "local" annotations, we are able to detect "global" conflicts.

[2]For information about the SBPM area, see e.g. the SUPER IP (EU-funded Integrated Project, http://www.ip-super.org), Semantic Business Process Management Working Group (http://www.sbpm.org), SBPM workshop [31].

1.2 Overview of the approach

Our verification techniques are aimed at helping with the creation of correct business process models. Hence they will be used by a human process modeler. This will happen interactively, i.e., on-line while the human modeler is creating or adapting the process within a BPM modeling environment. The general setting is that the modeler frequently uses the verification as a push-button operation for cross-checking the process and pointing out any bugs, should they exist. To enable such verification, the modeler uses some convenient paradigm for annotating the preconditions/effects of individual tasks (we get back to this below). Note here that the verification can serve as much for debugging the annotations—these may have a value per se by making explicit the intended meaning of the process—as it serves for debugging the control-flow structure. The modeling continues until the process is complete, and proved by the verification to be correct with respect to all four verification tasks.

Figure 1 illustrates the interplay between business process models, annotations, and verification. The figure shows an example of a business process model (top of the figure), which has been partly annotated with terms from an ontology (middle of the figure). Upon calling our verification techniques, a list of problems is returned (bottom of the figure). As shown in the picture, the ontology consists of two taxonomies (a, b, c vs. d, e, f, g, h). To indicate that $x \sqsubseteq y$, i.e., that x is a sub-concept of y, an arrow from x to y is displayed. Further, the ontology states mutual exclusion between c and g, indicated by a dashed double arrow.

In the shown process model, a precondition conflict exists because, although T4 and T5 are parallel, the effect of T4—c—contradicts the precondition of T5—g. Hence, if T4 happens to be executed prior to T5, then T5's precondition will not be satisfied at the point where T5 is scheduled for execution, i.e., when T5's incoming edge carries a token. Similarly, T7 is not executable because T4 is necessarily executed beforehand, which will invalidate T7's precondition.[3]

Why is such a situation—an activity being scheduled for execution at a time when its precondition is not satisfied—problematic? The problem arises (1) if the process is being executed by a standard (non-semantic) engine, e.g. based on BPEL, or (2) if the execution engine is aware of the semantic annotations but cannot observe their truth value at execution time. In both cases, *the activity will be enacted regardless of the unsatisfied precondition*. This may lead to undefined behavior and errors. Scenario (1) is, clearly, all we are going to get in industrial applications, in the foreseeable future. In particular it corresponds to our current use cases at SAP, which we will detail later on (Sect. 7). Scenario (2) is quite likely if the process coordinates activities across a heterogeneous system landscape: while it is conceivable that the meaning of activities is being shared at the modeling level, realizing such communication at IT level is much more challenging and may not even be wanted.

Using our technology requires computational resources for the verification, and human resources for creating the annotations. Regarding computation, most impor-

[3]Note that, due to subsumption, there aren't any other bugs in the process model, with respect to our four verification tasks. Note also that preconditions are *not* understood as events that trigger activity execution. Rather, they are evaluated when workflow execution has reached the corresponding activity.

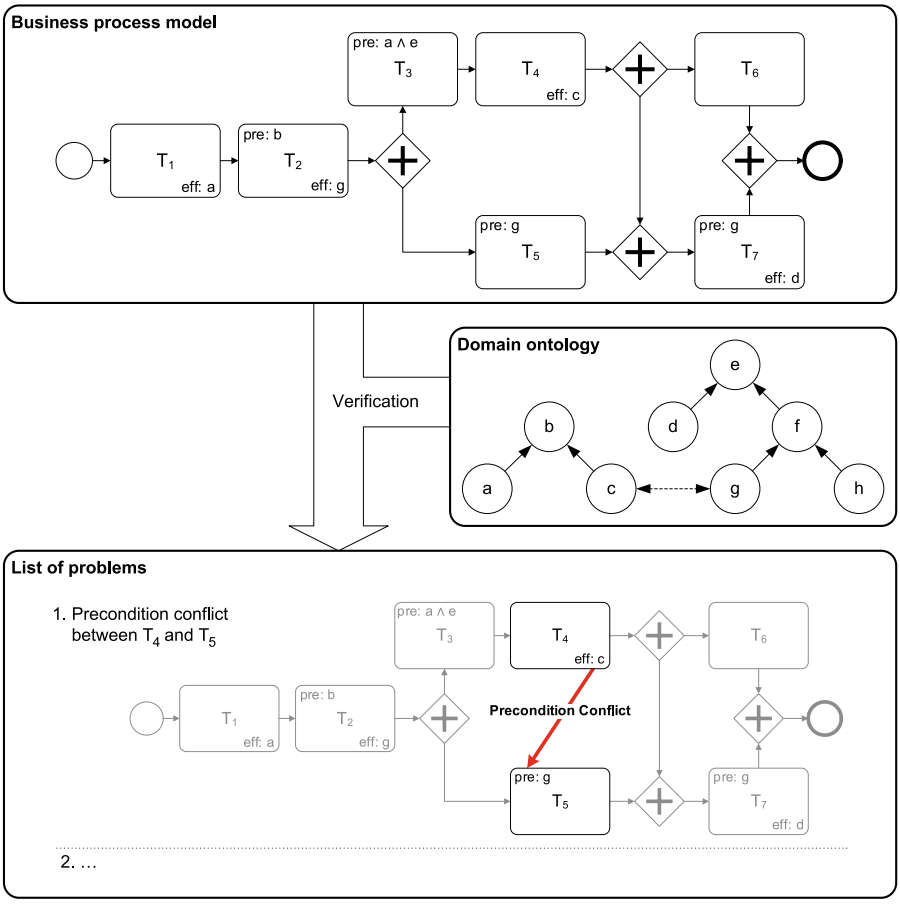


Fig. 1 Illustration of the interplay between business process models, annotations, and verification. The modeler (partly) annotates the process (*top of picture*) with terms from the ontology (*middle*). Verification is called for debugging, yielding a list of problems (*bottom*). In the picture, the ontology consists of two taxonomies (a, b, c vs. d, e, f, g, h) and mutual exclusion between c and g. Note that the annotation is partial, i.e., the modeler does not have to annotate every task node with precondition and effect

tantly there is a tight constraint on response time—human patience. This partly motivates our quest for tractable classes, cf. the next section. As for modeling effort, note first that there is a huge difference between (a) creating the ontology and (b) annotating the preconditions/effects in the process. (a) is done once and for all for an entire domain, while (b) must be done all the time. Fortunately, our experience is that, once (a) has been accomplished, (b) is comparatively easy. As we demonstrate in Sect. 7, annotations can be selected in simple drop-down menus; previous work [12] has shown that value ranking based on relevance parameters such as process context can be quite effective. It should also be noted in this context that the process annotation may be *partial*, as indicated in Fig. 1. We do not require all activities to be annotated. Instead, the modeler may annotate preconditions/effects in a piecemeal fashion as required.

1.3 Summary of formalization and contribution

Our contribution is threefold:

(1) *We devise a formal execution semantics for annotated business processes.* This is technically not challenging, but remarkable because it requires to combine notions from the workflow community with notions from the AI actions and change community. We combine control flow, i.e., token passing, with the AI notion of state changes induced by logical preconditions/effects in the presence of a domain axiomatization. To the best of our knowledge, ours is the first work considering this combination.

(2) *We investigate the borderline between classes of processes that can, or cannot, be verified in polynomial time.* This constitutes the main technical body of the paper. Apart from the theoretical relevance of the investigated borderline, it is important in our application setting, cf. Sect. 1.2. We determine that the complexity of precondition/effect conflict checking is the same as that of the logic underlying the ontology axioms. We identify a maximal tractable class of processes regarding reachability and executability checking.

(3) *We provide initial evidence of the practicality of our approach.* Our techniques are implemented within three SAP Research prototypes. The prototypes show how our technology can be accessed via standard user interactions, and how semantic annotations can be obtained at low cost leveraging existing models at SAP.

In what follows, we explain our formalization and these contributions in more detail.

First, some words are in order regarding our formalism. Our annotated business processes combine syntax for (a) control flow and (b) semantic annotations. Regarding (a), we presume that the reader is basically familiar with process modeling languages like BPEL [58], EPCs [41, 69], BPMN [59], UML Activity Diagrams [10], or YAWL [77]. Our formalization is oriented at BPEL. We cover several major modeling elements, namely parallel splits/joins, xor splits/joins, and structured loops in the form of sub-processes that may be repeated. As for (b), we allow: preconditions/effects annotated at individual activities (governing when an activity can be executed/how it affects the world); conditions annotated at the outgoing edges of xor splits (governing which edge is taken); and conditions annotated at loops (governing whether a loop is repeated or exited). All these are logical formulas, which for the sake of simplicity we restrict to be conjunctions of literals. Our ontology axioms take the form of universally quantified clauses. These can be used to state many typical domain properties, such as subsumption $\forall x : \neg p(x) \vee q(x)$, disjointness $\forall x : \neg p(x) \vee \neg q(x)$, or coverage $\forall x : \neg p(x) \vee q_1(x) \vee \cdots \vee q_n(x)$. The investigation of richer annotations, e.g. using Description Logic [4], is an open topic.

We define an execution semantics for annotated processes, i.e., we define what the state space of a process is. Each state consists of two parts: (a) the positions of the tokens; and (b) the logical state, a truth value assignment to the propositions formed with the ontology vocabulary. For (a), our formalism relies on a straightforward token passing semantics. As for (b), this is trivial *without* ontology axioms: simply modify the state with the effect literals of the activity that was executed. *With* axioms, however, logical state transitions are a long-standing topic for a whole research area, see

e.g. [15, 32, 46, 91]. A useful way to think about the ontology axioms is as "physical laws". As a simple example, the ontology might say that a claim can never be accepted and rejected at the same time. Say an activity rejects a claim. The effect of that activity contradicts "one half" of the axiom, necessitating truth of the other half. Hence, by physical law, the claim is not accepted any longer. With more complex "laws", in particular dependencies between more than two properties of the domain, contradictions between effects and axioms are no longer so easily handled. In a nutshell, the problem is that the contradictions can be solved in several ways, yielding several possible outcome states. It is not clear which of these should be considered physically possible. In our work, we adopt the "possible models approach" (PMA) suggested by Winslett [91], which is in wide-spread use and, in particular, underlies most formalisms relating to the execution of semantic Web services, e.g. [5, 23, 47]. The PMA admits all outcome states that differ from the previous state in a minimal way. Intuitively, "minimality" here ensures that the world does not change without a reason. In the above example, physical law will invalidate acceptance of the claim under consideration; it will not affect, e.g., the status of any other claim.

To briefly position our formalism with respect to the literature, first note that traditional verification techniques, as considered in the workflow community (e.g. [78]) and the model checking community (e.g. [18]), do not deal with ontologies.[4] Without ontology axioms, on the other hand, our formalism can be compiled into Petri nets and other known paradigms. This is obvious for our token-passing semantics and easy to see for preconditions etc; a related approach [57] has previously appeared. However, our results regarding tractability cannot be derived from the literature via such compilation. We get back to this below. There is a body of related work addressing business process models extended beyond control flow, namely [6, 9, 20, 40, 43, 48, 49, 62, 64, 68]. None of them combines control flow token passing with AI actions and change as we do, and none of them addresses similar verification tasks.

Our core technical contribution is the investigation of the borderline between tractable and intractable verification. For effect and precondition conflicts, it turns out that this borderline is the same as that of the logic underlying the ontology axioms. Whether or not two activities are parallel depends only on the control flow, not on the semantic annotations. Control-flow parallelism can be determined in polynomial time. This follows from earlier results in Petri net theory: Kovalyov and Esparza [44] devise an algorithm for free-choice Petri nets that determines concurrency in cubic time. We improve on this result by devising an algorithm that exploits the particular structure of our control-flows, and runs in quadratic time. Once parallelism has been determined, checking for precondition/effect conflicts reduces to satisfiability tests. With our particular formalism for axioms, this means that we inherit the well-known tractability/intractability results for clausal formulas.

For reachability and executability, matters are more complicated, because those tasks are more tightly linked to the overall behavior of the process. We show that neither can be checked in polynomial time (unless $\mathbf{P} = \mathbf{NP}$) if we allow either of:

[4] Many existing verification techniques are heavily based on logic, i.e., on the manipulation of formulas representing certain properties of the system, like the set of states that can be reached within a given number of steps. However, semantic descriptions formalize the behavior of the system, not its properties.

unrestricted axioms; Horn axioms; xor conditions; or loop conditions. We further show that, even in the class of what we call *basic* processes, with binary axioms and without xor/loop conditions, it is **NP**-hard to test reachability and it is **coNP**-hard to test whether or not a particular individual activity is executable. Our positive result is that, for basic processes, we can test in polynomial time whether *all* activities are executable. The proof is constructive, i.e., we devise a polynomial-time verification algorithm. None of these results have been stated, or follow directly from, results in the literature. For the case where there are preconditions and effects, but no ontology axioms, results from Petri net theory can be re-used to identify two tractable classes for reachability checking. This has been recognized by [57]. One of these classes is a proper subset of basic processes; the other class is complementary.

Our algorithms are implemented as a back-end prototype that is used in three front-end tools at SAP Research. One of these addresses a different application (that we had not originally foreseen), namely a brokerage configuration task in service marketplaces [89]. The two other applications use the back-end within a process modeling environment, in an online setting exactly as described above in Sect. 1.2. One of these front-ends is an SAP Research prototyping platform called Maestro, the other is a research extension to the commercial SAP NetWeaver platform. Both applications show how the annotation of preconditions etc. can be done via standard GUI interactions, e.g., selecting the annotation literals in simple drop-down menus. The SAP NetWeaver application, moreover, demonstrates very nicely how we can leverage existing models at SAP. The ontology underlying the tool is taken from a pre-existing model at SAP, describing the status variables of more than 400 business objects which are typically affected by process activities at SAP. Apart from the fact that this ontology comes "for free", it is based on already established terminology, with which process modelers are likely to be familiar.

The paper is organized as follows. Section 2 introduces our formalism for semantic business process models. Section 3 formalizes the four verification tasks we consider, and states a few basic facts about their dependencies. Section 4 presents our results for the verification of effect and precondition conflicts. Section 5 contains our negative results on the computational complexity of reachability and executability checking, while Sect. 6 contains the positive results, i.e., the polynomial-time verification algorithm for basic processes. Section 7 gives details on our prototypes, and discusses the difficulty of creating semantic annotations. Section 8 discusses related work, Sect. 9 concludes the paper. For the sake of readability, the text gives only proof sketches. Full proofs are available in Appendix.

2 Annotated process graphs

We introduce our formalism for semantic business process models. As stated, the formalism combines control-flow (adopted from the workflow community) with preconditions/effects (adopted from the semantic Web service and AI communities). For the sake of readability, we first consider the former in isolation, then enrich it with the latter.

2.1 Process graphs

Our formalization of control-flow in process models covers xor splits, xor joins, parallel splits, parallel joins, and structured loops in the form of sub-graphs that may be repeated. The former four constructs (splits and joins) are a core set of the most consensual routing elements of various process modeling languages like EPCs [41, 69], BPMN [59], UML Activity Diagrams [10], and YAWL [77]. Indeed, these constructs are among those that are most frequently used in BPMN [93]. As for the latter construct (structured loops), these are oriented at the widely adopted BPEL language. BPEL directly enforces such a structure; recent process graph parsing techniques can be applied to untangle loops and move them into sub-processes [82, 92].

Our analysis and algorithms apply to any process model that can be expressed in terms of our constructs. Notationally, we build on [81], which—after straightforward extensions for structured loops—offers a concise and easy to read syntax for expressing the particular constructs we are interested in. (In contrast to the more generic nature of notations like Petri nets, which would complicate the formulation of the particular distinction lines that are relevant for us.) Formally, the syntax of process graphs is defined as follows.

Definition 1 There are two kinds of Process Graphs: *atomic* process graphs, and *complex* process graphs.

1. An *atomic process graph* is a directed graph $P = (N, E)$, where N is the disjoint union of $\{n_0, n_+\}$ (*start node*, *stop node*), N_T (*task nodes*), N_{PS} (*parallel splits*), N_{PJ} (*parallel joins*), N_{XS} (*xor splits*), and N_{XJ} (*xor joins*). For $n \in N$, $in(n)/out(n)$ denotes the set of incoming/outgoing edges of n. We require that: P is acyclic; for every split node n, $|in(n)| = 1$ and $|out(n)| > 1$; for every join node n, $|in(n)| > 1$ and $|out(n)| = 1$; for every $n \in N_T$, $|in(n)| = 1$ and $|out(n)| = 1$; $|in(n_0)| = 0$ and $|out(n_0)| = 1$ and vice versa for n_+; every node $n \in N$ is on a path from the start to the stop node.
2. Say $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are process graphs. Then a *complex process graph* is a triple $\mathcal{P} = (N, E, \lambda)$, where (N, E) is like an atomic process graph except that there is an additional kind of nodes, N_L, called *loop nodes*, where $|in(n)| = 1$ and $|out(n)| = 1$ for every $n \in N_L$. λ is a bijection from N_L into $\{\mathcal{Q}_1, \ldots, \mathcal{Q}_m\}$.

$\mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are *sub-graphs* of $\mathcal{P}$; a sub-graph of any $\mathcal{Q}_i$ is (recursively) also a sub-graph of $\mathcal{P}$. For any process graph $\mathcal{Q}$, we denote by $Sub(\mathcal{Q})$ the set of graphs containing $\mathcal{Q}$ as well as all its sub-graphs. We use superscripts in order to distinguish constructs belonging to particular sub-graphs; e.g., $N^{\mathcal{Q}_2}$ is the node set of sub-graph $\mathcal{Q}_2$. Further, we use $\mathcal{N}$ to denote the set of all nodes appearing in $\mathcal{P}$ itself or in any of its sub-graphs, and we use $\mathcal{E}$ to denote the set of all edges appearing in $\mathcal{P}$ itself or in any of its sub-graphs. For any set of nodes, we use the sub-scripts T, PS, PJ, XS, XJ, and L to distinguish the different kinds of nodes, as above.

We require that the direct sub-graph relation is a tree; precisely, the graph $(Sub(\mathcal{P}), \{(\mathcal{Q}, \mathcal{Q}') \mid \text{ex. } n \in N^{\mathcal{Q}} : \lambda^{\mathcal{Q}}(n) = \mathcal{Q}'\})$ is a tree with root $\mathcal{P}$. Further, we require that, for any $\mathcal{Q}, \mathcal{Q}' \in Sub(\mathcal{P})$ with $\mathcal{Q} \neq \mathcal{Q}'$, $N^{\mathcal{Q}} \cap N^{\mathcal{Q}'} = \emptyset$, i.e., the same node may not be re-used across several sub-graphs.

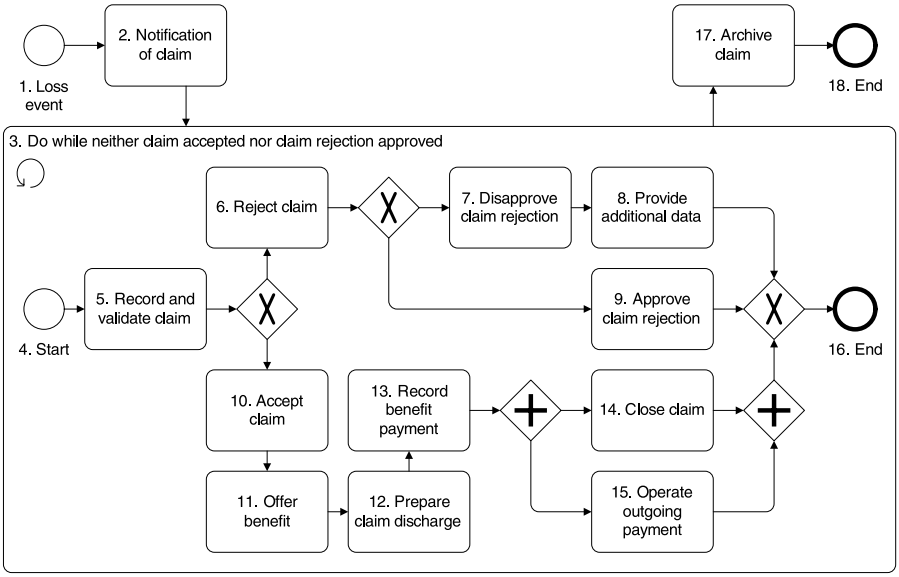


Fig. 2 Our running example: a claims handling process adapted from [45]

If $in(n) = \{e\}$, i.e., $in(n)$ is a singleton set, then we overload $in(n)$ to denote its single element, i.e., we identify $in(n)$ with e. Similarly, if $out(n) = \{e\}$ then we identify $out(n)$ with e. We denote $e_0 := out(n_0)$ and $e_+ := in(n_+)$.

Atomic process graphs are (sub-)graphs without loop nodes, whereas complex process graphs can contain loop nodes. A loop node is "implemented" by a subgraph, which, again, may be complex or atomic. The λ-function maps from loop nodes to sub-graphs. We require the compositional sub-graph structure to be a tree. We will generally refer to the root of the tree as the "top-level process", and refer to it with $\mathcal{P}$; sub-processes will generally be referred to with $\mathcal{Q}$. We do not distinguish between atomic and complex process graphs unless necessary, and we assume by convention a void function λ for atomic process graphs. If the context is clear, we omit the superscript indicating the process referred to. Note the difference between the set of nodes and edges of a single process graph, N, E, and the nodes and edges of all sub-graphs combined, $\mathcal{N}$, $\mathcal{E}$. Note also that *for singleton sets of incoming/outgoing edges, we overload in/out to denote both the respective set and its element*. While this abuses notation, it makes many of our formal statements a lot more readable, and will be used frequently.

Example 1 Figure 2 shows our running example, serving to illustrate many of our definitions. The example is based on a process model from the IBM Insurance Application Architecture [39] (a large collection of best practice process models from insurance industry); the model was published in [45]. It defines how a claim is handled by an insurer. For the purpose of our presentation, we slightly adapted and simplified the model.

The process starts when a loss event occurs (step 1 in Fig. 2). The insurance company is notified of the claim (step 2). Then a sub-process is repeated until either the claim has been accepted, or its rejection has been approved. In the sub-process, first the claim is recorded and validated. Then the insurer chooses to either accept or reject the claim.[5] In the former case, an according benefit is offered and paid, and the claim is closed (steps 10–15). In the latter case, the rejection needs to be either approved (step 9) or disapproved (step 7). If the rejection is disapproved, additional data needs be provided (step 8). Once the sub-process completes, the claim is archived in the final step, number 17 in Fig. 2. Note that step 14 can be executed in parallel to step 15, due to the parallel split and join nodes.

The semantics of process graphs are, similarly to Petri Nets, defined as a token game. Like the notation, this definition follows [81].

Definition 2 Let $\mathcal{P} = (N, E, \lambda)$ be a process graph. A *token marking* t of $\mathcal{P}$ is a function $t : \mathcal{E} \mapsto \mathbf{N}$ (the natural numbers). The *start marking* t_0 is $t_0(e) = 1$ if $e = e_0^{\mathcal{P}}$, $t_0(e) = 0$ otherwise. Let $\mathcal{Q}, \mathcal{Q}' \in Sub(\mathcal{P})$. Let t and t' be token markings. We say that there is a *transition* (or *token-transition*) from t to t' via n, written $t \xrightarrow{n} t'$, iff one of the following holds:

1. *Tasks, parallel splits and joins* (*tokens from INs to OUTs*).
   $n \in \mathcal{N}_T \cup \mathcal{N}_{PS} \cup \mathcal{N}_{PJ}$, for all $e_{in} \in in(n)$ we have $t(e_{in}) > 0$ and

$$t'(e) = \begin{cases} t(e) - 1 & e \in in(n) \\ t(e) + 1 & e \in out(n) \\ t(e) & \text{otherwise} \end{cases}$$

2. *Xor splits* (*token from IN to one OUT*).
   $n \in \mathcal{N}_{XS}$, $t(in(n)) > 0$, and there exists $e' \in out(n)$ such that

$$t'(e) = \begin{cases} t(e) - 1 & e = in(n) \\ t(e) + 1 & e = e' \\ t(e) & \text{otherwise} \end{cases}$$

3. *Xor joins* (*token from one IN to OUT*).
   $n \in \mathcal{N}_{XJ}$ and there exists $e' \in in(n)$ such that $t(e') > 0$ and

$$t'(e) = \begin{cases} t(e) - 1 & e = e' \\ t(e) + 1 & e = out(n) \\ t(e) & \text{otherwise} \end{cases}$$

[5] In the original model, this choice is encoded into a task node with non-deterministic outcome, governing which branch of the xor split to take. From the perspective of verification, i.e., as far as the possible execution traces are concerned, the two models are equivalent. The model as in Fig. 2 is more convenient for our illustration purposes.

4. *Entering a loop* (*push token downwards*).
   $n \in N_L^{\mathcal{Q}}$ with $\lambda^{\mathcal{Q}}(n) = \mathcal{Q}'$, $t(in(n)) > 0$, and

$$t'(e) = \begin{cases} t(e) - 1 & e = in(n) \\ t(e) + 1 & e = e_0^{\mathcal{Q}'} \\ t(e) & \text{otherwise} \end{cases}$$

5. *Repeating a loop* (*put token back on start*).
   $n = n_+^{\mathcal{Q}}$, $t(in(n)) > 0$, and

$$t'(e) = \begin{cases} t(e) - 1 & e = in(n) \\ t(e) + 1 & e = e_0^{\mathcal{Q}} \\ t(e) & \text{otherwise} \end{cases}$$

6. *Exiting a loop* (*push token upwards*).
   $n = n_+^{\mathcal{Q}}$ with $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$, $t(in(n)) > 0$, and

$$t'(e) = \begin{cases} t(e) - 1 & e = in(n) \\ t(e) + 1 & e = out(n') \\ t(e) & \text{otherwise} \end{cases}$$

An *execution path*, or *token-execution path*, is a transition sequence $t_0 \xrightarrow{n_1} t_1 \xrightarrow{n_2} t_2 \cdots t_{k-1} \xrightarrow{n_k} t$. A token marking t is *reachable*, or *token-reachable*, if there exists an execution path ending in t. We say t' is reachable from t ($t \to t'$) if there exists a transition sequence such that $t \xrightarrow{n_1} t_1 \xrightarrow{n_2} t_2 \cdots t_{k-1} \xrightarrow{n_k} t'$.

The tokens are carried by edges, and the execution status of the process is given by the position of all tokens, the token marking. $t(e)$ denotes the number of tokens on an edge e for a given token marking t. A node which passes a token from (one or all of) its incoming edge to (one or all of) its outgoing edge is said to be executed. Token passing from edge e to e' and from token marking t to t' is denoted as $t'(e) = t(e) - 1$ and $t'(e') = t(e') + 1$.

Task nodes are executed when a token on the incoming edge is consumed and a token on the outgoing edge is produced. The execution of an xor (parallel) split node consumes the token on its incoming edge and produces a token on one (all) of its outgoing edges, whereas an xor (parallel) join node consumes a token on one (all) of its incoming edges and produces a token on its outgoing edge. A loop node consumes a token from its incoming edge and produces a token on the start edge of the sub-graph that is associated with the loop node. The execution of the end node of a sub-graph consumes a token from its incoming edge e_+, and produces a token either on the start edge e_0 of the same process graph (repeating the sub-graph's execution) or on the outgoing edge of the loop node (exiting the loop). Note that this is a "do-while" semantics: the loop is executed at least once.

Soundness, first introduced by *Van der Aalst* in [74], is an important correctness criterion for business process models. The soundness property is defined for workflow nets, i.e., a Petri nets with one source and one sink. A workflow net is sound iff: (1) for

every token marking reachable from the source, there must exist a firing sequence to the sink (option to complete); (2) there is no reachable token marking which has a token both in the sink and in some other edge (proper completion); and (3) there are no dead transitions, i.e., for every transition there is an execution path that can fire it [74]. We adopt this definition. *Van der Aalst* shows that soundness of a Workflow net is equivalent to liveness and boundedness of the corresponding short-circuited Petri net. It follows easily from earlier results [79] that process graphs as per Definitions 1 and 2 map to free-choice Petri nets, where no output branch of an xor split $n \in N_{XS}$ can be disabled by other choices in the process graph.

We will later utilize a property of soundness related to so-called contact situations. A contact situation is a marking in which a node has tokens on its incoming edges as well as on its outgoing edges. Such a situation can never occur in a sound process.

Proposition 1 *Let $\mathcal{P} = (N, E, \lambda)$ be a sound process graph. Then the following does not exist: a token-reachable token marking t, a node $n \in \mathcal{N}$, and edges $e \in in(n), e' \in out(n)$ such that $t(e) > 0$ and $t(e') > 0$.*

Proof Assume a sound process graph $\mathcal{P}$ with a node n such that $t(e) > 0$ and $t(e') > 0$ for $e \in in(n), e' \in out(n)$. Since there is a path from n to the sink n_+ and $\mathcal{P}$ is sound, the nodes on the path can propagate the token from e' to n_+. Since $\mathcal{P}$ is free-choice, this firing sequence does not require the token from e. This means that there can still be a token on edge e after the sink n_+ has received a token via the firing from e'. The latter is a contradiction to condition (2) of the soundness assumption. □

2.2 Semantic annotations

For the semantic annotations, we use standard notions from logic, involving logical *predicates* and *constants*. Predicates provide the formal vocabulary, referring to properties of tuples of constants. Constants correspond to the entities of interest at process execution time.[6] We denote predicates with G, H, I and constants with c, d, e. *Facts* are predicates grounded with constants, *Literals* are possibly negated facts. If l is a literal, then $\neg l$ denotes l's opposite ($\neg p$ if $l = p$ and p if $l = \neg p$); if L is a set of literals then $\neg L$ denotes $\{\neg l \mid l \in L\}$. We identify sets L of literals with their conjunction $\bigwedge_{l \in L} l$. Given a set P of predicates and a set C of constants, $P[C]$ denotes the set of all literals based on P and C; if arbitrary constants are allowed, we write $P[]$.

A *theory* $\mathcal{T}$ is a closed (no free variables) first-order formula. Given a set C of constants, $\mathcal{T}[C]$ denotes $\mathcal{T}$ with quantifiers interpreted over C. For the purpose of our formal semantics, $\mathcal{T}$ can be arbitrary. For computational purposes, we will consider the following standard restrictions. A *clause* is a universally quantified disjunction of atoms, e.g., $\forall x. \neg G(x) \vee \neg H(x)$. There are two classes of clauses that are important in our analysis, because their propositional variants (without variables) are known to be tractable: Horn clauses [37] and binary clauses [3]. A clause is Horn if it contains

[6] Hence our constants correspond to BPEL "data variables" [58]; note that the term "variables" in our context is reserved for variables as used in logic, quantifying over constants.

at most one positive literal. A clause is binary if it contains at most two literals. A theory is Horn (binary) if it is a conjunction of Horn (binary) clauses.

An *ontology* Ω is a pair $(P, \mathcal{T})$ where P is a set of predicates (Ω's formal terminology) and $\mathcal{T}$ is a theory over P (constraining the behavior of the application domain encoded by Ω). For upper-bounding the complexity of reasoning over $\mathcal{T}$, we will assume *fixed arity*: a fixed bound on the arity of the predicates P. This is a realistic assumption because, according to modeling experience in AI and related areas, predicate arities are typically small. For example, in Description Logics [4] the maximum arity is 2.

Definition 3 An *annotated process graph* is a tuple $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ where (N, E, λ) is a process graph, $\Omega = (P, \mathcal{T})$ is an ontology, and the *annotation* $\alpha = \{\text{pre}, \text{eff}, \text{rcon}, (\text{con}, \text{pos})\}$ is defined as follows (where Π denotes the powerset):

- pre is a partial function $\text{pre} : \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\} \mapsto \Pi(P[])$ mapping a task node or global end node to a set of literals (its *precondition*).
- eff is a partial function $\text{eff} : \mathcal{N}_T \cup \{n_0^{\mathcal{P}}\} \mapsto \Pi(P[])$ mapping a task node or global start node to a set of literals (its *effect*).
- rcon is a partial function $\text{rcon} : \mathcal{N}_L \mapsto \Pi(P[])$ mapping a loop node to a set of literals (its *repetition condition*).
- con is a partial function $\text{con} : \{e \in \mathcal{E} | \text{ex. } n \in \mathcal{N}_{XS} : e \in out(n)\} \mapsto \Pi(P[])$ mapping an xor split's outgoing edge to a set of literals (its *condition*).
- pos is a partial function $\text{pos} : \{e \in \mathcal{E} | \text{ex. } n \in \mathcal{N}_{XS} : e \in out(n)\} \mapsto \{1, \ldots, |out(n)|\}$ mapping an xor split's outgoing edge to an integer (its *position*, encoding the evaluation order of the xor split).

We require that:

- There is no n such that $\text{pre}(n)$ is defined and $\mathcal{T} \wedge \text{pre}(n)$ is unsatisfiable.
- There is no n such that $\text{eff}(n)$ is defined and $\mathcal{T} \wedge \text{eff}(n)$ is unsatisfiable.
- There is no n such that $\text{rcon}(n)$ is defined and $\mathcal{T} \wedge \text{rcon}(n)$ is unsatisfiable.
- There is no e such that $\text{con}(n)$ is defined and $\mathcal{T} \wedge \text{con}(e)$ is unsatisfiable.
- $\text{con}(e)$ is defined iff $\text{pos}(e)$ is defined.
- There do not exist n, e, e' such that $e, e' \in out(n)$, $\text{pos}(e)$ and $\text{pos}(e')$ are defined, and $\text{pos}(e) = \text{pos}(e')$.

Any task node, as well as the top-level end node, may have a precondition pre. Any task node, as well as the top-level start node, may have an effect eff. Loop nodes may have a repetition condition rcon, and the outgoing edges of an xor split may have a condition con and an evaluation position pos. Note here the "may have". All the annotation functions are partial. This captures the situation where only parts of the process can be sensibly annotated, or where a developer wants to run verification tests on a model whose annotations have not yet been completed. This is important—forcing a user to fully annotate a process up front would adversely affect acceptance.[7]

[7] One may speculate that annotating an empty set of literals is equivalent to leaving the annotation undefined. Indeed, this is true (with some modifications regarding the role of pos for xor splits)—*except*

As stated, the annotation of task nodes—atomic actions that might be implemented by Web service invocations—in terms of logical preconditions and effects closely follows semantic Web service approaches such as OWL-S [1, 72] and WSMO [21, 66]. As also stated, all the involved sets of literals (pre(n), eff(n), con(e), rcon(n)) will be interpreted as conjunctions. (It is easy to extend our formalism to allow arbitrary formulas for pre(n), eff(n), con(e), rcon(n). Verification involving such formulas is likely to lead to harder decision problems; this remains a topic for future work.)

We now formally define the semantics of annotated process graphs.

Definition 4 Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph. Let C be the set of all constants appearing in any of the annotated pre(n), eff(n), rcon(n), con(e). A *state* s of $\mathcal{P}$ is a pair (t_s, i_s) where t is a token marking and i is a *logical interpretation* $i : P[C] \mapsto \{0, 1\}$. A *start state* s_0 is (t_0, i_0) where t_0 is as in Definition 2, and $i_0 \models \mathcal{T}[C]$, and $i_0 \models \text{eff}(n_0^{\mathcal{P}})$ in case $\alpha(n_0^{\mathcal{P}})$ is defined. Let $\mathcal{Q}, \mathcal{Q}' \in Sub(\mathcal{P})$. Let s and s' be states. We say that there is a *transition* from s to s' via n, written $s \xrightarrow{n} s'$, iff one of the following holds:

1. *Parallel splits and joins* (*straightforward*; *no change of logical interpretation*).
   $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{PJ}$, for all $e_{in} \in in(n)$ we have $t_s(e_{in}) > 0$, $i_s = i_{s'}$, and $t_s \xrightarrow{n} t_{s'}$ according to Definition 2.
2. *Xor joins* (*straightforward*; *no change of logical interpretation*).
   $n \in \mathcal{N}_{XJ}$, there exists $e' \in in(n)$ such that $t_s(e') > 0$, $i_s = i_{s'}$, and $t_s \xrightarrow{n} t_{s'}$ according to Definition 2.
3. *Xor splits* (*depends on condition*; *no change of logical interpretation*).
   $n \in \mathcal{N}_{XS}$, $t_s(in(n)) > 0$, $i_s = i_{s'}$, and

$$t_{s'}(e) = \begin{cases} t_s(e) - 1 & e = in(n) \\ t_s(e) + 1 & e = e' \\ t_s(e) & \text{otherwise} \end{cases}$$

   where either $e' \in out(n)$ and $\alpha(e')$ is undefined, or $e' = \text{argmin}\{\text{pos}(e) \mid e \in out(n),\ \alpha(e) \text{ is defined}, i_s \models \text{con}(e)\}$.
4. *Entering a loop* (*condition not tested due to do-while semantics*; *no change of logical interpretation*).
   $n \in N_L^{\mathcal{Q}}$ so that $\lambda^{\mathcal{Q}}(n) = \mathcal{Q}'$, with $t_s(in(n)) > 0$, $i_s = i_{s'}$, and

$$t_{s'}(e) = \begin{cases} t_s(e) - 1 & e = in(n) \\ t_s(e) + 1 & e = e_0^{\mathcal{Q}'} \\ t_s(e) & \text{otherwise} \end{cases}$$

for loop conditions. If such a condition is undefined, then we allow the process to non-deterministically choose whether to exit or repeat (cf. Definition 4 below). This cannot be encoded into any fixed set of literals. From a more general perspective, we prefer the notation using partial functions because it is more explicit.

5. *Repeating a loop* (*condition must be true; no change of logical interpretation*).
   $n = n_+^{\mathcal{Q}}$ so that $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$, $t_s(in(n)) > 0$, where either $\alpha(n')$ is undefined or $i_s \models \text{rcon}(n')$, with $i_s = i_{s'}$, and

$$t_{s'}(e) = \begin{cases} t_s(e) - 1 & e = in(n) \\ t_s(e) + 1 & e = e_0^{\mathcal{Q}} \\ t_s(e) & \text{otherwise} \end{cases}$$

6. *Exiting a loop* (*condition must be false*; *no change of logical interpretation*).
   $n = n_+^{\mathcal{Q}}$ so that $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$, $t_s(in(n)) > 0$, where either $\alpha(n')$ is undefined or $i_s \not\models \text{rcon}(n')$, with $i_s = i_{s'}$, and

$$t_{s'}(e) = \begin{cases} t_s(e) - 1 & e = in(n) \\ t_s(e) + 1 & e = out(n') \\ t_s(e) & \text{otherwise} \end{cases}$$

7. *Executing a task* (*affects the logical interpretation, may have ambiguous outcome*).
   $n \in \mathcal{N}_T$, $t_s(in(n)) > 0$, $t_s \xrightarrow{n} t_{s'}$ according to Definition 2, and either: $\alpha(n)$ is undefined and $i_s = i_{s'}$; or $i_s \models \text{pre}(n)$ and $i_{s'} \in \text{PMA-min}(i_s, \mathcal{T}[C] \wedge \text{eff}(n))$, where $\text{PMA-min}(i_s, \mathcal{T}[C] \wedge \text{eff}(n))$ is the set of all i that satisfy $\mathcal{T}[C] \wedge \text{eff}(n)$ and that are minimal with respect to the partial order defined by $i_1 \leq i_2$: iff $\{p \in P[C] \mid i_1(p) \neq i_s(p)\} \subseteq \{p \in P[C] \mid i_2(p) \neq i_s(p)\}$.

An *execution path* is a transition sequence $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} s_2 \cdots s_{k-1} \xrightarrow{n_k} s$, where s_0 is a start state. A state s is *reachable* if there exists an execution path ending in s. We say s' is reachable from s ($s \to s'$) if there exists a transition sequence such that $s \xrightarrow{n_1} s_1 \xrightarrow{n_2} s_2 \cdots s_{k-1} \xrightarrow{n_k} s'$. By $\mathcal{S}^{\mathcal{P}}$ we denote the set of reachable states; if the process referred to is clear from the context, we will sometimes omit the superscript $\mathcal{P}$.

A state s of the process now consists of the token marking t in combination with the logical interpretation i. The interpretation assigns truth values to all logical facts formed from the relevant predicates and constants. To avoid clumsiness of language, we will usually drop the "logical" in "logical interpretations". Also, we will often overload s with i_s, writing e.g. $s \models \phi$ instead of $i_s \models \phi$.

An execution of the process starts with a token on the outgoing edge of the start node, and with any interpretation that complies with the start node's effect (if any) and the implications of the logical theory. Parallel splits and joins, as well as xor joins, remain unaffected by the annotation. An xor split, in contrast, non-deterministically either selects an outgoing edge without annotation, or produces a token on the outgoing edge e with lowest position $\text{pos}(e)$ whose condition $\text{con}(e)$ is satisfied by the current interpretation, i_s. Note that this is a hybrid between a deterministic and a non-deterministic semantics, depending on how many output edges are annotated. If all edges are annotated, then we have a case distinction as handled in, e.g., BPEL, where the first case (smallest position) with satisfied condition is executed (Sect. 11.2 in [58]). If no edges are annotated, then the analysis must foresee that a case distinction may be created later on during the modeling. No assumptions can be made on the

form of that case distinction, so any possibility must be taken into account. Definition 2 just generalizes these two extremes in the straightforward way.

Entering a loop node also remains the same as in Definition 2, since each loop is executed at least once (do-while semantics). If the annotation of the loop node is undefined, then we can non-deterministically choose whether to repeat or exit the loop. If the repetition condition rcon is defined, then the loop is repeated iff the current state's interpretation satisfies rcon.

The execution of any routing node does not affect the logical interpretation. In contrast, the execution of an annotated task node (which is possible only if the current state satisfies the precondition pre) changes the interpretation according to the effect eff and the implications of $\mathcal{T}$. The tricky bit lies in the definition of the possible outcome states i'. The semantics defines this to be *the set of all* i' *that comply with* $\mathcal{T}$ *and* eff(n)*, and that differ minimally from* i. This draws on the AI literature for a solution to the *frame* and *ramification* problems. The latter problem refers to the need to make additional inferences from eff(n), as implied by $\mathcal{T}$. This is reflected in the requirement that i' complies with both. The frame problem refers to the need to not change the previous state arbitrarily—e.g., if an activity makes a payment via a credit card C1, then any other credit card C2 should not be affected. This is reflected in the requirement that i' differs minimally from i, such that there is no i'' that makes do with fewer changes. As explained in the introduction, this semantics follows the so-called possible models approach (PMA): the set PMA-min$(i_s, \mathcal{T}[C] \wedge \text{eff}(n))$ in the handling of task nodes as per Definition 4 is exactly as defined in the original PMA paper by Winslett [91]. The PMA underlies most formalisms relating to the execution of semantic Web services [5, 23, 47]. Alternative semantics from the AI literature (see [32] for an excellent overview) could be used in principle; this is a topic for future research.

We next give two examples. The first provides the semantic annotations for our running example.

Example 2 Consider again Fig. 2. The semantic annotations of the task nodes are listed in Table 1. The ontology specifies the following axioms:

(1) $\forall x : \neg\text{claimAccepted}(x) \vee \neg\text{claimRejected}(x)$
(2) $\forall x, y : \neg\text{benefitOffered}(x, y) \vee \neg\text{benefitPosted}(x, y)$
$\forall x, y : \neg\text{benefitOffered}(x, y) \vee \neg\text{benefitPaid}(x, y)$
$\forall x, y : \neg\text{benefitPosted}(x, y) \vee \neg\text{benefitPaid}(x, y)$
(3) $\forall x : \text{claimRejectionDisapproved}(x) \Rightarrow \neg\text{claimRejected}(x)$

Axiom (1) means that no claim may be both accepted and rejected at the same time; in other words, a claim may be in at most one of its possible states. Similarly, axioms (2) mean that no benefit may be in more than one of its possible states. Axiom (3) expresses a somewhat more subtle dependency, namely that a claim is no longer rejected if its rejection has been disapproved.

The following example illustrates the subtleties of the PMA semantics.

Example 3 We first illustrate the ramification problem. We show how ontology axioms may lead to "side effects", i.e., to literals that are not mentioned explicitly in the

Table 1 Preconditions and effects of all task nodes in the process from Fig. 2. Node labels abbreviated

Node	Precondition	Postcondition
1. Loss event		lossEvent(e)
2. Notification	lossEvent(e)	claim(c)
5. Record claim	claim(c)	claimRecorded(c), claimValidated(c)
6. Reject claim	claim(c)	claimRejected(c)
7. Disapprove rejection	claimRejected(c)	claimRejectionDisapproved(c)
8. Additional data	claim(c)	claimChanged(c)
9. Approve rejection	claimRejected(c)	claimRejectionApproved(c)
10. Accept claim	claim(c)	claimAccepted(c)
11. Offer benefit	claimAccepted(c)	benefit(b), benefitOffered(b, c)
12. Prepare discharge	benefitOffered(b, c)	dischargePrepared(c)
13. Record payment	benefitOffered(b, c), dischargePrepared(c)	benefitPosted(b, c)
14. Close claim	claimFinalized(c)	claimClosed(c)
15. Payment	benefitPosted(b, c)	benefitPaid(b, c)
17. Archive claim	claim(c)	claimArchived(c)

effect annotation of a task node n, but that change their value nevertheless when n is executed.

Say we have some process $\mathcal{P}_{expl}$ using the ontology presented in Example 2 (the exact form of $\mathcal{P}_{expl}$ is not relevant in what follows). Say we have a task node n that is annotated with $\text{eff}(n) = \{\text{claimRejected}(c)\}$, i.e., that rejects a claim. Say we execute n in a state s where the claim is accepted, i.e., $i_s(\text{claimAccepted}(c)) = 1$. Which are the possible outcome states s', with $s \xrightarrow{n} s'$? By the definition of PMA-min$(i_s, \mathcal{T}[C] \wedge \text{eff}(n))$ in Definition 4, any such state s' must satisfy the conjunction of effect $\text{eff}(n)$ and axioms $\mathcal{T}$. In particular, s' must satisfy $\forall x : \neg\text{claimAccepted}(x) \vee \neg\text{claimRejected}(x)$, cf. the previous example; we refer to this axiom with ϕ in the following. Together with the effect claimRejected(c), ϕ of course implies that s' must satisfy $i_{s'}(\text{claimAccepted}(c)) = 0$. That is, the value of claimAccepted(c) is changed as a side-effect of applying n. Sloppily formulated, when we apply the effect claimRejected(c) to s, then ϕ is invalidated and we need to "repair" it by switching the value of claimAccepted(c).

To illustrate the frame problem, we now show how ontology axioms may lead to ambiguities in the outcome state, namely to several alternative outcome states depending on how and to what extent the previous state is kept intact. Note first that the clause ϕ is binary (contains only two literals). This means that, whenever one of the literals is invalidated, the other literal follows necessarily. That is, there is only one option to "repair" the clause. For that reason, binary clauses do not lead to ambiguities. This is not so for clauses with more than two literals.

Say we extend $\mathcal{P}_{expl}$ by an axiom stating that, if two distinct reviewers accept a claim, then the claim is accepted overall. That is, we have the new predicates

claimAcceptedRevA(.) and claimAcceptedRevB(.), as well as the new axiom:

$$\forall x : \text{claimAcceptedRevA}(x) \wedge \text{claimAcceptedRevB}(x) \Rightarrow \text{claimAccepted}(x)$$

We refer to this axiom with ϕ' in what follows. Suppose about our state s from above that $i_s(\text{claimAcceptedRevA}(c)) = 1$ and $i_s(\text{claimAcceptedRevB}(c)) = 1$. Upon executing n, as pointed out above, c is no longer accepted. So ϕ' is no longer true and we must "repair" it. Since, in difference to ϕ, ϕ' is not binary, this spawns a non-trivial behavior of the minimal change semantics. There are three options to "repair" ϕ': falsify claimAcceptedRevA(c), falsify claimAcceptedRevB(c), or falsify both. The first two options each yield a resulting state $s' \in \text{PMA-min}(i_s, \mathcal{T}[C] \wedge \text{eff}(n))$. The third option does *not* yield a resulting state s' because that option is not a minimal change.

3 Verification tasks

We now formalize our four verification tasks, relating to precondition conflicts, effect conflicts, reachability, and executability. As a helper notation, we first need to define when two task nodes are parallel.

Definition 5 Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph. For $e_1, e_2 \in \mathcal{E}$, we say that e_1 and e_2 are *parallel*, written $e_1 \parallel e_2$, if there exists a token-reachable token marking t such that $t(e_1) > 0$ and $t(e_2) > 0$. For $n_1, n_2 \in \mathcal{N}_T$, we say that n_1 and n_2 are parallel, written $n_1 \parallel n_2$, if $in(n_1) \parallel in(n_2)$.

Definition 6 Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph.

- A node n is *reachable* iff either $n = n_0$ or there exist a reachable state s and an edge $e \in in(n)$ so that $t_s(e) > 0$.
- A node n is *executable* iff either $n \notin \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$ or, for all reachable states s with $t_s(in(n)) > 0$, we have that $s \models \text{pre}(n)$.

$\mathcal{P}$ is reachable iff all $n \in \mathcal{N}$ are reachable. $\mathcal{P}$ is executable iff all $n \in \mathcal{N}$ are executable.

Let $n_1, n_2 \in \mathcal{N}_T$, $n_1 \parallel n_2$. We say that

- n_1 has a *precondition conflict* with n_2 if $\mathcal{T} \wedge \text{eff}(n_1) \wedge \text{pre}(n_2)$ is unsatisfiable;
- n_1 and n_2 have an *effect conflict* if $\mathcal{T} \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ is unsatisfiable.

Consider first precondition and effect conflicts. For illustration it is useful to consider the special case where $\mathcal{T}$ is empty. A precondition conflict, e.g., then means there exists $l \in \text{eff}(n_1) \cap \neg\text{pre}(n_2)$. Generally, precondition and effect conflicts indicate that the semantic annotations of different task nodes are in conflict: n_1 jeopardizes the precondition of n_2, or n_1 and n_2 jeopardize each other's effects. If n_1 and n_2 are ordered with respect to each other, then this kind of conflict cannot result in ambiguities and should not be taken to be a flaw. Hence Definition 6 postulates that $n_1 \parallel n_2$.

It is debatable to some extent whether precondition/effect conflicts represent flaws, or whether they are a natural phenomenon of the modeled process. We view them as flaws, because in a parallel execution it may happen that the conflicting nodes are enacted at the same time.

Consider now the notions of reachable and executable task nodes n. Reachability is important because, if n is not reachable, then it is superfluous; this certainly indicates a problem in the process model.[8] As for executability, if n is not executable then the process may reach a state where n is active—it has a token on its incoming edge—but its prerequisites for execution are not given. If the process is being executed by a standard (non-semantic) engine (cf. Sect. 1.2), e.g. based on BPEL, then the implementation of n will be enacted regardless of the unsatisfied precondition, which may lead to undefined behavior and errors. In general, the possibility to activate a task without establishing its precondition indicates that the process model does not take sufficient care of achieving the relevant conditions in all possible cases.

For illustration, consider our running example, i.e., the process from Fig. 2 and its annotation as per Table 1. The task node "21. Close claim" has a precondition claimFinalized(c), but there is no activity whose effect provides this assertion. Thus, this task will not be able to execute when it is activated during runtime. While this may be due to faulty annotation (e.g., "19. Prepare claim discharge" may have claimFinalized(c) as an additional effect), it may also be the case that another activity "Finalize claim" is actually missing.

Reachability and executability are both temporal properties on the behavior of the process, and of course it may be of interest to allow arbitrary verification properties via a suitable temporal logic (see e.g., [60, 80]). We leave this open for future work. The focus on reachability and executability is, in that sense, an investigation of special cases. Note that these special cases are of practical interest, and perhaps more so than the fully general case allowing arbitrarily complex quantification which may rarely be used in practice.

Reachability can sometimes be established as a side-effect of executability.

Proposition 2 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be a sound annotated process graph where α is undefined for all edges and loop nodes. If $\mathcal{P}$ is executable, then $\mathcal{P}$ is reachable.*

Proof Let $n \in N \setminus \{n_0\}$. By definition, there exists a sequence $\vec{e}$ of edges from n_0 to n. By soundness and executability, and because none of the edges in $\vec{e}$, nor any loop nodes passed by $\vec{e}$, are annotated with a condition, one can easily use $\vec{e}$ to construct an execution path that reaches n. □

The overall methodology we propose for debugging a given process model is to first remove any precondition and effect conflicts, and thereafter to ensure the process is executable. Proposition 2 then implies that the process is correct with respect to all four verification tasks considered herein—provided there are no xor/loop conditions.

[8] To understand our definition of reachability, note that all nodes except parallel joins can be token-executed as soon as one of their incoming edges is active. For parallel joins, soundness implies that, if an incoming edge is active, then the join can be token-executed eventually.

The latter restriction is of no consequence as far as the identification of tractable classes is concerned, because, as we will see, in the presence of xor/loop conditions we cannot verify reachability/executability efficiently anyway.

4 Checking precondition and effect conflicts

We now start the investigation of the computational complexity associated with our verification tasks. Recall that this is important (apart from its theoretical interest) because response times in the targeted application must be instantaneous, cf. Sect. 1.2.

In this section, we are concerned with checking the existence of precondition and effect conflicts. We devise an algorithm consisting of two parts:

(1) We determine which pairs of tasks in the process are parallel.
(2) For each such pair the respective preconditions and effects are tested for conflicts.

Step (1) runs in time polynomial in the size of the process. Step (2) consists of two satisfiability tests of the ontology axioms in conjunction with the preconditions/effects. Hence the overall complexity is the same as that of testing satisfiability in the logic underlying the semantic annotations. We now consider the two steps in detail.

For step (1), one can re-use results from the Petri net literature. Namely, Kovalyov and Esparza [44] show that, for free-choice Petri nets, step (1) can be done in time cubic in the size of the process. In our own work, we devised an algorithm that exploits the particular structure of our control-flows, making do with quadratic time. Since this is not of central relevance for our focus herein—it does not concern the borderline between tractable and intractable classes—we give only a brief summary. A full presentation can be looked up in a longer version of this paper [88].

In Petri nets, the "concurrency relation" is the set of pairs of places that may contain a token at the same time, i.e., for which there exists a reachable marking putting a token on both places. Kovalyov and Esparza [44] devise an algorithm that computes the concurrency relation in time $O(|\mathcal{E}| * (|\mathcal{E}| + |\mathcal{N}|)^2)$, provided the net is free-choice. The algorithm is initialized with a set of known pairs of parallel nodes (transitions and places). It then performs local propagations based on a candidate set that evolves from this initial set. For example, if a place in the pre-set of a transition t is parallel to some other node n (i.e., a transition or place), then the post-set of t is set to be parallel to n.

We devise an algorithm called *M-propagation*, which computes parallelism for each subgraph $\mathcal{Q} \in Sub(\mathcal{P})$ in isolation, and re-constructs the overall parallelism from that. The runtime performance of the algorithm is $O(|\mathcal{E}|^2 + \Sigma_{\mathcal{Q} \in Sub(\mathcal{P})} |N^{\mathcal{Q}}| * |E^{\mathcal{Q}}| * \max^{\mathcal{Q}})$, where $\max^{\mathcal{Q}}$ is the maximum number of incoming or outgoing edges any node in $\mathcal{Q}$ has. Presuming that the latter number (the branching factor in splits and joins) is fixed, this means that our algorithm runs in time quadratic in the size of the process, by contrast to the cubic time taken by [44]'s algorithm. This improvement is possible because our control-flows are less general than free-choice Petri nets.[9]

[9] We are not aware of any work computing concurrency, for general free-choice nets, with a better runtime bound than [44]; neither are the authors of [44] aware of any such work. In personal communication, Javier Esparza conjectured that a faster algorithm does not exist.

The enabling property is that our loops are structured. We will show that, due to this structure, the concurrency relation of the overall process can be directly constructed from the separate concurrency relations computed for each sub-process individually. Within each sub-process, the process graph is cycle-free, which means that we can compute concurrency according to a topological order of the sub-process.

Consider a sub-process $\mathcal{Q}$. M-propagation determines pair-wise parallelism for all edges within $\mathcal{Q}$. The outcome of the algorithm is a function M^* mapping pairs of edges to Booleans, where $M^*(e, e') = 1$ iff $e \parallel e'$. M^* is computed by a propagation algorithm, in which M^* is maintained in the form of a matrix. Initially, all edges are set to not be parallel to themselves; everything else is set to be unknown. Each propagation step performs changes corresponding to some node n in the process whose predecessors have all been processed already. Note here that such a strategy is possible because the graph is acyclic. The propagation step sets matrix entries for the outgoing edges of n based on the matrix entries of its incoming edges. Task nodes, loop nodes, and xor splits/joins do not affect parallelism and so the entries are simply copied; the only subtlety is that, for xor joins n, the outgoing edge is parallel to an edge $e \notin in(n)$ iff *at least one* of its incoming edges is parallel to e. For parallel splits, the outgoing edges are marked to be pairwise parallel; otherwise the entries are copied. The outgoing edge of a parallel join n is parallel to an edge $e \notin in(n)$ iff all its incoming edges are parallel to e. We get:

Theorem 1 *Let $\mathcal{P} = (N, E, \lambda)$ be a sound process graph, and let $\mathcal{Q} \in Sub(\mathcal{P})$. For all $n_1, n_2 \in N_T^{\mathcal{Q}}$ we have $n_1 \parallel n_2$ iff $M^*(in(n_1), in(n_2)) = 1$. The time required to compute M^* is $O(|E^{\mathcal{Q}}|^2 + |N^{\mathcal{Q}}| * |E^{\mathcal{Q}}| * \max^{\mathcal{Q}})$, where $\max^{\mathcal{Q}}$ is the maximum number of incoming or outgoing edges any node in $\mathcal{Q}$ has.*

The runtime bound here holds because each propagation step takes time at most $|E^{\mathcal{Q}}| * \max^{\mathcal{Q}}$; $|N^{\mathcal{Q}}|$ steps are performed. The time for initializing the matrix is $O(|E^{\mathcal{Q}}|^2)$. To see that it suffices to compute the parallelism for each subgraph in isolation, observe the following. Say e is the incoming edge of a task node $n \in N_T^{\mathcal{Q}}$. Say e' is the incoming edge of a loop node $n' \in N_L^{\mathcal{Q}}$, and say that $\mathcal{Q}'$ is a sub-process of $\lambda^{\mathcal{Q}}(n')$, or is $\lambda^{\mathcal{Q}}(n')$ itself. If e and e' are parallel, then any edge $e'' \in E^{\mathcal{Q}'}$ is parallel to e. If e and e' are not parallel, then no edge $e'' \in E^{\mathcal{Q}'}$ can be parallel to e. Hence the overall parallelism relation can be read off the sub-graph relation and the parallelism within sub-graphs. Since the sub-graph relation can be stored in a way so that it can be looked up in constant time, the overall parallelism relation can be constructed from the relations of the individual sub-graphs in time $O(|\mathcal{E}|^2)$. Hence we obtain the overall bound $O(|\mathcal{E}|^2 + \Sigma_{\mathcal{Q} \in Sub(\mathcal{P})} |N^{\mathcal{Q}}| * |E^{\mathcal{Q}}| * \max^{\mathcal{Q}})$ as stated above. Note that parallelism according to Definition 5 is not affected by the semantic annotations, and hence M-propagation does not need to consider those; we will get back to this shortly. First, note how we can detect precondition and effect conflicts.

Corollary 1 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be a sound annotated process graph, and let M^* be the accumulated M-propagation result for $\mathcal{P}$. Then, for any two task nodes $n_1, n_2 \in \mathcal{N}_T$:*

- n_1 *has a* precondition conflict *with* n_2 *iff* $M^*(in(n_1), in(n_2)) = 1$ *and* $\mathcal{T} \wedge \text{eff}(n_1) \wedge pre(n_2)$ *is unsatisfiable*;
- n_1 *and* n_2 *have an* effect conflict *iff* $M^*(in(n_1), in(n_2)) = 1$ *and* $\mathcal{T} \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ *is unsatisfiable*.

Using Corollary 1, it is obvious how to perform step (2) in our overall algorithm. We detect precondition and effect conflicts simply by performing a loop over all pairs of edges e, e', and executing the described satisfiability tests for every pair where $M^*(in(n_1), in(n_2)) = 1$.

Since the computation of M^* is polynomial, and the number of edge pairs is quadratic, the only source of exponential worst-case complexity is the satisfiability testing. In other words, the borderline between tractable classes and intractable classes is the same as that of the satisfiability tests. When restricting, as we do, the preconditions and effects to be conjunctions of logical atoms, then the borderline is identical to that of the logic used for formulating the ontology axioms. Note that this property does not depend on the particular logic we selected in our framework.

The logic we chose here—clausal formulas—has been investigated in depth in the literature. In our particular setting, quantification is over a finite set of constants, hence we can compile into propositional logic. That compilation is exponential only in the nesting depth of the quantifiers in the clauses. For binary clauses, the nesting depth is bounded as soon as predicate arity is bounded, and hence binary formulas form a tractable class for fixed arity. This is a useful result because it concerns a non-trivial class of formulas. Binary clauses can be used to specify many common ontology properties, such as subsumption relations $\forall x : G(x) \Rightarrow H(x)$, attribute image type restrictions $\forall x, y : G(x, y) \Rightarrow H(y)$, and role symmetry $\forall x, y : G(x, y) \Rightarrow G(y, x)$. With what we have just derived, precondition and effect conflicts can be detected efficiently in processes using only such clauses.

A potentially yet much more useful class of formulas is that of Horn clauses. These allow the specification of implications of the form $G_1 \wedge \cdots \wedge G_k \Rightarrow H$, as well as integrity constraints of the form $\neg G_1 \vee \cdots \vee \neg G_k$. Clearly, this language is quite powerful. To ensure that compilation into propositional logic is polynomial-time, we need to directly assume a bound on nesting depth. This assumption is more debatable than fixed predicate arity, but is given in some typical examples such as the Horn rule defining role transitivity $\forall x, y, z : G(x, y) \wedge G(y, z) \Rightarrow G(x, z)$.

One important aspect of precondition and effect conflicts, as handled here, is the underlying definition of "parallelism". Definition 5 defines this exclusively based on the control-flow, i.e., two edges are parallel iff there exists a *token-reachable token marking* that puts a token on both. An alternative definition would be to say that two edges are parallel iff there exists a *reachable state* that puts a token on both. We refer in what follows to the former as *token parallelism*, and to the latter as *execution parallelism*. The difference between the two is that execution parallelism, in difference to token parallelism, takes into account the logical states. This can make a difference in the presence of xor split and loop conditions, and in the presence of non-executable task nodes: such constructs allow traversal only by a subset of the token execution paths, and hence there may be less pairs of parallel edges. If a pair of edges is execution-parallel, then it is token-parallel; but not vice versa.

To some extent, it is a matter of taste which notion of parallelism one uses as the underlying definition for precondition and effect conflicts. Token parallelism has two advantages. First, it is more conservative, not relying on the annotation of xor splits and loops, or on non-executable task nodes, to prevent the co-occurrence of conflicting preconditions/effects. Second, token parallelism leads to easier verification problems. Determining whether or not two edges are execution-parallel is hard. Namely, it is easy to see that the hardness results reported in Sect. 5 for reachability hold for that problem as well. It remains an open question how to develop verification techniques for dealing with precondition/effect conflicts based on execution parallelism.

5 Computational hardness of executability and reachability checking

Precondition and effect conflicts are "local" in the sense that they concern only the respective pair of nodes; whether or not some pair of nodes has a conflict does not influence whether or not some other pair has. This is not so for reachability and executability. If a node n is not reachable, then neither is any other node that can be reached only on paths through n. If a node n is not executable, then it can be traversed only by a particular subset of the execution paths that reach it, and hence some later node may become unreachable, or may become executable. In that sense, reachability and executability are more "global" phenomena. As we shall see now, this leads to some quite unfavorable properties regarding computational complexity. We consider four different decision problems:

(1) Is $\mathcal{P}$ executable, i.e., are all nodes $n \in \mathcal{N}$ executable?
(2) Is a particular $n \in \mathcal{N}$ executable?
(3) Is $\mathcal{P}$ reachable, i.e., are all nodes $n \in \mathcal{N}$ reachable?
(4) Is a particular $n \in \mathcal{N}$ reachable?

The difference between (1) and (2) is that, for (2), we admit the case where some other node $n' \in \mathcal{N}$ is not executable. Similar for (3) and (4). Most of the time, this difference does not have an impact on computational complexity. But in one particular case, (2) is **coNP**-hard while (1) is in **P**. That case is the class of what we have termed "basic processes":

Definition 7 Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$, $\Omega = (P, \mathcal{T})$, be an annotated process graph. $\mathcal{P}$ is *basic* if α is undefined for all edges and loop nodes, and $\mathcal{T}$ is binary.

In words, basic processes restrict the ontology axioms to mention at most two literals in each clause, and they restrict the annotations to not define any conditions for choosing xor branches or for determining whether or not a loop is repeated. As per Definition 4, this means that xor-branches are fully non-deterministic, i.e., the execution is free to choose which branch to take. Likewise, the execution is free to choose whether to repeat a loop or exit it. Intuitively, this simplifies the verification problem because the available choices are the same regardless of what the logical state is. The reason why binary clauses are easier for the verification is that, with

axioms consisting only of such clauses, the outcome of an activity—i.e., the logical state after execution of the activity—can be computed in polynomial time. The latter is obviously not the case for general clauses, and neither, as we shall see, is it the case for Horn clauses. We will get back to these issues below. For the moment, recall that binary clauses are a non-trivial language that can be used to specify many common ontology properties, cf. Sect. 4.

We prove in the remainder of this section that all the decision problems (1)–(4) are hard for any sensible generalization of basic processes, and that decision problems (2)–(4) are hard even for basic processes. We prove in the next section that decision problem (1) for basic processes is in **P**. In that sense, basic processes form a maximal tractable class for this kind of verification. Our focus is exclusively on this fact, i.e., for the hard cases we prove only hardness and do not consider membership. Establishing the precise complexity classes of the various problems is a topic for future research.

We now formally state the hardness results, and explain what the sources of complexity are. We first consider the decision problems relating to executability, then those relating to reachability.

Theorem 2 *Assume a sound annotated process graph* $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ *without effect conflicts, where* $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ} \cup N_L$, $\mathrm{eff}(n_0)$ *is a complete assignment, all predicates have arity* 0, *and either* $\mathcal{P}$ *is atomic or for all* $n \in N_L$ *we have* $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$. *The following problem is* $\mathbf{\Pi_2^p}$*-hard even if* $\mathcal{P}$ *is known to be reachable*:

- *Is* $\mathcal{P}$ *executable, or is* $n \in N$ *executable, given that* $\mathcal{P}$ *is basic except that* $\mathcal{T}$ *may involve arbitrary clauses*?

The following problems are **coNP***-hard even if* $\mathcal{P}$ *is known to be reachable*:

- *Is* $\mathcal{P}$ *executable, or is* $n \in N$ *executable, given that* $\mathcal{P}$ *is basic except that* $\mathcal{T}$ *may involve arbitrary Horn clauses*?
- *Is* $\mathcal{P}$ *executable, or is* $n \in N$ *executable, given that* $\mathcal{P}$ *is basic except that* $con(e)$ *may be defined for some* $e \in E$?
- *Is* $\mathcal{P}$ *executable, or is* $n \in N$ *executable, given that* $\mathcal{P}$ *is basic except that* $con(n)$ *may be defined for some* $n \in N_L$?
- *Is* $n \in N$ *executable, given that* $\mathcal{P}$ *is basic*?

Proof Sketch: The first two results, $\mathbf{\Pi_2^p}$-hardness when $\mathcal{T}$ is unrestricted and **coNP**-hardness when $\mathcal{T}$ is Horn, are entirely due to the complexity of computing activity outcomes, i.e., determining whether or not a particular literal is necessarily true after a task node has been executed. The control-flows used in the proof constructions are trivial, with only 3–4 nodes arranged in a sequence. The hardness is proved via the design of a particular set of axioms, and of the task node preconditions and effects. Those constructions are adapted from the proofs given by Eiter and Gottlob [28] to show that "belief update" is $\mathbf{\Pi_2^p}$-hard (**coNP**-hard) for unrestricted (Horn) formulas. In both cases, the axioms take a rather intricate form, assuming as input a Quantified Boolean Formula (QBF) formula $\psi = \forall X.\exists Y.\phi[X, Y]$ for the unrestricted case, and a QBF formula $\psi = \forall X.\phi[X]$ for the Horn case. One task node n_t has a single effect

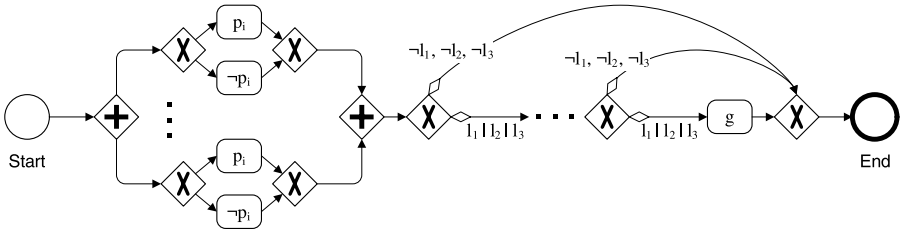


Fig. 3 Schematic illustration of 3SAT reduction for Theorem 2

literal t, and the following task node n has a precondition q. The constructions are such that ψ is valid iff q is necessarily true after n_t. The latter is, of course, the case iff n is executable.

Intuitively, the source of complexity in both cases is the need to figure out what is true in all possible "minimal" changes to the previous state. Any candidate for an outcome state can be tested for minimality easily, but the number of candidates is exponential. This complexity combines unfavorably with the complexity of reasoning about the axioms, and hence we get $\mathbf{\Pi_2^p}$ (rather than **coNP**) for unrestricted clauses and **coNP**(rather than **P**) for Horn clauses.

For all the other results, essentially the same construction can be used, reducing the complement of 3SAT to the respective decision problem. Figure 3 shows an illustration. The construction assumes as input a CNF formula ψ. It starts with a parallel split/join including one xor split/join for every variable p_i in the CNF formula, allowing the execution to set p to be either true or false. In this way, we allow to generate all possible truth value assignments. Afterwards, we filter those assignments, removing all but those that comply with all clauses in ψ. The construction used for doing so is easiest to understand for the case of xor splits annotated with conditions. For every clause $C = \{l_1, l_2, l_3\}$ in the formula, we include an xor split $split(C)$ with four outgoing branches. Three of those are annotated with the condition l_i, for $l_i \in \{l_1, l_2, l_3\}$, and lead to the xor split for the next clause. The fourth edge is annotated with $\neg l_1 \wedge \neg l_2 \wedge \neg l_3$, and leads directly to a final xor join just in front of the end node. The node marked g in Fig. 3 lies at the end of the sequence of xor splits $split(C)$, and is hence reachable if and only if the CNF is satisfiable. To obtain our proof for executability, we now simply introduce a new variable q, obtain a formula ψ' by inserting q into every clause, perform our construction for ψ', and make q a precondition of the node marked g. Then, g is executable iff q is true in all satisfying assignments to ψ', which is the case iff ψ is unsatisfiable. Note that g is definitely reachable because we are free to set q to be true.

If we are not allowed to annotate the outgoing edges of xor splits, then we have to find a replacement for those annotations. If we are allowed to annotate loop nodes, then we can replace the xor edge conditions with loops. Where before we had an outgoing edge annotated with l_i, we now have a loop with condition $\neg l_i$, meaning that the loop will be exited, and hence the path will be traversed, only if l_i is true. For xor edges annotated with a conjunction of literals, i.e., with $\neg l_1 \wedge \neg l_2 \wedge \neg l_3$, the construction is only slightly more complicated.

Finally, consider the case where we may neither annotate xor edges not loop nodes, but our decision problem is to figure out whether some particular node $n \in N$ is

executable. This decision problems allow the presence of other task nodes n' that are not executable. We can use such task nodes to filter truth value assignments, much in the same way as we did before. Indeed, wherever previously we had an xor outgoing edge e annotated with condition ϕ, we replace e with a construction $e' \to n \to e''$ where $\text{pre}(n) = \phi$. In this way, the execution paths that pass through e are the same as those that pass through e''. The only difference is that, in the new construction, some task nodes are definitely not executable—e.g., the task nodes encoding the conditions of the first clause are not, since any precondition l_i they have may be invalidated by setting the respective variable to the opposite value.

All of the constructions are made so that, trivially, all nodes are reachable. In that sense, the hardness of executability does not depend on the hardness of reachability. All the constructions comply with the restrictions mentioned in the claim (in particular, the parallel split/join in Fig. 3 can be replaced by a sequence). Except for the result regarding hardness of executability checking in basic processes, it does not matter to the constructions whether we ask for executability of a particular node, or of the whole process. Note that the last three results hold even for empty $\mathcal{T}$, i.e., without any clauses in the ontology. □

Note that the last proof argument, i.e., the one regarding executability of a particular node $n \in N$ in a basic process (decision problem (2) in the above), does not work if we ask whether *all* nodes are executable (decision problem (1) in the above). In such a setting, we can no longer use task nodes to "filter" execution paths, like we did here. Indeed, as indicated, decision problem (1) will turn out to be solvable in polynomial time, for basic processes.

Our hardness results for reachability are closely related to those for executability.

Theorem 3 *Assume a sound annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ} \cup N_L$, $\text{eff}(n_0)$ is a complete assignment, all predicates have arity 0, and either $\mathcal{P}$ is atomic or for all $n \in N_L$ we have $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$. The following problem is $\mathbf{\Sigma_2^p}$-hard:*

- *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary clauses*?

The following problems are **NP***-hard*:

- *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary Horn clauses*?
- *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is executable, and basic except that $con(e)$ may be defined for some $e \in E$*?
- *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is executable, and basic except that $con(n)$ may be defined for some $n \in N_L$*?
- *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic*?

Proof Sketch: Like for executability, the first two results, $\mathbf{\Sigma_2^p}$-hardness when $\mathcal{T}$ is unrestricted and **NP**-hardness when $\mathcal{T}$ is Horn, are entirely due to the complexity of computing activity outcomes, i.e., determining whether or not a particular literal is necessarily true after a task node has been executed. The constructions are

essentially the same as in the proof of Theorem 2. We consider a QBF formula $\psi = \forall X.\exists Y.\phi[X, Y]$ for the unrestricted case, and a QBF formula $\psi = \forall X.\phi[X]$ for the Horn case. We still have the task node n_t, and the following task node n. The only difference is that, now, n has a precondition $\neg q$, rather than q. Then, ψ is not valid iff q is not necessarily true after n_t, which is of course the case iff $\neg q$ might be true after n_t. Hence at least one execution path can traverse through n iff ψ is not valid, and so the node behind n is reachable iff ψ is not valid. Importantly, n is not executable: irrespectively of the precise form of ψ we can construct a path where n's incoming edge is active but $\neg q$ is false. In that sense, this proof of hardness for deciding reachability relies on the hardness of executability.

If we are allowed to annotate xor outgoing edges with conditions, then we can use exactly the same proof argument as used for the respective result of Theorem 2, except that we do not have to introduce the new variable q—as already stated in the proof of Theorem 2, the node marked g in Fig. 3 is reachable iff the CNF formula ψ is satisfiable. If we are allowed to have annotated loop nodes or non-executable task nodes, then, as before, we can use those to simulate xor conditions. Note here that, in the constructions using xor/loop conditions, the process is executable, i.e., those results do not rely on the hardness of executability.

All the constructions comply with the restrictions mentioned in the claim. In none of the constructions does it matter whether we ask for reachability of a particular node, or of the whole process. Note that the last three results hold even for empty $\mathcal{T}$, i.e., without any clauses in the ontology. □

As discussed in the proof sketch, for those results where executability is not explicitly mentioned in Theorem 3, the proof of hardness for deciding reachability is due to the hardness of deciding executability. This is a necessity, not a coincidence of our proof arguments. If the process in question is executable, then reachability follows trivially. Namely, in the respective classes of processes, no conditions are allowed at xor splits and at loops. With Proposition 2, this means that executability implies reachability. Hence any reduction of a computationally hard problem to these decision problems must make use of non-executable task nodes.

Table 2 provides an overview of our complexity results. This is rather dominated by intractable cases. From a practical perspective, these results mean that, for all the listed classes of processes, if we wish to build technology able to automatically verify any process in the class, then (unless $\mathbf{P} = \mathbf{NP}$) that technology will have to have runtime that is worst-case exponential in the size of the process. While that does not necessarily mean that the technology will be useless, obtaining sufficiently quick response times is certainly a challenge. We get back to this in Sect. 9.

Note that all the hardness results hold even without effect conflicts. The same restriction applies also to our positive result, i.e., our polynomial algorithm works correctly only when there are no effect conflicts. Hence, in the debugging of a process model under creation, effect conflicts should be found and removed first, and thereafter executability should be checked. Our positive result ensures that—to ascertain or disprove correctness of the overall process—this last step can be performed efficiently. We will now discuss that result in detail.

Table 2 Overview of our complexity results. The results for deciding whether $n \in \mathcal{N}$ is reachable are the same as those for deciding whether $\mathcal{P}$ is reachable. All results are valid for (only for, in the case of the membership result) processes without effect conflicts. For all hardness results, we can further impose that $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ} \cup N_L$, that $\text{eff}(n_0)$ is a complete assignment, that all predicates have arity 0, and that either $\mathcal{P}$ is atomic or for all $n \in N_L$ we have $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$. Every hardness result for executability holds even if $\mathcal{P}$ is known to be reachable; the same is true vice versa for the reachability results, except those marked (*), where executability implies reachability by Proposition 2

Class of $\mathcal{P}$	$\mathcal{P}$ exec?	$n \in \mathcal{N}$ exec?	$\mathcal{P}$ reach?
Basic but $\mathcal{T}$ unrestricted	$\mathbf{\Pi_2^p}$-hard	$\mathbf{\Pi_2^p}$-hard	$\mathbf{\Sigma_2^p}$-hard (*)
Basic but $\mathcal{T}$ Horn	**coNP**-hard	**coNP**-hard	**NP**-hard (*)
Basic but con(e) may be def	**coNP**-hard	**coNP**-hard	**NP**-hard
Basic but con(n) may be def	**coNP**-hard	**coNP**-hard	**NP**-hard
Basic	in **P**	**coNP**-hard	**NP**-hard (*)

6 Polynomial-time executability checking for basic processes

As stated, we now presume that the process under consideration is basic, and that it does not contain any effect conflicts. We design a polynomial-time algorithm determining whether or not the process is executable. Using this verification test as a debugging facility, the process modeler can remove flaws from the process until it is executable. Then, by Proposition 2, the process is also reachable, and thus has been established to be correct with respect to all four verification tasks considered herein.

Our verification algorithm computes, for every edge e in the process, what we call e's *state intersection*: the literals that must hold true whenever e carries a token. Formally:

$$\mathcal{SI}(e) := \bigcap_{s \in \mathcal{S}, t_s(e) > 0} s$$

where a state is written as the set of literals it satisfies, i.e., $s = \{l \in P[C] \mid i_s \models l\}$. Note that executability of a task node n is equivalent to $\text{pre}(n) \subseteq \mathcal{SI}(in(n))$.

The state intersections are computed by a fixpoint algorithm that performs propagation steps over the process structure, maintaining for every edge e a set $I(e)$ of literals. We call the algorithm I-propagation. Its formal definition may be a little hard to read at first, but its underlying ideas are straightforward. At any point during the execution of the algorithm, $I(e)$ is an approximation of $\mathcal{SI}(e)$. The approximation is aimed to be sound, i.e., to guarantee that literals outside of $I(e)$ are not contained in e's state intersection.[10] At its start, I-propagation assigns a trivially sound approximation: $I(e_0)$ is set to $\overline{\text{eff}(n_0)}$, i.e., the effect of the start node n_0; for all other edges e, $I(e)$ is set to $P[C]$, i.e, the set of all possible literals. Thereafter, the propagation commences. Each propagation step corresponds to a simple form of local reasoning about the consequences of (hypothetically) executing a particular process node n, determining which literals may be invalidated by such execution. This results in

[10] That guarantee is not actually given in general, but is given under the conditions formally stated below in Theorem 4. I-propagation is easiest to understand by thinking of it as a sound approximation.

a new approximation $I'(out(n))$ of the state intersection of n's outgoing edge. That new approximation is, in general, unrelated to the previous one $I(out(n))$: neither is guaranteed to contain the other. That is to say, the new approximation may (A) be tighter than the previous one, i.e., $I(out(n)) \setminus I'(out(n)) \neq \emptyset$. But also, due to looping behavior, the new approximation may (B) be less tight than the previous one, i.e., $I'(out(n)) \setminus I(out(n)) \neq \emptyset$. We will give examples for (A) and (B) below. To obtain the best of both approximations, we simply intersect them, i.e., we update our approximation of $\mathcal{SI}(out(n))$ to be $I(out(n)) \cap I'(out(n))$. Clearly, with this updating rule, the sets $I(e)$ shrink monotonically over the propagation steps. The propagations are performed until a fixpoint is reached, i.e., until no more literals can be removed.

A detailed explanation of I-propagation, including a number of examples, will be given below. Now, we give the formal definition. For the remainder of the paper, given a node n we write $\overline{\text{eff}(n)} := \{l \in P[C] \mid \mathcal{T} \wedge \text{eff}(n) \models l\}$ if $\alpha(n)$ is defined; else we set $\overline{\text{eff}(n)} := \emptyset$.

Definition 8 Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph. Say α uses the constants C. We define the function $I_0 : \mathcal{E} \mapsto 2^{P[C]}$ as $I_0(e) = \overline{\text{eff}(n_0)}$ if $e = out(n_0)$, $I_0(e) = P[C]$ otherwise. Let $I, I' : \mathcal{E} \mapsto 2^{P[C]}$, $n \in \mathcal{N}$. We say that I' is the *propagation of I at n* iff one of the following holds:

1. $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{XS}$ and

$$I'(e) = \begin{cases} I(e) \cap I(in(n)) & e \in out(n) \\ I(e) & \text{otherwise} \end{cases}$$

2. $n \in \mathcal{N}_{PJ}$ and

$$I'(e) = \begin{cases} I(e) \cap (\bigcup_{e' \in in(n)} I(e')) & e = out(n) \\ I(e) & \text{otherwise} \end{cases}$$

3. $n \in \mathcal{N}_{XJ}$ and

$$I'(e) = \begin{cases} I(e) \cap (\bigcap_{e' \in in(n)} I(e')) & e = out(n) \\ I(e) & \text{otherwise} \end{cases}$$

4. $n \in \mathcal{N}_T$ and

$$I'(e) = \begin{cases} I(e) \cap (\overline{\text{eff}(n)} \cup (I(in(n)) \setminus \neg\overline{\text{eff}(n)})) & e = out(n) \\ I(e) \setminus \neg\overline{\text{eff}(n)} & e \parallel in(n) \\ I(e) & \text{otherwise} \end{cases}$$

5. $n \in N_L^{\mathcal{Q}}$ so that $\lambda^{\mathcal{Q}}(n) = \mathcal{Q}'$ and

$$I'(e) = \begin{cases} I(e) \cap I(in(n)) & e = e_0^{\mathcal{Q}'} \\ I(e) & \text{otherwise} \end{cases}$$

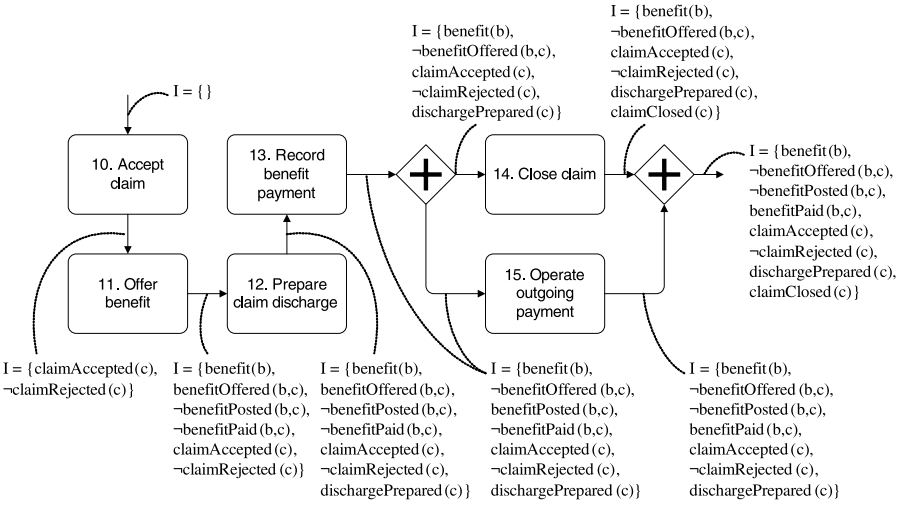


Fig. 4 Outcome of I-propagation on a part of the example process from Fig. 2. The literals lossEvent, claim, claimRecorded, and claimValidated are present in all shown sets, and are omitted

6. $n = n_+^{\mathcal{Q}}$ so that $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$ and

$$I'(e) = \begin{cases} I(e) \cap I(in(n)) & e = e_0^{\mathcal{Q}'} \\ I(e) \cap I(in(n)) & e = out(n') \\ I(e) & \text{otherwise} \end{cases}$$

If I' is the propagation of I at n, then I' is *valid*, written $I \xrightarrow{n} I'$, iff $I \neq I'$.

An I*-propagation path* is a sequence $I_0 \xrightarrow{n_1} I_1 \xrightarrow{n_2} I_2 \ldots I_{k-1} \xrightarrow{n_k} I_k$ so that, for all $0 \leq j < k$, I_{j+1} is the valid propagation of I_j at n_{j+1}. If I^* is the endpoint of an I-propagation path, and if I^* is a fixpoint, i.e., for all nodes n the propagation of I^* at n is not valid, then I^* is called an I*-propagation result*.

Figure 4 shows the outcome of I-propagation on part of our example process. The literals lossEvent(c), claim(c), claimRecorded(c), and claimValidated(c) are true for all shown edges, hence they are omitted in the figure. We now explain the various propagation steps of Definition 8 in detail. The cases discussed in the following correspond directly to the cases in Definition 8.

Splits: If n is a parallel split or an xor split, the propagation simply forwards I from the incoming edge to every outgoing edge. This is because splits do not change the state of the world. Note in this context that the outgoing edges of the parallel split in Fig. 4, i.e., the incoming edges of nodes 14 and 15, have different I sets. This is due to the effect of node 15: propagation over that node affects the incoming edge of node 14 as per the second line of Definition 8 case 4; see also the explanation of task nodes below.

Parallel joins: Say e' is n's outgoing edge. We intersect $I(e')$ with the union of the sets $I(e)$ for all of n's incoming edges e. This is justified per the assumed absence of

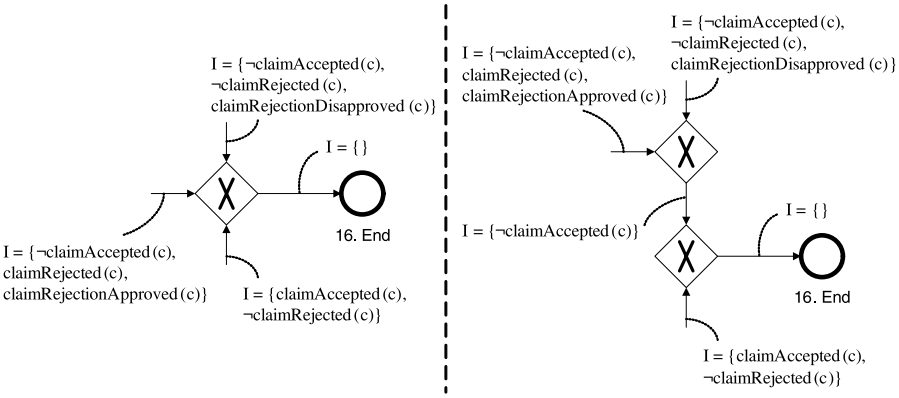


Fig. 5 I-propagation over an xor join on a part of the example process from Fig. 2, showing only the literals for claims in terms of acceptance, rejection, rejection approval, and rejection disapproval. *Left*: directly on the xor join of Fig. 2. *Right*: a variant where the original xor join has been split up to illustrate the behavior at xor joins

effect conflicts. A parallel join can only fire if there is a token on all of its incoming edges; for all such cases we know that the literals $I(e)$ of these edges hold. Since there are no effect conflicts, the sets $I(e)$ do not contradict each other. Hence, for a literal l to be guaranteed to hold after execution of n, it suffices if l is guaranteed to hold on one of the incoming edges. (In the presence of effect conflicts, the outcome of parallel branches depends on the order of execution.) See the parallel join in Fig. 4 (subsequent to steps 14 and 15) for illustration: the I sets of the 2 incoming edges are combined, cf. benefitPaid(b, c) and claimClosed(c).

Xor joins: We set $I(e')$ to the intersection of the sets $I(e)$ for all of n's incoming edges. This is adequate because a literal l holds after an xor join only if all paths leading to the join guarantee that l holds (any one of the paths may be executed).[11] Figure 5 shows this behavior in two variants, for the part of the process including the sub-graph's end node 16 and the preceding xor join. On the left-hand side, no literal is in the intersection of the I sets of all three edges, and hence the I set of the outgoing edge is empty. On the right-hand side, for better illustration a variant is shown where the single xor join has been split up into two xor joins. In that variant, ¬claimAccepted(c) holds on both incoming edges of the first join node, and hence holds also on the outgoing edge of that node. Note that this is not the case for claimRejected(c), because, due to the ontology axioms in our example, claimRejectionDisapproved(c) implies ¬claimRejected(c).

Task nodes: These are the most complicated propagation steps. By $\overline{\text{eff}_n}$ we denote n's explicit and implicit effects, containing not only eff(n) but also all its implications together with $\mathcal{T}$. Note here that, if $\mathcal{T}$ consists of binary clauses and predicate

[11] Note here that we throw away any information we might have regarding what happened only in some of the alternative executions (consider Fig. 5). For executability checking, that information is irrelevant. It is an open topic to explore extended algorithms maintaining more information. Note that the size of such information is likely to grow exponentially in the number of consecutive xor joins, unless clever restrictions or approximations are made.

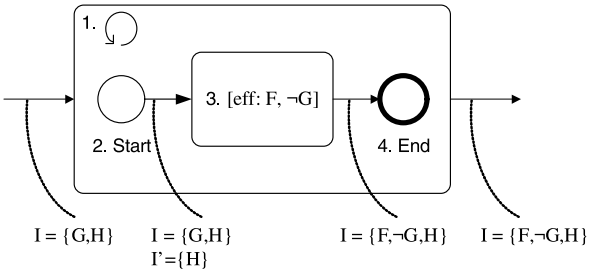


Fig. 6 I-propagation across an end node, in an illustrative toy example

arity is fixed, then $\overline{\text{eff}}_n$ can be computed in polynomial time. Say n has the incoming edge e and the outgoing edge e'. Three different actions need to be performed. (1) We write $\overline{\text{eff}}_n$ into $I(e')$. (2) We copy every literal l from $I(e)$ to $I(e')$, unless $\neg l$ is already present in $I(e')$. (3) We go through the list of all edges e'' that are parallel to e (by M-propagation we know which edges to consider), and remove from $I(e'')$ all literals l where $\neg l$ is contained in $\overline{\text{eff}}_n$.

(1) and (2) are direct consequences of the semantics of annotated task nodes, cf. Sect. 2.2. (1) must be done simply because any effect forces a direct change on the world. (2) must be done since the world is required to change minimally, i.e., if a property is true before and is not affected, then it is still true. (3) deals with the case where an edge e'' parallel to e' inherited a literal l which is in conflict with $\overline{\text{eff}}_n$ (l cannot be established by the effect of a task node connected to e'' since that would be an effect conflict). In this situation, l is *not* guaranteed to hold whenever e'' carries a token: n may be fired, leading to $\neg l$. This is best understood using an example. Consider Fig. 4. The task node n we consider is step 15, "Operate outgoing payment". The preceding parallel split, let's denote it by n', has two outgoing edges. One of those leads to n; the other one, which we denote with e'', leads to step 14, " Close claim". Say n' fires, putting a token on both of the edges. In this situation, we know due to the execution of step 13, "Record benefit payment", that benefitPosted(b, c) holds. Due to $\mathcal{T}$, ¬benefitPaid(b, c) is also certain to hold. Accordingly, I-propagation over n' (as explained above) keeps these literals in $I(e'')$. However, say n fires next. Then e'' still carries a token, but both literals have been inverted. Hence benefitPosted(b) and ¬benefitPaid(b) may be false when e'' carries a token. The two literals must thus be removed from $I(e'')$. (3) does that. The annotation of e'' in Fig. 4 shows the outcome.

Loop nodes: For a loop node n with $\lambda(n) = \mathcal{Q}$, we intersect $I(e_0^{\mathcal{Q}})$ with $I(in(n))$. This is adequate because an execution of the process will always enter into a loop, due to the defined do-while-semantics.

End nodes of sub-graphs: At an end node of a sub-graph, the loop can either be repeated or exited. In basic process graphs, this decision is non-deterministic. We thus intersect both $I(out(n'))$ and $I(e_0^{\mathcal{Q}})$ with $I(e_+^{\mathcal{Q}})$, where $\lambda(n') = \mathcal{Q}$.

The behavior of this procedure is illustrated with a toy example in Fig. 6: a loop node $(1, n)$ contains a sub-graph $(\mathcal{Q})$ with only a start (2), task $(3, n')$, and end node (4). $I(in(n))$ comprises two literals G, H, which are copied onto $I(e_0^{\mathcal{Q}})$. The task node (3) has the effect $\text{eff}(n') = \{\neg G, F\}$, thus replacing G with $\neg G$ on its outgoing edge, and adding F. This results in the set $I = \{F, \neg G, H\}$ at the outgoing

edge of n'; in what follows we refer to this set as I_+. The propagation over the end node of the loop copies, as described above, I_+ to the start edge $e_0^{\mathcal{Q}}$ of the loop—more precisely, I_+ is intersected with $I(e_0^{\mathcal{Q}})$, yielding a changed $I(e_0^{\mathcal{Q}})$ (denoted as I' in Fig. 6). Notably, bot (A) and (B) from the discussion above Definition 8 can be observed, i.e., neither I_+ nor $I(e_0^{\mathcal{Q}})$ are a priori tighter approximations of $\mathcal{SI}(e_0^{\mathcal{Q}})$. As for (A), I_+ serves to remove further invalid literals from $I(e_0^{\mathcal{Q}})$: G was beforehand presumed to be a member of the state intersection, which it is not. As for (B), I_+ contains the additional literals $\neg G, F$ which $I(e_0^{\mathcal{Q}})$ already concluded to not be contained in $\mathcal{SI}(e_0^{\mathcal{Q}})$.

It is important to note here that step (2) for task nodes can be done in such a simple way only because $\mathcal{T}$ is restricted to disjunctions of at most 2 literals. The minimal change semantics as per Definition 4 can get quite intricate in the presence of more complex $\mathcal{T}$—cf. the proofs of Theorems 2 and 3. For illustration, reconsider Example 3. We execute a task node with effect claimRejected(c) in a state s where both claimAcceptedRevA(c) and claimAcceptedRevB(c) hold. Suppose that the two facts necessarily hold prior to n, i.e., claimAcceptedRevA(c), claimAcceptedRevB(c) $\in I(in(n))$. As discussed in Example 3, execution of n invalidates either claimAcceptedRevA(c) or claimAcceptedRevB(c). In particular, after n, neither of the two facts is guaranteed to hold—although their opposites are not implied by $\mathcal{T} \wedge \text{eff}(n)$! Step (2) does not recognize this, and wrongly includes both facts into $I(out(n))$. Situations like this (and other more complicated situations) cannot appear when $\mathcal{T}$ consists of binary clauses only; hence for basic process graphs (1) and (2) suffice.

We now analyze the properties of I-propagation formally. We first observe that I-propagation yields a unique result, terminates in polynomial time, and is correct provided the process is executable. The reader may be alerted at this point since, as stated above, we wish to use I-propagation for testing *whether* the process is executable. We will show below that this is indeed possible. The analysis of I-propagation is more natural, and easier to understand, when first considering the more restricted case where executability holds a priori.

Theorem 4 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an executable basic sound annotated process graph without effect conflicts. Say we run I-propagation on $\mathcal{P}$. There exists exactly one I-propagation result I^*. For all $e \in \mathcal{E}$, $\mathcal{SI}(e) = I^*(e)$. With fixed arity, the time required to compute I^* is polynomial in the size of $\mathcal{P}$.*

Proof Sketch: The main result is correctness. First, we prove that, provided $\mathcal{P}$ is executable and basic, it does not affect the state intersections to remove $\mathcal{T}$ and replace each eff(n) with $\overline{\text{eff}(n)}$. The behavior for a single state and task node is the same in both cases because, intuitively, $\overline{\text{eff}(n)}$ captures all forced consequences of $\mathcal{T}$, and with binary clauses every consequence of $\mathcal{T}$ is forced. The claim for the overall process follows because the process structure is not changed, and logical states do not affect the possible token executions when there are no xor/loop conditions and when all task nodes are executable. We can hence, without loss of generality, assume that $\mathcal{T}$ is empty (we get back to this below the proof).

Our core argument focuses on the literals that are "deleted" during I-propagation, i.e., the literals l and edges e for which there exists an I-propagation step so that, after that step, $l \notin I(e)$. We show soundness and completeness of the literal deletion: for soundness, if l is removed at e then there exists a reachable state s with $t_s(e) > 0$ and $s \not\models l$; vice versa for completeness.

Completeness is shown as follows. Given any execution path $s_0 \xrightarrow{n_1} s_1 \cdots s_{k-1} \xrightarrow{n_k} s_k$, we construct a sequence $I_1, \ldots, I_k$ by performing I-propagation of I_j at n_j if that is possible, i.e., if it results in any changes, and setting $I_{j+1} := I_j$ otherwise. Obviously, a sub-sequence of $I_1, \ldots, I_k$ corresponds to an I-propagation path. We show by induction over j that, for all e where $t_{s_j}(e) > 0$, we have $s_j \models I_j(e)$; in particular, if $s_j \not\models l$ then $l \notin I_j(e)$. The proof distinguishes the different kinds of nodes n_j. For example, say n_j is a task node, and $t_{s_{j+1}}(e) > 0$. Then either $e = out(n_j)$, or $e \parallel out(n_j)$. In the former case, we have the induction hypothesis for $in(n_j)$ because we must have $t_{s_j}(in(n_j)) > 0$; in the latter case we have that property for e itself. In both cases, it is easy to see that $s_{j+1} \models I_{j+1}(e)$ because the propagation step over n_j deletes all literals that become false when executing n_j.

Soundness is more tricky to prove. We assume an I-propagation path $I_0 \xrightarrow{n_1} I_1 \cdots I_{k-1} \xrightarrow{n_k} I_k$. We prove by induction over j that, for every e and l where $l \notin I_j(e)$, there exists an execution path ending in a state s so that $t_s(e) > 0$ and $s \not\models l$. Again, we distinguish the different kinds of nodes n_j. The most tricky kind are parallel joins. By the definition of I-propagation over parallel join nodes, we either have (a) $l \notin I_j(e)$ or (b) $e = out(n_j)$ and for every $e_i \in in(n_j) : l \notin I_j(e_i)$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. For case (b), we need to construct a reachable state s' where $s' \not\models l$ and $t_{s'}(e_i) > 0$ for all $e_i \in in(n_j)$; in s', we can execute n_j and are done. However, the induction hypothesis gives us such a state only for every *individual* $e_i \in in(n_j)$. For each $e_i \in in(n_j)$, we have a state s_i with $t_{s_i}(e_i) > 0$ and $s_i \not\models l$. We prove in a separate lemma that, for every set of edges e_i that are pairwise parallel and where such states s_i exist, we can construct a state s' as desired. The proof is by induction over the process structure. The lemma makes use of the absence of effect conflicts: if $l \in \text{eff}(n), l \in \text{eff}(n')$ but $l \notin \mathcal{SI}(out(n)), l \notin \mathcal{SI}(out(n'))$ due to effect conflicts, then we may never be able to falsify l at both e and e' together (an example for this is given below this proof sketch).

The soundness arguments for the other kinds of nodes n_j are similar but have a more direct connection between induction hypothesis and claim. For xor split/end nodes, to construct the desired execution paths we exploit the fact that, in a basic process, an execution path may choose any outgoing edge/may choose to repeat or exit the loop; similar for executability and task nodes.

Uniqueness follows directly from correctness. As for computation time, the main observation consists in an upper bound for the number of propagation steps performed by I-propagation until a fixpoint I^* is reached. Denote with $\|I\|$ the total number of literals annotated by I, in sum over all edges in the process. Since I-propagation steps always intersect their outcome with the previous outcome, and since every valid step must make at least one change, it is obvious from Definition 8 that whenever we have $I \xrightarrow{n} I'$ we must have $\|I\| > \|I'\|$. If $\|I'\| = 0$, then certainly a fixpoint is reached. Obviously $\|I_0\| \leq |\mathcal{E}| * |P[C]|$, so this is the desired upper bound. With

fixed arity, $|P[C]|$ is $O(|P| * |C|)$, where C is the set of constants mentioned by α. Overall, we can derive that the runtime of I-propagation is $O(|P[C]|^3 + |\mathcal{N}| * |P[C]| * \max_{\text{eff}} + (|P[C]| * \max_E + |\mathcal{E}| * |P[C]|) * |\mathcal{E}| * |P[C]|)$, where $\max_{\text{eff}}$ is the maximum number of effect literals any task node has, and $\max_E$ is the maximum number of incoming or outgoing edges any node has. □

Note that the runtime is low-order polynomial. If we assume that $|P|$ is fixed, then the runtime is roughly cubic in the size of the process graph.

It may seem odd that we can assume an empty $\mathcal{T}$ without loss of generality. It is important to note here that this holds only for the purpose of executability checking. When removing $\mathcal{T}$ (and accordingly extending effects), the space of reachable states does *not* stay the same, because $\mathcal{T}$ imposes restrictions on how particular pairs of literals can be combined. However, that does not affect the state intersections.

To illustrate why we need to disallow effect conflicts, consider the following example. We have a parallel split node n_{split}, a parallel join node n_{join}, and four task nodes $n_{1\neg p}$, $n_{2\neg p}$, n_{1p}, n_{2p} where each $n_{i\neg p}$ has the effect $\neg p$ and each n_{ip} has the effect p. The edges are $(n_{split}, n_{1\neg p})$, $(n_{split}, n_{2\neg p})$, $(n_{1\neg p}, n_{1p})$, $(n_{2\neg p}, n_{2p})$, $e_1 := (n_{1p}, n_{join})$, $e_2 := (n_{2p}, n_{join})$. That is, we have two parallel branches, on each of which p is first made false and then made true. Consider the edges going into the join node, e_1 and e_2. Even though e_1 is the outgoing edge of a task node with effect p, we have $p \notin \mathcal{SI}(e_1)$ because $n_{2\neg p}$ may be executed while e_1 still carries a token—note here the effect conflict between n_{1p} and $n_{2\neg p}$. The same is true of e_2, i.e., $p \notin \mathcal{SI}(e_2)$. So for each e_i there exists a reachable state where e_i is active and p is false. However, there does *not* exist a reachable state where *both* e_i are active and p is false! If both e_i are active then either n_{1p} or n_{2p} was executed last, so p is necessarily true. In consequence, $p \in \mathcal{SI}(out(n_{join}))$; since $p \notin \mathcal{SI}(e)$ for any $e \in in(n_{join})$, this means that, in the presence of effect conflicts, the state intersection of the outgoing edge of a parallel split is no longer a function of the state intersections of the incoming edges. It is currently an open issue whether this problem can be overcome by maintaining supplementary information during I-propagation, in addition to the sets $I(e)$; see also the outlook in Sect. 9.

Theorem 4 presumes that the process is executable. For checking whether that is the case, of course we cannot make that assumption. The key observation here is that, if the process is not executable, and even if xor/loop conditions are defined, the outcome of I-propagation is conservative.

Lemma 1 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph without effect conflicts, which is basic except that α may be defined for edges and loop nodes. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $e \in \mathcal{E}$, $\mathcal{SI}(e) \supseteq I^*(e)$.*

Proof Sketch: This is surprisingly easy to show by considering the modified process $\mathcal{P}_0 = (N, E, \lambda, \Omega, \alpha^0)$ which is like $\mathcal{P}$ except that all preconditions and all xor/loop conditions have been removed. Obviously, Theorem 4 applies to $\mathcal{P}_0$ and so in particular we get completeness of literal deletion, i.e., $\mathcal{SI}^{\mathcal{P}_0}(e) \supseteq I_0^*(e)$ for all $e \in \mathcal{E}$, where $\mathcal{SI}^{\mathcal{P}_0}(e)$ is the state intersection for $\mathcal{P}_0$, and I_0^* is an I-propagation result for

$\mathcal{P}_0$. The claim then follows because the execution paths of $\mathcal{P}$ are a subset of those of $\mathcal{P}_0$; hence the state intersections in $\mathcal{P}$ are supersets of the respective ones in $\mathcal{P}_0$; hence $\mathcal{SI}(e) \supseteq \mathcal{SI}^{\mathcal{P}_0}(e) \supseteq I_0^*(e)$. □

Putting Theorem 4 and Lemma 1 together, we immediately get our two main results regarding executability checking.

Theorem 5 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be a basic annotated process graph without effect conflicts. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then $\mathcal{P}$ is executable iff for all $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\} : \text{pre}(n) \subseteq I^*(in(n))$.*

Proof Sketch: First, clearly $\mathcal{P}$ is executable iff, for every $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\} : \text{pre}(n) \subseteq \mathcal{SI}(in(n))$. If $\mathcal{P}$ is executable, then Theorem 4 applies, meaning that $I^*(in(n)) = \mathcal{SI}(in(n))$ and hence $\text{pre}(n) \subseteq I^*(in(n))$. If $\mathcal{P}$ is not executable, then Lemma 1 applies, meaning that $I^*(in(n)) \subseteq \mathcal{SI}(in(n))$. So if $l \in \text{pre}(n) \setminus \mathcal{SI}(in(n))$, then $l \in \text{pre}(n) \setminus I^*(in(n))$ and hence $\text{pre}(n) \not\subseteq I^*(in(n))$. □

Theorem 6 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph without effect conflicts, which is basic except that α may be defined for edges and loop nodes. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$, if $\text{pre}(n) \subseteq I^*(in(n))$ then n is executable.*

Proof Sketch: Say that n is not executable. Then $l \in \text{pre}(n) \setminus \mathcal{SI}(in(n))$, for some l. With Lemma 1 we have $I^*(in(n)) \subseteq \mathcal{SI}(in(n))$. Hence $l \in \text{pre}(n) \setminus I^*(in(n))$, and $\text{pre}(n) \not\subseteq I^*(in(n))$ in contradiction. □

Theorem 5 means that, for basic processes, we can use I-propagation for executability checking, as desired. We simply run I-propagation up to its fixpoint and check whether all precondition literals remained at the respective edges. Theorem 6 constitutes a weaker kind of verification, providing a sufficient but not necessary test for correctness. Note, however, that this test is applicable to *individual task nodes*, not only to the overall process. If, for any individual task node n, $\text{pre}(n) \subseteq I^*(in(n))$, then n is executable. This result holds (provided $\mathcal{T}$ is binary) even in the presence of xor conditions, loop conditions, and (other) non-executable task nodes. Hence, even under such circumstances, we can use Theorem 6 to ascertain the correctness of some nodes, and to point out potential bugs regarding others.

7 Prototypical implementation

The presented work is implemented as a back-end prototype that is used in three front-end applications at SAP Research. The back-end component implements the algorithms devised herein, on a generic level as presented. The front-end components are integrated prototypes with graphical user interfaces tailored for specific scenarios.

The back-end prototype is implemented in Java. The implementation is comparatively simple: the main algorithms, i.e., M-propagation and I-propagation, each consist only of a few hundred lines of code. Even without any code optimizations, the

backend component performs quite well. For example, a non-trivial process with 40 nodes and 46 edges was processed in 0.2 seconds on a Pentium M CPU running at 1.6 GHz, with negligible memory consumption. Considering in addition to this that the worst-case behavior is low order polynomial in the number of nodes and edges, performance is unlikely to be problematic in typical real-world settings.

Two of the front-end prototypes use our verification methods within a process modeling environment, in an online setting where the modeler frequently checks for bugs, as described in Sect. 1.2. Note that it is of paramount importance that the annotation function α as per Definition 3 is allowed to be *partial*: otherwise, we would force the modeler to completely annotate every task in the process up-front, rather than adding the annotations as needed and useful.

Presuming that the process in question is basic as per Definition 7, based on our techniques the modeler debugs a process as follows: (1) find and remove any precondition/effect conflicts; and (2) thereafter remove any bugs leading to non-executable task nodes. Once (1) and (2) are completed, reachability follows by Proposition 2. Hence, at that point, the resulting process is correct with respect to all four verification tasks identified herein.

In what follows, we give a brief outline of the three front-end prototypes, in turn. We conclude the section with a discussion of the practical difficulty of writing ontologies.

7.1 Maestro

Maestro is a process modeling tool, developed at SAP Research, for the BPMN notation. Maestro is mainly used for early prototyping. The tool has been extended to allow for semantic annotation, as proposed in [12]. It has been integrated with service discovery [50] and service composition [33] facilities, as well as the verification functionality described herein. The tool was presented in the form of a demonstration at DASFAA 2009 [14]. A screenshot of the tool is shown in Fig. 7. In the screenshot, a semantic annotation can be seen for the "Customer Quote" document, shown as a BPMN data object above the "Submit quote to customer" task. The object and task are associated via an edge showing the semantic annotation: "$<$ approved (Customer Quote)" indicates that "approved" is a precondition; "$>$ sent (Customer Quote)" indicates that "sent" is an effect. Both annotations refer to the status of the Customer Quote object, prior respectively after execution of the task. In the situation shown, I-propagation has just been performed, and the task "Submit quote to customer" is highlighted in red to indicate that it is not executable. The reason for that—the precondition whose truth is not guaranteed—is shown in the text window at the bottom of the screenshot.

7.2 SAP NetWeaver BPM process composer

SAP NetWeaver BPM Process Composer is the modeling part of SAP's future product for Business Process Management:

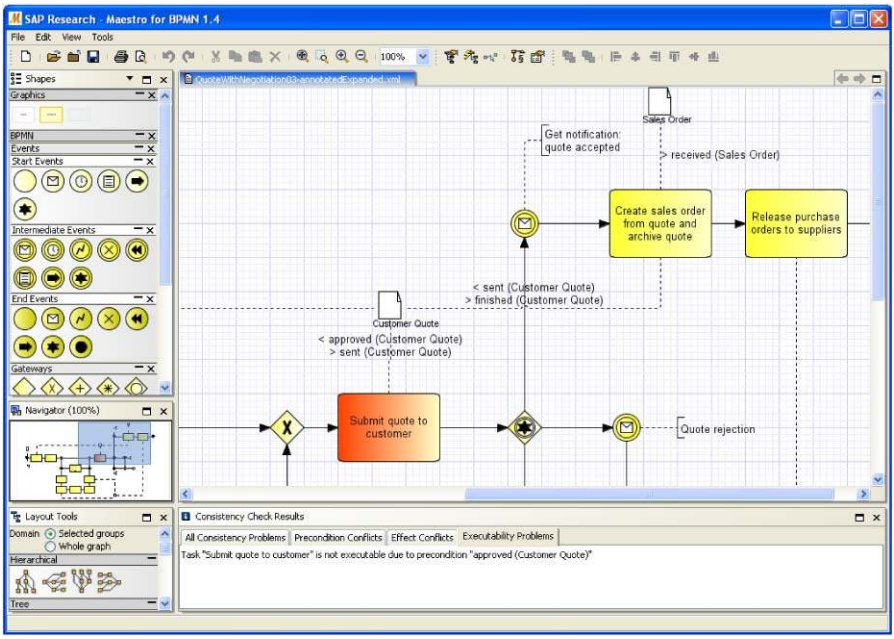


Fig. 7 Screenshot of Maestro for BPMN. Semantic annotations are associated with edges between BPMN data objects and tasks. The task "Submit quote to customer" is highlighted in red to indicate that it is not executable

"*SAP NetWeaver BPM delivers a suite of state-of-the-art, standards-based tools that enable customers to quickly and efficiently model processes and execute them without time-consuming, error-prone coding. It leverages the service-enabled functionality of SAP Business Suite applications, and of third-party software, to create and modify processes. This ultimately leads to significant increases in speed, flexibility, quality and time to value.*"

We implemented a research extension to the tool suite, comprising semantic annotation of tasks, service composition methods [33], as well as our verification methods. The extensions may undergo a pilot customer evaluation project in the next years. Due to the complexity of such a commercialization process, and due to re-structuring activities within SAP, a more definite statement cannot be made at the time of writing.

Screenshots are shown in Fig. 8. The bottom part of the figure shows how a non-executable task is highlighted, in a fashion similar (if differently visualized) to Maestro above. A more remarkable feature of the prototype is *the ease of creating semantic annotations*. As shown in the top part of Fig. 8, from the point of view of the user this amounts to selecting values in drop-down menus. The user double-clicks a task to open the window shown in the bottom right of the screenshot. She can then select the "Initial conditions" (the preconditions) and "Goals" (the effects) for the task. Both is done via selecting business objects from a drop-down menu (left hand side in each of "Initial conditions" respectively "Goals"), and afterwards selecting their desired status value from another drop-down menu (right hand side, shown open for

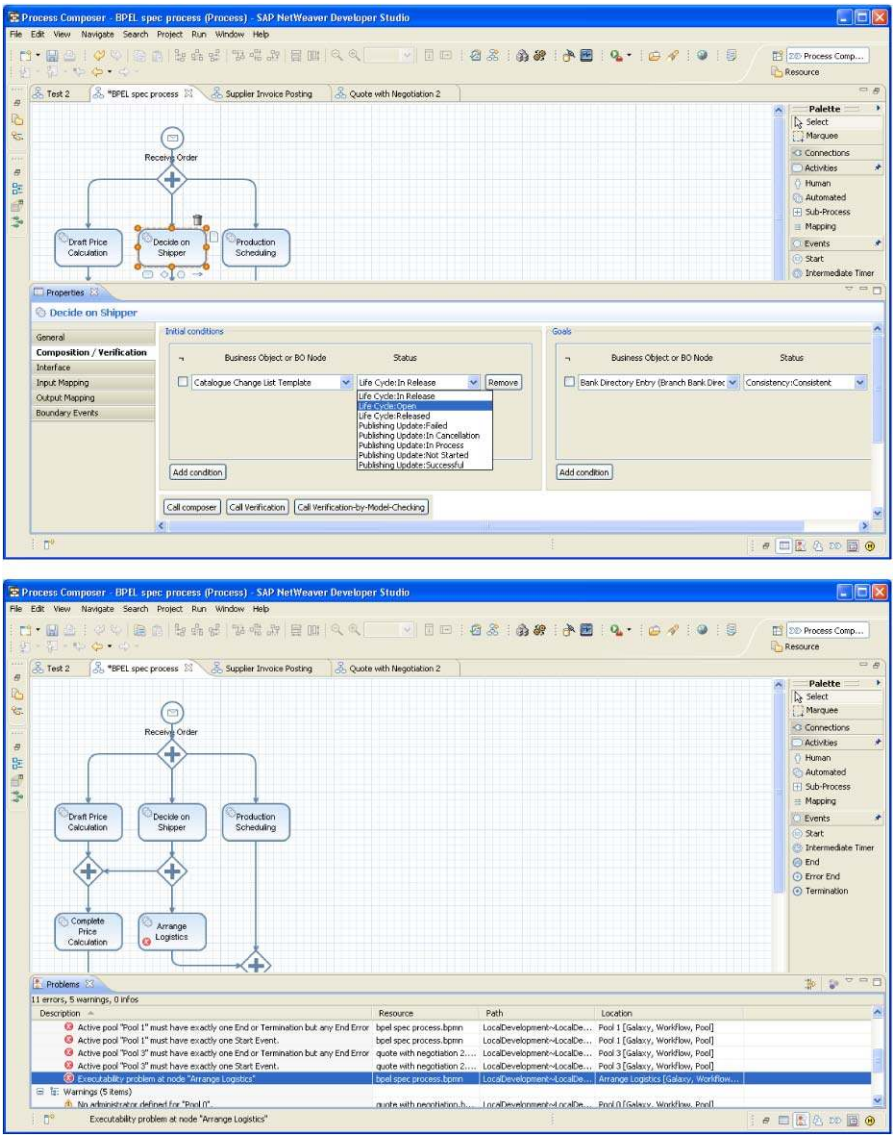


Fig. 8 Screenshots of the SAP NetWeaver BPM Process Composer, with our extensions. Semantic annotation (*top*) is done via simple drop-down menus. Executability checking (*bottom*) is a push-button operation highlighting erroneous tasks

"Initial conditions"). This interface, of course, assumes that the business objects and their possible states have been modeled beforehand. Such modeling is, fortunately for our approach, a well-established activity at SAP anyway; we will get back to this below in Sect. 7.4. What's important to note for now is that, once such a model has been established, doing the actual annotations is a matter of a few commonly used mouse movements.

The question remains how easy the concept of semantic annotations, and their suitable content, will be to grasp, for SAP customers. The prospects appear to be good. The annotations are made in terms of business object states. That is the language of *business users*—standard terms from business administration, logistics, etc.—*not* the language of developers. It can be expected that SAP customers will be familiar with the language, and find it easy to deal with. That said, an actual customer evaluation has not yet been performed, and remains an open topic.

7.3 Automatic service configuration for service brokers

This application differs considerably from the BPM scenarios above, for which the presented work was originally performed. However, the application uses the same back-end component. The application addresses service marketplaces, which are nowadays steadily gaining momentum. In a service marketplace, services from a variety of service providers can be registered, published, advertised, discovered, and brokered. Marketplace brokers manage the "front-desk" of service marketplaces, brokering the trade between service providers and consumers. In separate work [89], we provide technology aimed at lowering the entrance barrier for service providers, i.e., reducing their effort for registering services with a broker. Such registration requires utilization of the broker's service delivery management components. For doing so, a suitably combined process must be composed. Our technology helps with this activity, based on simple semantic annotations not unlike the ones used in the aforementioned prototypes. The technology can automatically filter the available options, and it can automatically suggest process fragments. The back-end provided herein, specifically I-propagation, is required by both these features for determining the possible execution states of the involved processes.

7.4 How to obtain the ontologies?

The creation of ontologies is a time-consuming and error-prone task. Clearly, this may seriously impede the practical usefulness and uptake of our technology—a problem we share with many approaches in the Semantic Web area. The cost of modeling is the prize to pay, and at this stage it cannot be predicted how serious the problem is, or under which circumstances. That said, our specific approach has two aspects that make the problem seem manageable at least in some applications:

1. *Ontologies are a possibility, not a requirement.* Our methodology *allows* the specification of ontologies; it does not *enforce* them. Recall that ontologies $(P, \mathcal{T})$ consist of two parts: their terminology (the predicates P) and their axiomatization (the theory $\mathcal{T}$). Clearly, the main difficulty is to obtain $\mathcal{T}$; P will often be easy to come by based on pre-existing terminology (cf. point 2 below). For our algorithms to work, only P is required. More importantly, the problem we address remains substantial, even without axiomatizations, both in theory and practice. All our hardness results regarding xor/loop conditions, as well as reachability and individual-task executability checking, hold with empty $\mathcal{T}$. Similarly, axiomatizations are no source of complications for I-propagation. That algorithm is tailored

for the use with binary clauses, which may be compiled away prior to even starting the algorithm (cf. the proof of Theorem 4). From a practical perspective, it is quite conceivable that one may make do without complicated axiomatizations. For example, such axiomatizations are not present in our extension to the SAP NetWeaver BPM Process Composer. Even when reducing the annotation to simple keywords—predicates without any arguments—the verification may still yield useful insights into the process structure, so long as the same keywords are used across different tasks in the process.

2. *We can leverage existing models at SAP.* As hinted above in Sect. 7.2, our approach combines naturally and effortlessly with existing models and methodologies at SAP, particularly those underlying SAP NetWeaver. In the relevant applications, data is warehoused in the form of the content of business objects. Individual transactions change these contents, and thereby the information contained in the warehouse. Each task in a business process corresponds—depending on the level of abstraction of the process—to a single transaction or to a combination thereof. In any case, the behavior of the task can be naturally expressed in terms of how it affects the relevant business objects. Now, annotating the latter in full is not feasible; a single object may contain thousands of attributes. However, motivated by precisely that complexity, SAP has developed models at an intermediate level of abstraction, where the content of each object is represented in terms of a number of high-level status variables. While these models were originally designed for very different purposes, *we can use these very same models as the input for our technology.* Indeed, this is exactly what underlies the implementation shown in Fig. 8. In other words, leveraging existing models at SAP, we obtain the ontology for free! Plus, the ontology is exhaustive (>400 business objects and their status variables), and the potential customers are already familiar with its terminology.

Interestingly, the SAP model even gives an example of how ontologies—and specifically the clausal theories we consider herein—can be put to productive use. There are several kinds of dependencies between status variables that are not represented in the current model. Current work at SAP Research addresses this shortcoming. The dependencies can be conveniently expressed in terms of the clausal theories we consider herein. Formalizing all the dependencies of course involves intensive cooperation with the relevant development groups at SAP. This is currently ongoing.

8 Related work

We give an overview of related work, and we discuss in detail the most relevant technical connections to our work. Such a connection mainly exists to the area of Petri nets, which has been used as the basis for control-flow verification, and where tractable classes have been identified. Since our focus is on checking the properties of a model, the field of model checking is, of course, related as well. Finally, there is a growing body of work extending process models and their verification beyond control-flow. We review these three areas in turn, after briefly discussing our own related work.

8.1 Own related work

Some predecessors of the presented work have discussed, from a general perspective, the use of semantic technology in BPM [84, 87]. Our tools have been showcased as demonstrations, partially related to this work [13, 14].

Earlier versions of the presented work have been published at the Semantic BPM Workshop 2008 [86], and at ECOWS 2008 [85]. The present paper goes far beyond these works not only in terms of detail of write-up, but also in terms of technical results. Most importantly, the previous versions restricted the processes to be plain DAGs, i.e., to not contain any loops. The addition of loops required a significant extension to nearly all technical devices of the paper, most particularly to I-propagation whose correctness is much easier to understand and prove when not dealing with loops.

There are some works that build on I-propagation, for the purpose of compliance checking. One line of work [34, 90] uses the outcome of I-propagation to approximate the truth status of clausal compliance constraints, i.e., clauses constraining the desired process states (not to be confused with the clauses herein, which form part of the *execution semantics* of the process). Another line of work modifies I-propagation (without loops) to propagate reparation chains over deontic logic primitives [30]. None of these works has I-propagation, as devised herein, as its original contribution.

8.2 Petri nets

Petri net theory has come up with a wealth of complexity results for various classes of Petri nets, including in particular tractability results for a number of restricted classes. We already discussed in Sect. 4 how one of these results [44] can be exploited to determine parallel task nodes in our framework, which is half of the job of finding all precondition/effect conflicts. Task node parallelism does not depend on the annotation, hence this is an application of Petri net theory to non-annotated process graphs. We can also obtain an application to annotated process graphs, via compiling such graphs into Petri nets. However, the results obtainable in this way are substantially weaker than what we proved herein.

How can annotated process graphs be compiled into Petri nets? First, in the presence of ontology axioms, clearly such a compilation is not possible, at least not in a straightforward/natural way. Petri nets do not cater for a "minimal change semantics" of transitions between states. Note that this is quite fundamental, as is reflected e.g. by the complexity of determining the truth of a literal in the outcome state, which is **coNP**-hard for Horn axioms and $\mathbf{\Pi_2^p}$-hard in general, cf. the proof of Theorem 2. Encoding this into a Petri net would require to encode each task node into some form of worst-case exponential search.

If there are no ontology axioms (or, as far as executability checking is concerned, if the axioms are binary, cf. the proof of Theorem 4), then a straightforward compilation exists. Encode each task as a transition, and encode edges as places. Joins and splits can then be encoded using the rules defined in [75]. Loops, i.e., transitions into and out of sub-graphs, are encoded in the straightforward fashion. Next, enumerate all

facts that can be built from the predicates and constants. Create an additional place for each fact, as well as one for its negation. Add an arc for each precondition/effect literal to the respective place; similarly, encode xor/loop conditions.

Apart from the process structure, we also need to express our verification tasks in terms of Petri net queries. Reachability in the annotated process model is equivalent to the question whether the control flow pre-place of a task may ever carry a token. Executability in the annotated process model is equivalent to the question whether, whenever the control flow pre-place of a task carries a token, the precondition pre-places carry a token as well. To ask whether a particular task node n is executable, we need to ask for every precondition p of n whether a state is reachable where the control flow pre-place of n carries a token, but p does not. To ask whether the overall process is executable, we need to ask these questions for every task node.

To test reachability in our annotated process graph, we hence need to be able to test, in a Petri net, whether a given place can be active. This is a fairly common query for Petri nets. To test executability, we need to be able to test whether a given place p can be active while another place p' is not. This is a rather unusual query. It is related to what has been termed "implicit places" (see e.g. [8, 29, 76]). An implicit (or "redundant") place is a place p' that always carries a token if any other place p does, where p and p' occur together in the input of any transition. Note that this notion refers to *all* transitions and places p, while what we are interested in is the connection (if any) between p' and *one particular* place p. It is an open question whether techniques for detecting implicit places can be adapted to perform this kind of test. We remark that the only known polynomial-time technique to detect implicit places is the detection of "structural implicit places" [8, 76], which are a special case of implicit places; i.e., this technique corresponds to a sufficient but not necessary criterion for executability and is hence not a verification method in our sense.

What we can derive are two tractable classes for checking reachability. A closely related topic was previously investigated by [57], who are concerned amongst other things with a Petri net based formalization of compositions of semantic Web services, and with verification of reachability. Narayanan and McIlraith [57] use a formalism that does not encompass any ontology axioms, but that is otherwise closely related to ours. They use a Petri net encoding similar to what we sketched above. They state two tractability results, based on restricting the process in a way so that the compiled Petri net becomes free-choice [25], respectively conflict-free [38]. These results, in the form stated in [57], are slightly flawed—the stated restrictions do not suffice to make the Petri net free-choice/conflict-free. However, the results can easily be repaired, by imposing more restrictions. We have:

(1) If every literal l appears in at most one task node precondition, xor condition, or loop condition, then the compiled Petri net is free-choice.
(2) If the process has no loops and no xor splits, and for every fact p we have that either p is not made false by any task node, or is contained in the precondition of at most one task node, then the compiled Petri net is conflict-free.

Both for free-choice and conflict-free Petri nets, it can be decided in polynomial time whether there exists a reachable marking activating a given place. Hence, (1) and (2) identify tractable classes for checking reachability in annotated process graphs.

These tractability results are not implied by our results—we propose to establish reachability as a side-effect of establishing executability, in a class of processes where reachability checking is **NP**-hard. Note that class (2) is a subset of the tractable class we identify,[12] and a very restricted one at that. Hence this class is considerably more impractical than ours. Class (1), on the other hand, could be useful since it allows xor/loop conditions, albeit in a restricted form. The tractability of reachability in class (1) clearly is complementary to the results proved herein.

8.3 Model checking

Model checking is concerned with checking the properties of some formal model of a piece of software or hardware under consideration; see [18] as an entry point into the vast amount of literature. There are two key differences to our work. First, model checking has not been concerned with ontologies, i.e., with ontology axioms that form part of the model to be checked. Like for Petri nets, this is a fundamental difference to our approach, and a natural encoding into traditional model checking formalisms is not possible. Hence we can only consider the restricted case where the axioms are empty (or, as far as executability checking is concerned, where all axioms are binary clauses).

The second major difference to model checking is that model checking has traditionally not been concerned with the identification of tractable fragments, which is the core contribution of our work. Model checking is usually concerned with very general formalisms, which are far from tractable. The focus then is on theoretical analysis of algorithms addressing such formalisms, and on the development of search techniques for enhancing empirical performance, such as symbolic representations (e.g. [16, 17]), constraint propagation (e.g. [19, 51, 71]), search space reduction (e.g. [36, 73]), and clever implementation techniques (e.g. [7]). In our view, the main importance of model checking for our work lies in the potential of applying (adaptations of) these search techniques to the intractable cases identified by Theorems 2 and 3. We have already performed an initial experiment in this direction; we will get back to this in the outlook, Sect. 9.

8.4 Beyond control-flow

Verification of process models has been studied for quite a while, mostly from a control flow perspective. In this context, different notions of soundness have been proposed; for an overview see [78]. There is a growing body of contributions beyond pure control-flow verification. Those relate to semantic checks and data flow analysis.

The approach of [48, 49] checks a notion of *semantic correctness* that builds on annotations to tasks as being mutually exclusive or dependent. In the first case they cannot co-occur in a trace, in the second case they must appear in a certain order. For semantic correctness the process must comply with the annotations. This approach provides somewhat similar features as linear temporal logic [80]. This kind

[12] There are no ontology axioms; with no loops and no xor splits, clearly there cannot be any xor/loop conditions; the restriction on facts implies that there can't be any effect conflicts.

of annotations can be simulated using a subset of our framework (using only preconditions/effects, with an empty ontology). In that sense, [48, 49] can be viewed as a special case of our framework.

In the area of *access control* the approach of [9] extends process models with predicates, constants, and variables. The meaning of these constructs relates to constraints on role assignments, while in our model they directly affect the executability of tasks. The work of [56] describes methods to *check compliance* of a process against rules for role assignment. This is related to our approach in that an ontology could (to some extent) be defined to model such rules; but not vice versa since we build on general logic while [56] covers some practical special cases. The paper by [67] addresses amongst others *life cycle compliance*. This can be partly reformulated in terms of preconditions, effects, and ontological axioms. Our running example illustrates some constraints related to the life cycle of business objects, i.e., when certain actions can only be performed if the business object is in the required state.

In [54], the preconditions and effects of service compositions are calculated on the basis of atomic services of which the compositions consist. Similar to our approach, the preconditions and effects of the atomic services are formulas, and the processes are assumed to be sound, have a single start and end node, respectively, and the routing constructs are and/xor join/split. However, Meyer [54] does not deal with loops and neither with ontological axiomatizations. There is no formal discussion of the algorithms or their properties. In particular, there is no proof of correctness and no consideration of complexity. The algorithm is based on computing the reachability graph of the composition's workflow, which is exponential in size of the workflow. This is in contrast to our investigation of polynomial time algorithms.

In [43], based on annotations of task nodes with logical effects, the authors use a propagation algorithm somewhat reminiscent of our I-propagation. There are, however, a number of important differences between the two approaches. Koliadis and Ghose [43] allow CNF effects which are considerably more expressive than our purely conjunctive effects. On the other hand, their propagation algorithm is exponential in the size of the process (the size of the propagated constructs multiplies at every XOR join). Further, Koliadis and Ghose [43] circumvent the consideration of loops, by assuming that entire sub-processes are annotated with effects. That is, sub-processes are handled as if they were atomic task nodes. This makes the analysis simpler, but, obviously, seriously impedes the ability to model sub-processes at a fine granular level. Koliadis and Ghose [43] do not consider preconditions, and they do not consider ontology axioms constraining the domain behavior. Finally, [43] do not provide a formal semantics for their effect annotations, and consequently, in difference to us, do not prove any formal correctness properties for their algorithms.

Another related line of work is data flow analysis, where dependencies are examined between the points where data is generated, and where it is consumed; some ideas related to this are implemented in the ADEPT system [63]. Data flow analysis builds on compiler theory [2] where data flows are typically examined for sequential programs mostly; it does neither consider theories $\mathcal{T}$ nor logical conflicts, and hence explores a direction complementary to ours. To some extent, our concepts can be applied in this area by expressing data dependencies as preconditions, effects, and ontological axioms.

ADEPT also provides a major contribution on the problem of process schema evolution [65]. In the latter area, also techniques from AI planning have already been combined with workflow concepts [6, 20, 40]. These combinations, however, have a very different purpose—and hence take a very different form—than the combination of workflows with AI that we develop herein. Whereas we aim at verification and hence define a formal state-based execution semantics, the previous approaches aimed at automatically creating new/adapted processes based on plan-like representations of the dependencies between process steps. Apart from that, a distinguishing feature of our approach is the handling of axiomatizations. Recall (cf. the discussion below Definition 4) that this impacts the formalization quite substantially, necessitating to deal with actions and change in the presence of state axioms. Finally, none of these previous works investigated the borderline between tractable and intractable cases.

9 Conclusion and discussion of open questions

Reducing the time span for design and deployment of process models is a very relevant problem [22, 70]. Towards this end, we devise a verification method exploiting semantic annotations. Our formalism is unique in the way it combines notions from the workflow community and from the AI literature, providing an execution semantics that integrates control-flow with logical preconditions and effects relative to an ontology. Based on the formalism, one can detect execution problems in process graphs with sound control flow, hence enabling verification beyond soundness.

We have investigated the tractability borderline of such verification, which is important because response time is a critical factor in practice. For precondition/effect conflicts, we have shown that the borderline is the same as that of reasoning in the logic underlying the ontology axioms. We have determined the class of basic processes, where, presuming effect conflicts have already been removed, executability of the overall process can be checked in polynomial time. The latter is not the case for any of the most relevant extensions of basic processes, so the class is maximal in that sense. Our algorithms are implemented within two BPM modeling environment prototypes, which show that user-friendly interfaces can be designed, and that ontologies can (sometimes) be obtained cheaply, by leveraging existing models.

One line of current research regards the enhancement of the SAP model underlying our SAP NetWeaver application, making explicit the dependencies between status variables. The most important open issue, in our view, is that of customer evaluation. As we stated in Sect. 7, such evaluation of our SAP prototypes is on the agenda, but at the time of writing it is not foreseeable when the evaluation will actually take place. Apart from that, several technical points are left open by our current results.

One question is whether executability of basic processes can be verified efficiently also in the presence of effect conflicts. If so, then the debugging facility provided becomes more flexible, allowing to check executability without prior removal of effect conflicts, and allowing to tolerate effect conflicts (to not view them as bugs) in case such behavior is intended. Further, such a solution would enable us to deal with precondition/effect conflicts based on execution parallelism (cf. Sect. 4) rather than

token-parallelism. This is because, for executable basic processes, the two notions of parallelism are equivalent. Hence the modeler could establish executability first, and thereafter use the algorithms from Sect. 4 to tackle precondition/effect conflicts based on execution parallelism.

An important open line of research regards the intractable cases we identified. Computational hardness is certainly a challenge for verification in an online setting, but not necessarily a deal-breaker. If the process models are not too large, and if the verification makes use of advanced search techniques (like symbolic representations, constraint propagation, search space reduction, or process decomposition), then the response times may be tolerable. We have made some initial experiments encoding processes with empty ontology (no axioms) for the explicit-state model checker SPIN [35]. The results are not encouraging, taking excessive runtime and/or memory even in fairly small processes. However, certainly that is not the end of the story. The performance of SPIN can possibly be improved by using different encodings, or some algorithmic extensions to SPIN (e.g. [26]). Also, one option we deem particularly promising is the use of SAT solvers (e.g. [27, 55]) for the verification, in the style of the bounded model checking approach [19]. In some cases, a single SAT call suffices for testing whether a particular node is executable. For example, we have already shown that this the case for basic processes without loops.

Another topic is to enhance the debugging information returned by the verification. The techniques presented herein provide only minimal information, namely the process nodes that are directly involved in a failure. It would be more desirable to return some approximation of what may *cause* the failure, and perhaps to make suggestions for bug fixes. We have made some initial steps in that direction, based on the local information obtained by I-propagation, i.e., the information pertaining to particular edges in the process. Using this information, unsatisfied preconditions can be traced back to activities that contribute to validating/invalidating them [34].

Last but not least, a long term research topic remains to investigate richer semantic annotations. In particular, an investigation of Description Logic [4], variants of which are widely used in the semantic Web community (e.g. [1, 21, 66, 72]), would be desirable.

Acknowledgements We would like to thank the anonymous reviewers for their excellent feedback and insights. Further, we want to acknowledge the contributions of Ulrich Benz during the early beginnings of this work. Part of this work has been supported by the EU Integrated Project SUPER and the Australian SmartServices CRC.

Appendix: Proofs

A.1 Control-flow properties

If two edges are parallel, then, in particular situations, we can "choose" which one to activate last:

Lemma 2 *Let $\mathcal{P} = (N, E, \lambda)$ be a sound process graph. Let $e \neq e' \in \mathcal{E}$ so that $e \parallel e'$, and $e \in out(n)$ where $n \notin \mathcal{N}_{PS}$. Let t' be a token-reachable token marking where*

$t'(e) > 0$ *and* $t'(e') > 0$. *Then there exists a token-reachable token marking* t *so that* $t \xrightarrow{n} t'$.

Proof Since $e \parallel e'$, we know that t' as claimed exists. Say t' is reached on the execution path $\vec{p} = \langle t_0 \xrightarrow{n_0} t_1 \xrightarrow{n_1} \cdots \xrightarrow{n_{k-1}} t_k \rangle$ where t_0 is the start marking and $t_k = t'$. By prerequisite we have $t_k(e) > 0$. Define i to be the highest index of a marking that activates e, i.e., $n_{i-1} = n$ and consequently $t_i(e) > 0$, such that e remains activated until t_k. That is, we have: (*) *for all* $i \leq j \leq k : t_j(e) > 0$.

Now, consider the token markings t_{i-1} and t_i, as well as the nodes $n_{i-1} = n$ and n_i. We know that n is executable in t_{i-1}, and that n_i is executable in t_i. We prove that we can re-order n and n_i in $\vec{p}$, and still obtain a valid execution path. Once this is proved, we are done: iterating the argument, we can move n upwards in $\vec{p}$ and, ultimately, execute it last.

Consider the re-ordered sequence $\langle t_0 \xrightarrow{n_0} \cdots \xrightarrow{n_{i-2}} t_{i-1} \xrightarrow{n_i} t_i' \xrightarrow{n_{i-1}} t_{i+1}' \rangle$. It suffices to show that:

1. n_i is token-executable in t_{i-1},
2. $n_{i-1} = n$ is token-executable in t_i', and
3. $t_{i+1}' = t_{i+1}$.

As for 1., we know that n_i is token-executable in t_i, which differs from t_{i-1} only in that n is executed beforehand. Observe that executing n puts a token only on e. This is obvious for non-split nodes, which have only a single outgoing edge. For xor splits it is clear because otherwise we would not have $t_i(e) > 0$, in contradiction to (*). This covers all cases because by prerequisite n is not a parallel split. Now, n_i cannot have e as an incoming edge: tokens from incoming edges are always removed by definition, except for xor joins. But incoming edges of xor joins cannot be parallel: the process is assumed to be sound so that would be a contradiction to Proposition 1. Thus, if n_i had e as an incoming edge, then we would not have $t_{i+1}(e) > 0$, in contradiction to (*). Hence n_i must be executable in t_{i-1} already, as desired.

As for 2., we know that n is token-executable in t_{i-1}. This differs from t_i' only in that n_i is not executed beforehand. Now, token-executability of n (of any node) depends of course only on the activation of n's incoming edges, and execution of n_i (of any node) removes tokens only from its incoming edges. Hence, if n is not executable in t_i', then we can derive that $in(n) \cap in(n_i) \neq \emptyset$, which is of course not possible since every edge has exactly one target node.

As for 3., we have already seen that n_i does not consume any tokens set by n. Likewise, it is obvious that n does not consume any tokens set by n_i, or else n could not be executed prior to n_i in $\vec{p}$. Hence the effects of the nodes on the token structure are mutually independent, from which the claimed property follows. This concludes the argument. □

In the next lemma, and at some points further below, we will make use of *flow orderings*. Given a process graph, a flow ordering is a bijective function $\# : E \mapsto \{0, \ldots, |E| - 1\}$, numbering the edges such that (i) every incoming edge of a node has a lower number than any of the node's outgoing edges, and (ii) all outgoing edges of a split node are consecutively enumerated. Flow orderings obviously exist, since

(N, E) is acyclic. For example, a flow ordering results from a breadth-first traversal of the process graph. We assume in the rest of the paper that, for every process graph under consideration, a flow ordering # is fixed. This is just a simple device to be able to more conveniently state certain proof arguments. We use the following helper notations. $\#^{-1}$ is the inverse function of #, i.e., $\#^{-1}(i) = e$ iff $\#(e) = i$. If E is a set of edges, then $\#E_{\max} := \max\{\#(e) \mid e \in E\}$ is the maximum number of any edge in E, and analogously for $\#E_{\min}$. For example, given a node n, $\#in(n)_{\max} = \max\{\#(e) \mid e \in in(n)\}$ is the maximum number of any incoming edge.

Lemma 3 *Let $\mathcal{P} = (N, E, \lambda)$ be a sound process graph, and let $\mathcal{Q} \in Sub(\mathcal{P})$. Let $e \neq e' \in E^{\mathcal{Q}}$ so that $e \parallel e'$, $e \in out(n)$ where $n \in N^{\mathcal{Q}}_{PS}$, and $\#^{\mathcal{Q}}(e') < \#out(n)_{\min}$. Let t' be a token-reachable token marking where $t'(e) > 0$ and $t'(e') > 0$. Then there exist token-reachable token markings t, t'' so that $t \xrightarrow{n} t''$ where $t''(e) > 0$ and $t''(e') > 0$.*

Proof Since $e \parallel e'$, we know that t' as claimed exists. Say t' is reached on the execution path $\vec{p} = \langle t_0 \xrightarrow{n_0} t_1 \xrightarrow{n_1} \cdots \xrightarrow{n_{k-1}} t_k \rangle$ as in the proof to Lemma 2. Since, obviously, parts of the path outside the sub-process $\mathcal{Q}$ in question do not matter, and $\vec{p}$ ends within $\mathcal{Q}$, in the following we will ignore the part of $\vec{p}$ outside $\mathcal{Q}$, i.e., we act as if the path was completely contained within $\mathcal{Q}$.

Virtually all arguments in the proof to Lemma 2 remain intact, with a single exception, namely the proof that n_i is token-executable in t_{i-1}. Precisely, the only part of the proof of Lemma 2 that makes use of the prerequisite $n \notin \mathcal{N}_{PS}$, is the argument given here to show that execution of n does not put a token on any edge $e'' \in in(n_i)$. We need to find a different argument for this.

Such an argument can be based on the new prerequisite of our claim here, namely that $\#^{\mathcal{Q}}(e') < \#out(n)_{\min}$. What we prove is that: (*) *a suitable $\vec{p'}$ can be constructed so that, for all nodes n_j in $\vec{p'}$, $\#in(n_j)_{\max} < \#out(n)_{\min}$*. This immediately proves the claim: if $e'' \in out(n) \cap in(n_i)$, then, by construction of #, it follows that $\#^{\mathcal{Q}}(e'') \geq \#out(n)_{\min}$ in contradiction to (*).

Say $n_j \neq n$ is the node in $\vec{p}$ with maximal $\#in(n_j)_{\max}$. If $\#in(n_j)_{\max} < \#out(n)_{\min}$, there is nothing to prove. Else, construct $\vec{p'}$ by removing n_j, i.e., set $\vec{p'} := \langle t_0 \xrightarrow{n_0} \cdots \xrightarrow{n_{j-1}} t_j \xrightarrow{n_{j+1}} t'_{j+2} \xrightarrow{n_{j+2}} \cdots \xrightarrow{n_{k-1}} t'_k \rangle$.

First, observe that $\vec{p'}$ is still a valid execution path, i.e., for $j + 2 \leq l \leq k - 1$, we have that n_l is executable in t'_l. If that were not the case, then obviously there would exist an l so that $in(n_l) \cap out(n_j) \neq \emptyset$, i.e., n_j produces a token needed by n_l. However, by construction of #, $\#in(n_j)_{\max} < \#out(n_j)_{\min}$. So, if $e'' \in in(n_l) \cap out(n_j)$, then $\#in(n_l)_{\max} \geq \#^{\mathcal{Q}}(e') > \#in(n_j)_{\max}$ which is a contradiction since $\#in(n_j)_{\max}$ is assumed to be maximal.

Second, observe that $t'_k(e) > 0$ and $t'_k(e') > 0$. Obviously, the only chance for that not to happen is if e, respectively e', is contained in $out(n_j)$. So $t'_k(e) > 0$ is obvious since $n_j \neq n$. As for $t'_k(e') > 0$, assume that $e' \in out(n_j)$. Then, by construction of #, we have $\#^{\mathcal{Q}}(e') > \#in(n_j)_{\max}$. Further, by assumption we have $\#in(n_j)_{\max} \geq \#out(n)_{\min}$. Finally, by prerequisite we have $\#out(n)_{\min} > \#^{\mathcal{Q}}(e')$. We can conclude that $\#^{\mathcal{Q}}(e') > \#^{\mathcal{Q}}(e')$ which is of course a contradiction.

Iterating the argument, we can remove from $\vec{p}$ all nodes where $\#in(n_j)_{\max} \geq \#out(n)_{\min}$, and still obtain an execution path at whose end both e and e' are active. This proves (*) and hence concludes the argument. □

A.2 Binary theories can be compiled away

We sometimes need to distinguish the state spaces of different processes. We then indicate the process as a superscript: $\mathcal{SI}^{\mathcal{Q}}(e)$ denotes the state intersection relative to $\mathcal{Q}$, i.e., the set of literals that are always true when e is activated in an execution path of $\mathcal{Q}$.

Lemma 4 *Let $\mathcal{Q} = (N, E, \lambda, \Omega, \alpha)$, $\Omega = (P, \mathcal{T})$, be an executable basic annotated process graph. Denote by C the set of all constants appearing in any of the annotated* $\mathrm{pre}(n), \mathrm{eff}(n)$. *Let $\mathcal{Q}' = (N, E, \lambda, \Omega', \alpha')$ be the modification of $\mathcal{Q}$ where $\Omega' = (P, 1)$ and $\alpha' \equiv \alpha$ except that, for all $n \in \mathcal{N}_T$, $\mathrm{eff}'(n) := \{l \in P[C] \mid \mathcal{T}[C] \wedge \mathrm{eff}(n) \models l\}$ if $\mathrm{eff}(n)$ is defined, and $\mathrm{eff}'(n) := \{l \in P[C] \mid \mathcal{T}[C] \models l\}$ otherwise. Then, for every $e \in \mathcal{E}$, we have: $\mathcal{SI}^{\mathcal{Q}}(e) = \mathcal{SI}^{\mathcal{Q}'}(e)$.*

Proof In what follows, we denote a state by the set of literals it makes true. We first prove the following: given a reachable state s with a token on $in(n)$ for a task node n, in $\mathcal{Q}$ exactly one state s' can be reached by executing n in s, namely the state $s' := (s \setminus \neg\mathrm{eff}'(n)) \cup \mathrm{eff}'(n)$.

Recall that, by definition, the states s' reachable by executing n in s are all those where $s' \in \text{PMA-min}(s, \mathcal{T}[C] \wedge \mathrm{eff}(n))$, which is defined to be the set of all states that satisfy $\mathcal{T}[C] \wedge \mathrm{eff}(n)$ and that differ in a set-inclusion minimal set of values from s.

First, for any $s' \in \text{PMA-min}(s, \mathcal{T}[C] \wedge \mathrm{eff}(n))$ it is clear by definition that $\mathrm{eff}'(n) \subseteq s'$. The definition of s' as given above changes *only* those values. It suffices to show that $s' \models \mathcal{T}[C]$: then, we have $s' \models \mathcal{T}[C] \wedge \mathrm{eff}(n)$, and clearly the set of changed values is a proper subset of any other state with the same property. Assume to the contrary of the claim that $(l \vee l') \in \mathcal{T}[C]$ and $s' \not\models l \vee l'$, i.e., $\neg l \in s'$ and $\neg l' \in s'$; note here that $\mathcal{T}$ is binary and hence every clause has at most two literals. If $\neg l \in \mathrm{eff}'(n)$, then $l' \in \mathrm{eff}'(n)$—because, given the clause $l \vee l'$, l' is a logical consequence of $\neg l$. With $\mathrm{eff}'(n) \subseteq s'$ we obtain a contradiction, proving that $\neg l$ cannot be contained in $\mathrm{eff}'(n)$. Similarly, we can disprove $\neg l' \in \mathrm{eff}'(n)$. Hence, by construction of s', $\{\neg l, \neg l'\} \subseteq s$. But then, $s \not\models \mathcal{T}[C]$ which is a contradiction because s is reachable.

With the above, we know that, for any reachable state s and any task node n, the (single) transition induced in $\mathcal{Q}$ is exactly the same as the transition induced in $\mathcal{Q}'$. Hence, obviously since the graph structure is not changed in any other way, any possible difference in the sets $\mathcal{SI}(e)$ would have to be due to different start states. So let us consider the start states in $\mathcal{Q}$ and $\mathcal{Q}'$.

The start states in $\mathcal{Q}$ are all those with $s_0 \models \mathcal{T}[C]$, and $s_0 \models \mathcal{T}[C] \wedge \mathrm{eff}(n_0)$ in case $\alpha(n_0)$ is defined. In $\mathcal{Q}'$, by construction the start states are all those where $s_0 \models 1 \wedge \mathrm{eff}'(n_0)$, with $\mathrm{eff}'(n_0) = \{l \in P[C] \mid \mathcal{T}[C] \models l\}$ in case $\alpha(n_0)$ is undefined, and $\mathrm{eff}'(n_0) = \{l \in P[C] \mid \mathcal{T}[C] \wedge \mathrm{eff}(n_0) \models l\}$ in case $\alpha(n_0)$ is defined.

Obviously, this means that the set of start states of $\mathcal{Q}'$ is a superset of the set of start states of $\mathcal{Q}$—any start state of $\mathcal{Q}$ is a start state of $\mathcal{Q}'$, but not vice versa. However, likewise obviously, the set of literals true in *all* start states is the same in both cases, i.e., we have $\mathcal{SI}^{\mathcal{Q}}(e_0) = \mathcal{SI}^{\mathcal{Q}'}(e_0)$.

Let e be any edge in the graph. Consider, for the moment, only the workflow structure of the graphs, i.e., the token executions. Since $\mathcal{Q}'$ does not change the graph structure, the set of token execution paths leading from (a state with a token on) e_0 to (a state with a token on) e is the same in both $\mathcal{Q}$ and $\mathcal{Q}'$. Let's call this set of paths $\vec{P}$. By prerequisite, every task node is executable, there are no conditions at the outgoing edges of xor splits, and there are no conditions at loop nodes. Thus we know that every path $\vec{p} \in \vec{P}$ can be executed from every possible start state s_0, in both $\mathcal{Q}$ and $\mathcal{Q}'$. The change that $\vec{p}$ makes to s_0 is the accumulated effect of the task nodes executed on $\vec{P}$. From the above, we know that this is the same in both $\mathcal{Q}$ and $\mathcal{Q}'$. We can write the resulting state s as $s = (s_0 \setminus \neg\text{eff}(\vec{p})) \cup \text{eff}(\vec{p})$, where $\text{eff}(\vec{p})$ denotes the accumulated effect of $\vec{p}$—what exactly that latter effect is does not play a role in our argument below. The important point is that $\text{eff}(\vec{p})$ is a function, i.e., is well-defined.

Consider now the sets $\mathcal{SI}^{\mathcal{Q}}(e)$ and $\mathcal{SI}^{\mathcal{Q}'}(e)$. With the above, we know that

$$\mathcal{SI}^{\mathcal{Q}}(e) = \bigcap_{s_0, \vec{p}} ((s_0 \setminus \neg\text{eff}(\vec{p})) \cup \text{eff}(\vec{p})),$$

where s_0 ranges over the start states of $\mathcal{Q}$ and $\vec{p}$ ranges over $\vec{P}$. Now, first, we can separate the "positive effects"—which occur irrespectively of the start state—out and get

$$\mathcal{SI}^{\mathcal{Q}}(e) = \left(\bigcap_{s_0, \vec{p}} (s_0 \setminus \neg\text{eff}(\vec{p}))\right) \cup \left(\bigcap_{\vec{p}} \text{eff}(\vec{p})\right).$$

Further, we can re-write $\bigcap_{s_0, \vec{p}} (s_0 \setminus \neg\text{eff}(\vec{p}))$ to $\bigcap_{s_0, \vec{p}} (s_0 \cap L(\vec{p}))$ where $L(\vec{p})$ is the complement of $\neg\text{eff}(\vec{p})$. We can re-write $\bigcap_{s_0, \vec{p}} (s_0 \cap L(\vec{p}))$ to $(\bigcap_{s_0} s_0) \cap (\bigcap_{\vec{p}} L(\vec{p}))$. Hence, overall, we have derived that

$$\mathcal{SI}^{\mathcal{Q}}(e) = \left(\left(\bigcap_{s_0} s_0\right) \cap \left(\bigcap_{\vec{p}} L(\vec{p})\right)\right) \cup \left(\bigcap_{\vec{p}} \text{eff}(\vec{p})\right).$$

In the same way, we can derive

$$\mathcal{SI}^{\mathcal{Q}'}(e) = \left(\left(\bigcap_{s'_0} s'_0\right) \cap \left(\bigcap_{\vec{p}} L(\vec{p})\right)\right) \cup \left(\bigcap_{\vec{p}} \text{eff}(\vec{p})\right),$$

where s'_0 ranges over the start states of $\mathcal{Q}'$. We need to prove that $\mathcal{SI}^{\mathcal{Q}}(e) = \mathcal{SI}^{\mathcal{Q}'}(e)$. Replacing both sides of the equation with the expressions we have just derived, the terms concerning $\vec{p}$ occur on both sides and can be removed. Thus we find that our desired equality is equivalent to $\bigcap_{s_0} s_0 = \bigcap_{s'_0} s'_0$, which we have already proved above. This concludes the argument. □

A.3 Correctness of I-propagation

We define aggregate-eff($\mathcal{Q}$), the aggregated effect literals of a sub-graph $\mathcal{Q}$, as follows:

$$\text{aggregate-eff}(\mathcal{Q}) := \bigcup_{n \in N_T^{\mathcal{Q}}} \text{eff}(n) \cup \bigcup_{n \in N_L^{\mathcal{Q}}} \text{aggregate-eff}(\lambda^{\mathcal{Q}}(n)).$$

Lemma 5 *Let $\mathcal{Q} = (N, E, \lambda, \Omega, \alpha)$ be an executable basic sound annotated process graph without effect conflicts, and let $t \geq 0$. Let $E^0 \subseteq E$ be a set of edges so that there exists a state $s \in \mathcal{S}$ where, for all $e \in E^0$, $t_s(e) > 0$. Let l be a literal so that, for each $e \in E^0$, there exists a state $s' \in \mathcal{S}$ where $s' \not\models l$ and $t_{s'}(e) > 0$. Then, there exists a state $s_0 \in \mathcal{S}$ where $s_0 \not\models l$ and, for all $e \in E^0$, $t_{s_0}(e) > 0$.*

Proof Let l be an arbitrary literal, and let $t \geq 0$ be arbitrary. We prove that the claim holds for all possible E^0, by induction over the process structure, as reflected in the enumeration function #. As the induction base case, we prove that the claim holds for every set E^0 where $\#E^0_{\max} \leq 0$. As the inductive step, we prove that, for every node n, if the claim holds for every E^0 where $\#E^0_{\max} \leq \#out(n)_{\min} - 1$, then the claim holds for every E^0 where $\#E^0_{\max} \leq \#out(n)_{\max}$.

Base case. Since e_0 is not parallel to any other edge (no edge can carry a token at the same time as e_0 does), the only set E^0 containing e_0 is the singleton $\{e_0\}$, for which the claim holds trivially.

Inductive case. Let $n \in N$. As stated, the induction hypothesis is that the claim holds for every E^0 where $\#E^0_{\max} \leq \#out(n)_{\min} - 1$. We prove that, under this hypothesis, the claim holds for every E^0 where $\#E^0_{\max} \leq \#out(n)_{\max}$.

To avoid clumsiness of language, we will use the following conventions. Whenever we write "E^0", we mean a set of edges with $\#E^0_{\max} \leq \#out(n)_{\min} - 1$ for which the prerequisite of the claim holds: there exists a state $s \in \mathcal{S}$ where, for all $e \in E^0$, $t_s(e) > 0$; and, for each single $e \in E^0$, there exists a state $s' \in \mathcal{S}$ where $s' \not\models l$ and $t_{s'}(e) > 0$. Similarly, whenever we write "$E^{0'}$", we mean a set of edges with $\#E^{0'}_{\max} \leq \#out(n)_{\max}$ for which the prerequisite of the claim holds. Further, since the induction hypothesis covers all other cases, we assume that $E^{0'} \cap out(n) \neq \emptyset$. Finally, since the case of $E^{0'} \subseteq out(n)$ is trivial for all kinds of nodes n, we assume that $E^{0'} \not\subseteq out(n)$. We distinguish the different kinds of nodes n:

1. $n \in N_T$. We distinguish three cases:
   (a) $l \in \overline{\text{eff}(n)}$. This case is trivial because no $E^{0'}$ exists. Assume the opposite was the case. Then there exists a state $s' \in \mathcal{S}$ where $t_{s'}(out(n)) > 0$ and $s' \not\models l$. Since, directly after executing n, l is true, this means that a task node parallel to n has made l false. Hence we have an effect conflict, in contradiction to the prerequisite.
   (b) $\neg l \in \overline{\text{eff}(n)}$. Let $E^{0'}$ be an arbitrary set of edges, with $out(n) \in E^{0'}$ and so that there exists a state $s \in \mathcal{S}$ where $t_s(e) > 0$ for every $e \in E^{0'}$. In order to reach s, n must be executed. Since n is not a parallel split, we can apply Lemma 2 to any pair of $out(n)$ and $out(n) \neq e \in E^{0'}$. Hence there exists an execution path to s on which n comes *last*. By prerequisite, n is executable, and so we

can execute it at this point. Obviously, and $s_0 \not\models l$. Hence the claim holds for $E^{0'}$, and we are done.

(c) $\{l, \neg l\} \cap \overline{\text{eff}}(n) = \emptyset$. For this case, we prove that there is a mapping from sets E^0 to sets $E^{0'}$. Precisely, we prove that we can construct each set $E^{0'}$ as $E^{0'} = E^0 \setminus \{in(n)\} \cup \{out(n)\}$ where E^0 is a set satisfying the prerequisite of the claim. Once this is proved, the claim follows easily: by induction hypothesis, we know that there exists a state $s_0 \in \mathcal{S}$ where $s_0 \not\models l$ and $t_{s_0}(e) > 0$ for all $e \in E^0$; in that state, we can execute n; the resulting state obviously satisfies the requirements of the claim.

It remains to prove the desired mapping. Let $E^{0'}$ be a set of edges with $E^{0'} \cap out(n) \neq \emptyset$ so that: there exists a state $s \in \mathcal{S}$ where, for all $e \in E^{0'}$, $t_s(e) > 0$; and, for each single $e \in E^{0'}$, there exists a state $s' \in \mathcal{S}$ where $s' \not\models l$ and $t_{s'}(e) > 0$. We need to prove that $E^0 := E^{0'} \setminus \{out(n)\} \cup \{in(n)\}$ has the same properties. The existence of the desired state $s \in \mathcal{S}$ follows by application of Lemma 2 to $E^{0'}$ and s: we get a path to s on which n is applied last; the predecessor state activates all edges in E^0 and hence serves as the desired state s for E^0. Regarding the existence of the state $s' \in \mathcal{S}$ with $s' \not\models l$ and $t_{s'}(out(n)) > 0$, there are two possible reasons for that. First, there exists a state $s'' \in \mathcal{S}$ with $s'' \not\models l$ and $t_{s'}(in(n)) > 0$; in that case there is nothing to prove. Second, there exists a task node n' parallel to n that falsifies l in its effect. But then, n' can be executed directly before n, and hence we are back in the first case, i.e., we can construct a state $s'' \in \mathcal{S}$ as appropriate.

2. $n \in N_L$. We distinguish three cases similar as for task nodes; the respective proofs are similar as well:
   (a) On every path through $\lambda(n)$, the last change to l makes l true; in particular, $l \in \text{aggregate-eff}(\lambda(n))$. This case is trivial because no $E^{0'}$ exists. Assume the opposite was the case. Then there exists a state $s' \in \mathcal{S}$ where $t_{s'}(out(n)) > 0$ and $s' \not\models l$. Since, directly after executing $\lambda(n)$, l is true, this means that a task node parallel to n has made l false. Hence we have an effect conflict, in contradiction to the prerequisite.
   (b) There exists a path $\vec{p}$ through $\lambda(n)$ where the last change makes l false; in particular, $\neg l \in \text{aggregate-eff}(\lambda(n))$. Let $E^{0'}$ be an arbitrary set of edges, with $out(n) \in E^{0'}$ and so that there exists a state $s \in \mathcal{S}$ where $t_s(e) > 0$ for every $e \in E^{0'}$. In order to reach s, n must be executed. Since n is not a parallel split, we can apply Lemma 2 to any pair of $out(n)$ and $out(n) \neq e \in E^{0'}$. Hence there exists an execution path to s on which n comes *last*. By prerequisite, $\lambda(n)$ is executable, and so we can execute $\vec{p}$ at this point. Obviously, the resulting state s_0 has $s_0 \not\models l$. Hence the claim holds for $E^{0'}$, and we are done.
   (c) $\{l, \neg l\} \cap \text{aggregate-eff}(\lambda(n)) = \emptyset$. This case is proved exactly as for task nodes. The only difference is that, rather than executing just n without affecting the value of l, we execute some path through $\lambda(n)$ without affecting the value of l. This does not affect the proof arguments.
3. $n \in N_{XS}$. There is a mapping from sets $E^{0'}$ to sets E^0. Namely, we can construct each $E^{0'}$ respectively as $E^{0'} = E^0 \setminus \{in(n)\} \cup \{e'\}$, where $e' \in out(n)$. This, like above, follows from Lemma 2 regarding parallelism, i.e., the existence of the state s in the prerequisite of the claim. Regarding the existence of the states s' in the

prerequisite of the claim, the argument is the same as before: a state s' which falsifies l and activates one of the outgoing edges can always be constructed from a state which falsifies l and activates the incoming edge.

By induction hypothesis we know that there exists a reachable state $s_0 \in \mathcal{S}$ where $s_0 \not\models l$ and, for all $e \in E^0$, $t_{s_0}(e) > 0$. In that state, we can execute n. Because, by prerequisite, the process graph is basic, in particular no conditions are annotated at the outgoing edges of any xor split. Hence, regardless of how s_0 interpretes the logical propositions, we can choose to execute n in a way so that a token is put on e'. The resulting state obviously satisfies the requirements of the claim.

4. $n \in N_{XJ}$. Like for xor splits, we have a mapping from sets $E^{0'}$ to sets E^0: every set $E^{0'}$ can be constructed from a set E^0 as $E^{0'} = E^0 \setminus \{e\} \cup \{out(n)\}$, where $e \in in(n)$. The proof for that is as before, and the claim follows as before.
5. $n \in N_{PJ}$. This case is also handled analogously: every set $E^{0'}$ can be constructed from a set E^0 as $E^{0'} = E^0 \setminus in(n) \cup \{out(n)\}$. That correspondence is proved as before, and the claim follows as before.
6. $n \in N_{PS}$. In this case, every set $E^{0'}$ can be constructed from a set E^0 as $E^{0'} = E^0 \setminus \{in(n)\} \cup E'$, where $E' \subseteq out(n)$; we argue this mapping below. With this mapping, the proof proceeds as before. By induction hypothesis we know that there exists a reachable state $s_0 \in \mathcal{S}$ where $s_0 \not\models l$ and, for all $e \in E^0$, $t_{s_0}(e) > 0$. In that state, we can execute n and put a token on every edge in E'. The resulting state obviously satisfies the requirements of the claim.

   It remains to prove that every set $E^{0'}$ can be constructed from a set E^0 as $E^{0'} = E^0 \setminus \{in(n)\} \cup E'$, where $E' \subseteq out(n)$. Let $E^{0'}$ be any set of edges with $E^{0'} \cap out(n) \neq \emptyset$ and $\#E^{0'}_{\max} \leq \#out(n)_{\max}$ so that: there exists a state $s \in \mathcal{S}$ where, for all $e \in E^{0'}$, $t_s(e) > 0$; and, for each single $e \in E^{0'}$, there exists a state $s' \in \mathcal{S}$ where $s' \not\models l$ and $t_{s'}(e) > 0$. We prove that $E^0 := E^{0'} \setminus out(n) \cup in(n)$ has the same properties. The existence of the state $s \in \mathcal{S}$ follows by application of Lemma 3 to $\mathcal{Q}$, $E^{0'}$, and s: we get a path on which n is applied last and whose end state activates all edges in $E^{0'}$; the predecessor state activates all edges in E^0. Regarding the existence of the state $s' \in \mathcal{S}$ with $s' \not\models l$ which activates the edges in E', there are two possible reasons for that. First, there exists a state $s'' \in \mathcal{S}$ with $s'' \not\models l$ and $t_{s'}(in(n)) > 0$; in that case there is nothing to prove. Second, there exists a task node n' parallel to n that falsifies l in its effect. But then, n' can be executed directly before n, and hence we are back in the first case, i.e., we can construct a state s'' as appropriate. This concludes the argument. □

Lemma 6 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an executable basic annotated process graph. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $e \in \mathcal{E}$, $\mathcal{SI}(e) \supseteq I^*(e)$.*

Proof Since $\mathcal{P}$ is executable and basic, we can apply Lemma 4. That is, we can compile the binary ontology into extended action effects without affecting the sets $\mathcal{SI}(e)$. Hence in what follows we can assume without loss of generality that the ontology is empty.

Let $e \in \mathcal{E}$ and let $l \in P[C]$, where C are the constants used by α. Assume that there exists an execution path $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} s_2 \cdots s_{k-1} \xrightarrow{n_k} s_k = s$ so that $t_s(e) > 0$ and $i_s \not\models l$.

We show that, then, there exists an I-propagation path $I_0 \xrightarrow{n'_1} I'_1 \xrightarrow{n'_2} I'_2 \cdots I'_{l-1} \xrightarrow{n'_l} I_l = I'$ so that $l \notin I(e)$.

Note first that, given a function $I : E \mapsto 2^{P[C]}$ and a node n, there exists at most one I' so that I' is the propagation of I at n, i.e., I' is completely determined; I' exists iff propagating I at n results in any changes.

Consider now the sequence of nodes $n_1, \ldots, n_k$. We construct a sequence $I_0, I_1, \ldots, I_k$ as follows. For all $0 \leq j < k$, if propagating I_j at n_j results in changes, then set I_{j+1} to the outcome of that propagation; else, set $I_{j+1} := I_j$. Obviously, we get an I-propagation path $I_0 \xrightarrow{n'_1} I'_1 \xrightarrow{n'_2} I'_2 \cdots I'_{l-1} \xrightarrow{n'_l} I'_l = I$ by removing from $I_0, I_1, \ldots, I_k$ those steps where no changes occur. We then have $I_k = I$, and hence it suffices to prove that $l \notin I_k(e)$.

In what follows, we denote $t_j := t_{s_j}$ and $i_j := i_{s_j}$. We prove by induction that, for all $0 \leq j \leq k$: for all e where $t_j(e) > 0$, we have $i_j \models I_j(e)$.

Base case, $j = 0$. The only e' with $t_j(e') > 0$ is the start edge e_0. Since $I_0(e_0) = \text{eff}(n_0)$, the claim follows.

Inductive case, $j \to j + 1$. We distinguish the different kinds of executions of the node $n := n_{j+1}$:

1. $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{XS}$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $t_j(e') > 0$, or (b) $e' \in out(n)$. In case (a), the claim follows immediately from the induction hypothesis because $i_{j+1} = i_j$ and $I_{j+1}(e') = I_j(e')$. As for case (b), since n can be executed in s_j, we have that $t_j(in(n)) > 0$, and hence by induction assumption we know that $i_j \models I_j(in(n))$. The claim then follows because $i_{j+1} = i_j$ and, for all $e' \in out(n)$, $I_{j+1}(e') \subseteq I_j(in(n))$.
2. $n \in \mathcal{N}_T$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $t_j(e') > 0$, or (b) $e' = out(n)$. In case (a), the claim follows immediately from the induction hypothesis because $i_{j+1} = \text{eff}(n) \cup (i_j \setminus \neg\text{eff}(n))$, writing an interpretation as the set of literals it satisfies; and $I_{j+1}(e') = I_j(e') \setminus \neg\text{eff}(n)$ because, with $t_j(e') > 0$ and $t_j(in(n)) > 0$, we have $e' \parallel in(n)$. As for case (b), since $t_j(in(n)) > 0$ by induction assumption we know that $i_j \models I_j(in(n))$. The claim then follows because $i_{j+1} = \text{eff}(n) \cup (i_j \setminus \neg\text{eff}(n))$ and $I_{j+1}(out(n)) \subseteq \text{eff}(n) \cup (I_j(in(n)) \setminus \neg\text{eff}(n))$.
3. $n \in \mathcal{N}_{PJ}$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $e' \neq out(n)$ and $t_j(e') > 0$, or (b) $e' = out(n)$. In case (a), $I_{j+1}(e') = I_j(e')$ and hence the claim follows from the induction hypothesis and $i_{j+1} = i_j$. As for case (b), since n can be executed in s_j, we have that $t_j(e') > 0$ for all $e' \in in(n)$. Hence by induction assumption we know that $i_j \models I_j(e')$ for all $e' \in in(n)$. The claim then follows because $i_{j+1} = i_j$ and $I_{j+1}(out(n)) \subseteq \bigcup_{e' \in in(n)} I_j(e')$.
4. $n \in \mathcal{N}_{XJ}$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $e' \neq out(n)$ and $t_j(e') > 0$, or (b) $e' = out(n)$. In case (a), $I_{j+1}(e') = I_j(e')$ and hence the claim follows from the induction hypothesis and $i_{j+1} = i_j$. As for case (b), since n can be executed in s_j, we have that $t_j(e') > 0$ for at least one $e' \in in(n)$. By induction assumption we know that $i_j \models I_j(e')$ for that e'. The claim then follows because $i_{j+1} = i_j$ and $I_{j+1}(out(n)) \subseteq \bigcap_{e' \in in(n)} I_j(e')$.
5. $n \in N_L^{\mathcal{Q}}$ with $\lambda^{\mathcal{Q}}(n) = \mathcal{Q}'$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $t_j(e') > 0$, or (b) $e' = e_0^{\mathcal{Q}'}$. In case (a), the claim follows immediately

from the induction hypothesis because $i_{j+1} = i_j$ and $I_{j+1}(e') = I_j(e')$. As for case (b), since n can be executed in s_j, we have that $t_j(in(n)) > 0$, and hence by induction assumption we know that $i_j \models I_j(in(n))$. The claim then follows because $i_{j+1} = i_j$ and $I_{j+1}(e_0^{\mathcal{Q}'}) \subseteq I_j(in(n))$.

6. $n = n_+^{\mathcal{Q}}$ and $t_{j+1}(e_0^{\mathcal{Q}}) > t_j(e_0^{\mathcal{Q}})$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $t_j(e') > 0$, or (b) $e' = e_0^{\mathcal{Q}}$. In case (a), the claim follows immediately from the induction hypothesis because $i_{j+1} = i_j$ and $I_{j+1}(e') = I_j(e')$. As for case (b), since n can be executed in s_j, we have that $t_j(in(n)) > 0$, and hence by induction assumption we know that $i_j \models I_j(in(n))$. The claim then follows because $i_{j+1} = i_j$ and $I_{j+1}(e_0^{\mathcal{Q}}) \subseteq I_j(in(n))$.
7. $n = n_+^{\mathcal{Q}}$ with $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$ and $t_{j+1}(out(n')) > t_j(out(n'))$. Consider the edges e' where $t_{j+1}(e') > 0$. We either have (a) $t_j(e') > 0$, or (b) $e' = out(n')$. In case (a), the claim follows immediately from the induction hypothesis because $i_{j+1} = i_j$ and $I_{j+1}(e') = I_j(e')$. As for case (b), since n can be executed in s_j, we have that $t_j(in(n)) > 0$, and hence by induction assumption we know that $i_j \models I_j(in(n))$. The claim then follows because $i_{j+1} = i_j$ and $I_{j+1}(out(n')) \subseteq I_j(in(n))$. □

Lemma 7 *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an executable basic sound annotated process graph without effect conflicts. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $e \in \mathcal{E}$, $\mathcal{SI}(e) \subseteq I^*(e)$.*

Proof Since $\mathcal{P}$ is executable and basic, we can apply Lemma 4. That is, we can compile the binary ontology into extended action effects without affecting the sets $\mathcal{SI}(e)$. Hence in what follows we can assume without loss of generality that the ontology is empty.

Assume an I-propagation path $I_0 \xrightarrow{n_1} I_1 \xrightarrow{n_2} I_2 \cdots I_{k-1} \xrightarrow{n_k} I_k$. We prove the following. For every $0 \leq j \leq k$ and for every edge $e \in \mathcal{E}$ and literal $l \in P[C]$ where $l \notin I_j(e)$, there exists an execution path ending in a state s so that $t_s(e) > 0$ and $s \not\models l$. The proof is by induction over j.

Base case, $j = 0$. By definition, $I_0(e) = P[C]$ except for $e = e_0$, where $I_0(e_0) = \text{eff}(n_0)$. Hence the only pairs e, l with $l \notin I_0(e)$ are those where $e = e_0$ and $l \notin \text{eff}(n_0)$. Obviously, every start state s_0 has $t_{s_0}(e_0) > 0$. If $l \notin \text{eff}(n_0)$ then by definition at least one start state s_0 exists where $s_0 \not\models l$. This shows the claim.

Inductive case, $j \to j + 1$. By induction hypothesis, we know that, for every edge $e \in \mathcal{E}$ and literal $l \in P[C]$ where $l \notin I_j(e)$, there exists an execution path ending in a state s so that $t_s(e) > 0$ and $s \not\models l$. We distinguish the different kinds of nodes $n := n_{j+1}$:

1. $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{XS}$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over split nodes, we either have (a) $l \notin I_j(e)$ or (b) $e \in out(n)$ and $l \notin I_j(in(n))$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. As for case (b), by induction hypothesis there exists an execution path ending in a state s so that $t_s(in(n)) > 0$ and $s \not\models l$. We can execute n in s. Since there are no conditions at the outgoing edges of xor splits, if n is an xor split then we can choose to put a token on e; if n is a parallel split then tokens are put on all outgoing edges, in particular on e.

Hence we can construct an execution path ending in a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument.

2. $n \in \mathcal{N}_T$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over task nodes, we have one of the following cases: (a) $l \notin I_j(e)$; or (b) $e \parallel out(n)$ and $l \in \neg\text{eff}(n)$; or (c) $e = out(n)$ and $l \in \neg\text{eff}(n)$; or (d) $e = out(n)$ and $l \notin I_j(in(n))$. In case (a), the induction hypothesis proves the claim. In case (b), we construct some execution path that activates both e and $out(n)$, and which executes n last. A token execution path of $\mathcal{P}$ doing so exists by Lemma 2; any token execution of $\mathcal{P}$ corresponds directly to an execution path because by prerequisite there are no conditions at the outgoing edges of xor splits, there are no conditions at loop nodes, and all task nodes are executable. In case (c), we simply construct some execution path that executes n last; at least one such path exists because by prerequisite $\mathcal{P}$ is sound, all task nodes are executable, there are no conditions at the outgoing edges of xor splits, and there are no conditions at loop nodes. In case (d), finally, by induction hypothesis there exists an execution path ending in a state s so that $t_s(in(n)) > 0$ and $s \not\models l$. Since n is executable by prerequisite, we can execute n in s, getting to a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument.
3. $n \in \mathcal{N}_{PJ}$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over parallel join nodes, we either have (a) $l \notin I_j(e)$ or (b) $e = out(n)$ and for every $e_i \in in(n) : l \notin I_j(e_i)$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. As for case (b), by induction hypothesis for every $e_i \in in(n)$ there exists an execution path ending in a state s_i so that $t_{s_i}(e_i) > 0$ and $s_i \not\models l$. We can thus apply Lemma 5, and obtain an execution path to a state s with $s \not\models l$ and $t_s(e_i) > 0$ for *all* $e_i \in in(n)$. We can execute n in s, getting to a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument.
4. $n \in \mathcal{N}_{XJ}$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over xor join nodes, we either have (a) $l \notin I_j(e)$ or (b) $e = out(n)$ and for at least one $e' \in in(n) : l \notin I_j(e')$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. As for case (b), by induction hypothesis there exists an execution path ending in a state s so that $t_s(e') > 0$ and $s \not\models l$. We can execute n in s, getting to a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument.
5. $n \in N_L^{\mathcal{Q}}$ with $\lambda^{\mathcal{Q}}(n) = \mathcal{Q}'$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over loop nodes, we either have (a) $l \notin I_j(e)$ or (b) $e = e_0^{\mathcal{Q}'}$ and $l \notin I_j(in(n))$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. As for case (b), by induction hypothesis there exists an execution path ending in a state s so that $t_s(in(n)) > 0$ and $s \not\models l$. We can execute n in s, getting to a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument.
6. $n = n_+^{\mathcal{Q}}$ with $\mathcal{Q} = \lambda^{\mathcal{Q}'}(n')$. Say that $e \in \mathcal{E}$ and $l \in P[C]$ where $l \notin I_{j+1}(e)$. By the definition of I-propagation over end nodes, we have one of the following cases: (a) $l \notin I_j(e)$; or (b) $e = e_0^{\mathcal{Q}}$ and $l \notin I_j(in(n))$; or (c) $e = out(n')$ and $l \notin I_j(in(n))$. In case (a), the induction hypothesis shows the existence of an execution path as desired, so there is nothing to prove. As for cases (b) and (c), by induction

hypothesis there exists an execution path ending in a state s so that $t_s(in(n)) > 0$ and $s \not\models l$. Since n is executable by prerequisite, we can execute n in s. Since by prerequisite there are no conditions at loop nodes, we can choose to repeat the loop or exit the loop, i.e., we can put a token on $e_0^{\mathcal{Q}}$ or $out(n')$ as desired for case (b) respectively for case (c). We hence, in both cases, get to a state s' where $t_{s'}(e) > 0$ and $s' \not\models l$. This concludes the argument. □

Theorem 4. *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an executable basic sound annotated process graph without effect conflicts. Say we run I-propagation on $\mathcal{P}$. There exists exactly one I-propagation result I^*. For all $e \in \mathcal{E}$, $\mathcal{SI}(e) = I^*(e)$. With fixed arity, the time required to compute I^* is polynomial in the size of $\mathcal{P}$.*

Proof First, it is a direct consequence of Lemmas 6 and 7 that, for all $e \in \mathcal{E}$, $\mathcal{SI}(e) = I^*(e)$, for any I-propagation result I^*. From this it follows directly that there exists exactly one such I^*.

For time complexity, there are three issues to consider: (1) the time taken for compiling binary clauses away, (2) the time taken within any single propagation step, and (3) the maximal number of propagation steps performed. (1) consists of computing, for every task node n, the set $\overline{\text{eff}(n)}$ of literals that are implied by $\text{eff}(n) \wedge \mathcal{T}$. This can be done as follows. We view $\mathcal{T}$ as a directed graph whose nodes are literals and whose edges correspond to the clauses. The number of nodes of the graph is the number of literals $|P[C]|$, where C is the set of constants mentioned by α. We compute the transitive closure of that graph, in time $O(|P[C]|^3)$. Then, for every effect $\text{eff}(n)$ and for every literal l, we ask whether there is an edge $(\neg l, l)$ in the transitive closure, or whether for any literal $l' \in \text{eff}(n)$ there is an edge (l', l) in the transitive closure. This is done in time $O(|\mathcal{N}| * |P[C]| * \max_{\text{eff}})$, where $\max_{\text{eff}}$ is the maximum number of effect literals any task node has.

As for (2), loop nodes and end nodes take time $O(|P[C]|)$ since sets can be intersected in linear time using, e.g., a bitvector representation. Parallel and xor joins/splits, accordingly, take time $O(|P[C]| * \max_E)$, where $\max_E$ is the maximum number of incoming or outgoing edges any node has. Task nodes take time $O(|\mathcal{E}| * |P[C]|)$.

As for (3), define, for any function $I : \mathcal{E} \mapsto 2^{P[C]}$, $||I|| := \sum_{e \in \mathcal{E}} |I(e)|$. That is, $||I||$ counts the total number of literals annotated by I, in sum over all edges in the process. I-propagation admits a propagation step from I to I' only if $I \neq I'$. Since we always have, for all $e \in \mathcal{E}$, that $I(e) \supseteq I'(e)$, this means that $||I'|| \leq ||I|| - 1$. If $||I'|| = 0$, then certainly a fixpoint is reached. Now, obviously $||I_0|| \leq |\mathcal{E}| * |P[C]|$. Hence $|\mathcal{E}| * |P[C]|$ is an upper bound on the number of propagation steps performed.

Overall, we get that the runtime is in $O(|P[C]|^3 + |\mathcal{N}| * |P[C]| * \max_{\text{eff}} + (|P[C]| * \max_E + |\mathcal{E}| * |P[C]|) * |\mathcal{E}| * |P[C]|)$. With fixed arity, $|P[C]|$ is $O(|P| * |C|)$, which concludes the argument. □

A.4 I-propagation can be used for executability checking

Lemma 1. *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph without effect conflicts, which is basic except that α may be defined for edges and loop nodes. Say*

we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $e \in \mathcal{E}$, $\mathcal{SI}(e) \supseteq I^*(e)$.

Proof The only differences of this claim compared to Lemma 6 is that we do not require $\mathcal{P}$ to be executable, and that conditions may be annotated at xor splits and loops.

Let $\mathcal{P}_0 = (N, E, \lambda, \Omega, \alpha_0)$ be like $\mathcal{P}$ except that $\text{pre}_0(n)$ has been set to $\emptyset$ for all $n \in \mathcal{N}_T$, and that α_0 is undefined on all xor edges and loop nodes. Obviously, $\mathcal{P}_0$ is executable and basic. Hence we can apply Lemma 6, and get that $\mathcal{SI}^{\mathcal{P}_0}(e) \supseteq I_0^*(e)$ where I_0^* is an I-propagation result for $\mathcal{P}_0$. Since $\mathcal{P}_0$ differs from $\mathcal{P}$ only in terms of the task node preconditions and the xor/loop conditions, which are not considered by I-propagation, we have $I_0^* = I^*$ where I^* is an I-propagation result for $\mathcal{P}$, and hence $\mathcal{SI}^{\mathcal{P}_0}(e) \supseteq I^*(e)$. In what follows, we show that $\mathcal{SI}^{\mathcal{P}_0}(e) \subseteq \mathcal{SI}^{\mathcal{P}}(e)$. Obviously, this proves the claim.

Consider the execution paths through $\mathcal{P}$ and $\mathcal{P}_0$. Let us denote the set of these paths with $\vec{P}$ and $\vec{P}_0$, respectively. $\mathcal{P}_0$ does not alter the structure of $\mathcal{P}$ in any way other than removing preconditions and xor/loop conditions. So the only difference is that some paths are disallowed in $\mathcal{P}$—but are allowed in $\mathcal{P}_0$—due to preconditions or conditions that are not satisfied along the path. Hence we have $\vec{P} \subseteq \vec{P}_0$. Consider now a particular edge $e \in \mathcal{E}$, and consider the sets of states

(A) $\{s \mid s \in \mathcal{S}^{\mathcal{P}}, t_s(e) > 0\}$
(B) $\{s \mid s \in \mathcal{S}^{\mathcal{P}_0}, t_s(e) > 0\}$

With what we just said about paths, we have that (B) is a superset of (A). Now, by definition, $\mathcal{SI}^{\mathcal{P}}(e)$ is the set of literals satisfied by all states in (A), and $\mathcal{SI}^{\mathcal{P}_0}(e)$ is the set of literals satisfied by all states in (B). Since (B) is a superset of (A), this means that $\mathcal{SI}^{\mathcal{P}_0}(e) \subseteq \mathcal{SI}^{\mathcal{P}}(e)$, which is what we needed to show. This concludes the argument. □

Theorem 5. *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be a basic annotated process graph without effect conflicts. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then $\mathcal{P}$ is executable iff for all $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\} : \text{pre}(n) \subseteq I^*(in(n))$.*

Proof Recall that a task node $n \in \mathcal{N}_T$ is executable iff, for all reachable states s so that $t_s(in(n)) > 0$, we have $s \models \text{pre}(n)$. In other words, whenever a path of transitions reaches n with a token, n's precondition is satisfied. $\mathcal{P}$ is executable if all its nodes are executable. Obviously, a node n is executable iff $\text{pre}(n) \subseteq \mathcal{SI}(in(n))$.

First, consider the direction from left to right. $\mathcal{P}$ is executable, so we can apply Lemma 7 and get that $I^*(e) \supseteq \mathcal{SI}(e)$ for all $e \in \mathcal{E}$. Let $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$ be arbitrary. We have $\mathcal{SI}(in(n)) \subseteq I^*(in(n))$. Since n is executable, we have $\text{pre}(n) \subseteq \mathcal{SI}(in(n))$. Hence $\text{pre}(n) \subseteq I^*(in(n))$ as desired.

Now, consider the direction from right to left. We can apply Lemma 1, and get that $I^*(e) \subseteq \mathcal{SI}(e)$ for all $e \in \mathcal{E}$. Assume to the contrary of the claim that $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$ so that $\text{pre}(n) \subseteq I^*(in(n))$, but n is not executable, i.e., ex. $l \in \text{pre}(n) \setminus \mathcal{SI}(in(n))$. We have that $l \in \text{pre}(n)$ and hence $l \in I^*(in(n))$. With the above, this implies that $l \in \mathcal{SI}(e)$, in contradiction. Hence all $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$ are executable, which concludes the proof. □

Theorem 6. *Let $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ be an annotated process graph without effect conflicts, which is basic except that α may be defined for edges and loop nodes. Say we run I-propagation on $\mathcal{P}$, and I^* is an I-propagation result. Then, for all $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$, if $\mathrm{pre}(n) \subseteq I^*(in(n))$ then n is executable.*

Proof Recall that a task node $n \in \mathcal{N}_T$ is executable iff, for all reachable states s so that $t_s(in(n)) > 0$, we have $s \models \mathrm{pre}(n)$. In other words, whenever a path of transitions reaches n with a token, n's precondition is satisfied.

We can apply Lemma 1, and get that $I^*(e) \subseteq \mathcal{SI}(e)$ for all $e \in \mathcal{E}$. Assume to the contrary of the claim that $n \in \mathcal{N}_T \cup \{n_+^{\mathcal{P}}\}$ so that $\mathrm{pre}(n) \subseteq I^*(in(n))$, but n is not executable, i.e., ex. $l \in \mathrm{pre}(n) \setminus \mathcal{SI}(in(n))$. We have that $l \in \mathrm{pre}(n)$ and hence $l \in I^*(in(n))$. With the above, this implies that $l \in \mathcal{SI}(e)$, in contradiction. Hence n is executable, which concludes the proof. □

A.5 Complexity results

Lemma 8 *Assume a sound atomic annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T$, $\mathrm{eff}(n_0)$ is a complete assignment, all predicates have arity 0, and $\mathcal{P}$ is basic except that $\mathcal{T}$ is not binary. Even if $\mathcal{P}$ is known to be reachable, deciding whether $\mathcal{P}$ is executable, or whether some $n \in N$ is executable, is $\mathbf{\Pi_2^p}$-hard for general $\mathcal{T}$, and* **coNP***-hard if $\mathcal{T}$ is Horn. Deciding whether $\mathcal{P}$ is reachable, or whether some $n \in N$ is reachable, is $\mathbf{\Sigma_2^p}$-hard for general $\mathcal{T}$, and* **NP***-hard if $\mathcal{T}$ is Horn.*

Proof Let us first consider the general case, with no restrictions on $\mathcal{T}$. The proofs are by reduction of validity of a QBF formula $\forall X.\exists Y.\phi[X, Y]$, where ϕ is in CNF. The process graphs $\mathcal{P}$ in our construction are very similar for reachability and executability; we first consider the common parts, then explain the details below.

We have a node $n_t \in N$ which is connected to the start node n_0 via an edge $(n_0, n_t) \in E$. We set $\mathrm{pre}(n_t) = \emptyset$. The main trick of the proof lies in the definitions of Ω, $\mathrm{eff}(n_0)$, and $\mathrm{eff}(n_t)$. Those are adapted from the constructions used in the proof of Lemma 6.2 from [28]. The predicates P of Ω are all 0-ary, i.e., they have no arguments and are hence logical propositions. Precisely, we have the predicates $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$ from the formula $\forall X.\exists Y.\phi[X, Y]$, as well as new predicates $\{z_1, \ldots, z_m, q, t\}$. We define $\mathrm{eff}(n_0)$ to contain all x_i, all y_i, all z_i, q, and $\neg t$. So all facts except t are made true by the start state s_0; note that the start state is complete. We define $\mathrm{eff}(n_t)$ to be $\{t\}$. The complex part of the construction lies in the theory $\mathcal{T}$ of Ω. We define

$$\mathcal{T} := \left(\bigwedge_{i=1}^{m}(\neg t \vee x_i \vee z_i)\right) \wedge \left(\bigwedge_{i=1}^{m}(\neg t \vee \neg x_i \vee \neg z_i)\right) \wedge \left(\bigwedge_{C \in \phi}(\neg t \vee \neg q \vee C)\right) \wedge \left(\bigwedge_{i=1}^{n}(\neg t \vee \neg y_i \vee q)\right)$$

where ϕ is viewed as a set of clauses C. More readably, the theory is equivalent to:

$$t \Rightarrow \left[\left(\bigwedge_{i=1}^{m} x_i \equiv \neg z_i\right) \wedge (q \Rightarrow \phi) \wedge \left(\left(\bigvee_{i=1}^{n} y_i\right) \Rightarrow q\right)\right].$$

Note that $\text{eff}(n_t)$ is consistent with the theory: any interpretation that sets r and all y_i to 0 satisfies $\mathcal{T} \wedge \text{eff}(n_t)$. Hence n_t complies with Definition 3.

We now prove that (*) $\forall X.\exists Y.\phi[X, Y]$ *is valid iff* q *is true in any state* s *that results from executing* n_t. From this, the desired hardness results will be easy to obtain. We denote with S the set of states s that may be reached by executing n_t.

The theory conjuncts $x_i \equiv \neg z_i$ make sure that each $s \in S$ makes exactly one of x_i, z_i true. In particular, the different assignments to X are incomparable with respect to set inclusion. Hence, we have that for every assignment a_X of truth values to X, there exists a state $s \in S$ that complies with a_X: a_X is satisfiable together with $\mathcal{T} \wedge \text{eff}(n_t)$, and any other assignment a'_X is more distant from s_0 in at least one proposition (e.g., if $a'_X(x_i) = 1$ and $a_X(x_i) = 0$ then a_X is closer to s_0 than a'_X regarding the interpretation of z_i).

We first prove that, if q is true in any state s that results from executing n_t, then $\forall X.\exists Y.\phi[X, Y]$ is valid. Let a_X be a truth value assignment to X. With the above, we have a state $s \in S$ that complies with a_X. By assumption, s makes q true. Therefore, due to the theory conjunct $q \Rightarrow \phi$, we have $i_s \models \phi$. Obviously, the values assigned to Y by i_s satisfy ϕ for a_X.

For the other direction, say $\forall X.\exists Y.\phi[X, Y]$ is valid. Assume that, to the contrary of the claim, there exists a $s \in S$ so that $i_s \not\models q$. But then, due to the theory conjunct $(\bigvee_{i=1}^{n} y_i) \Rightarrow q$, we have that s sets all y_i to false. Now, because $\forall X.\exists Y.\phi[X, Y]$ is valid, there exists a truth value assignment a_Y to Y that complies with the setting of all x_i and z_i in s. Obtain s' by modifying s to comply with a_Y, and setting q to 1. We have that $i_{s'} \models \mathcal{T} \wedge \text{eff}(n_t)$. But then, s' is closer to s_0 than s, and hence $s \notin S$ in contradiction. This concludes the argument for (*).

To prove $\mathbf{\Pi_2^p}$-hardness of deciding executability, we now simply connect n_t via an edge (n_t, n_+) to the stop node, and set $\text{pre}(n_+) = \{q\}$. By (*), n_+ is executable iff $\forall X.\exists Y.\phi[X, Y]$ is valid; since the other nodes have no preconditions and are trivially executable, and since all nodes are trivially reachable, the claim follows.

To prove $\mathbf{\Sigma_2^p}$-hardness of deciding reachability, an only slightly more complex construction is required. We introduce another node $n_{\neg q} \in N$, and connect $(n_t, n_{\neg q})$ as well as $(n_{\neg q}, n_+)$. We set $\text{pre}(n_{\neg q}) = \{\neg q\}$, and $\text{eff}(n_{\neg q}) = \text{pre}(n_+) = \emptyset$. Then, by (*), n_+ is reachable iff $\forall X.\exists Y.\phi[X, Y]$ is *not* valid; the other nodes are trivially reachable; this concludes the argument.

Let's consider now the case where $\mathcal{T}$ is restricted to be Horn. The graphs (N, E) that we use for reachability/executability remain exactly the same. What changes is the semantic annotation. The latter is obtained by the following adaptation of the proof of Lemma 7.1 from [28]. The proof works by a reduction of satisfiability of a CNF formula $\phi[X]$. We use the 0-ary predicates $X = \{x_1, \ldots, x_m\}$, and new 0-ary predicates $Y = \{y_1, \ldots, y_n, z_1, \ldots, z_n, q, t\}$. As before, $\text{pre}(n_t) = \emptyset$ and $\text{eff}(n_t) = \{t\}$. We define $\text{eff}(n_0)$ to contain all x_i, all y_i, all $\neg z_i$, $\neg q$, and $\neg t$; note that this is a

complete assignment. The theory is:

$$\left(\neg t \vee \left(\bigvee_{i=1}^{n} \neg z_i\right) \vee q\right)$$
$$\wedge\left(\bigwedge_{i=1}^{n}((\neg t \vee \neg x_i \vee \neg y_i) \wedge (\neg t \vee \neg x_i \vee z_i) \wedge (\neg t \vee \neg y_i \vee z_i))\right)$$
$$\wedge\left(\bigwedge_{C \in \phi}(\neg t \vee C[-Y/+X])\right)$$

where ϕ is viewed as a set of clauses C, and $C[-Y/+X]$ for a clause C denotes the modification of C where every occurrence of a positive literal x_i is replaced with $\neg y_i$. More readably, the theory is equivalent to:

$$t \Rightarrow \left[\left(\left(\bigwedge_{i=1}^{n} z_i\right) \Rightarrow q\right) \wedge \left(\bigwedge_{i=1}^{n}((\neg x_i \vee \neg y_i) \wedge (x_i \Rightarrow z_i) \wedge (y_i \Rightarrow z_i))\right)\right.$$
$$\left.\wedge\left(\bigwedge_{C \in \phi} C[-Y/+X]\right)\right]$$

Obviously, this theory is in Horn format: every clause contains at most one positive literal. Note that $\text{eff}(n_t)$ is consistent with the theory: e.g., the interpretation that sets all propositions except t to 0 satisfies $\mathcal{T} \wedge \text{eff}(n_t)$. Hence n_t complies with Definition 3.

The key in this transformation is that ϕ is made Horn by replacing positive occurrences of x_i with $\neg y_i$. If the truth value of y_i is different from the value of x_i, for each i, then $C[-Y/+X]$ is satisfied by this assignment iff C is satisfied. The role of z_i is to indicate whether x_i and y_i are indeed different. The role of q is to indicate whether the latter is the case for all i.

We now prove that (**) $\phi[X]$ *is unsatisfiable iff* $\neg q$ *is true in any state* s *that results from executing* n_t. From this, the desired hardness results will be easy to obtain. We denote with S the set of states s that may be reached by executing n_t.

We first prove that, if there exists $s \in S$ so that $i_s \models \neg q$, then ϕ is satisfiable. Let L_0 be the set of literals on whose interpretation s agrees with s_0. We can conclude that $\mathcal{T} \wedge \text{eff}(n_t) \wedge \bigwedge_{l \in L_0} l \wedge \neg q$ is unsatisfiable, since otherwise we can construct a state s' that has $s' \models L_0 \wedge \neg q$ and that is hence closer to s_0 than s. The only part of $\mathcal{T} \wedge \text{eff}(n_t)$ that forces implication of q is $(\bigwedge_{i=1}^{n} z_i) \Rightarrow q$. Thus we infer that $\mathcal{T} \wedge \text{eff}(n_t) \wedge \bigwedge_{l \in L_0} l \models \bigwedge_{i=1}^{n} z_i$. The only part of $\mathcal{T} \wedge \text{eff}(n_t)$ that forces implication of z_i is if either x_i or y_i are true. Hence, for all i, either x_i or y_i are implied by $\mathcal{T} \wedge \text{eff}(n_t) \wedge \bigwedge_{l \in L_0} l$. Hence, in particular s satisfies, for all i, either $i_s \models x_i$ or $i_s \models y_i$. Hence the value of x_i and y_i is different for all i, and hence, with the above, the assignment that s makes to X satisfies ϕ.

For the other direction, assume that ϕ is satisfiable, by the truth value assignment a_X. We construct a state s so that $s \models q$ and $s \in S$. First, we set that for all x_i, $i_s \models x_i$

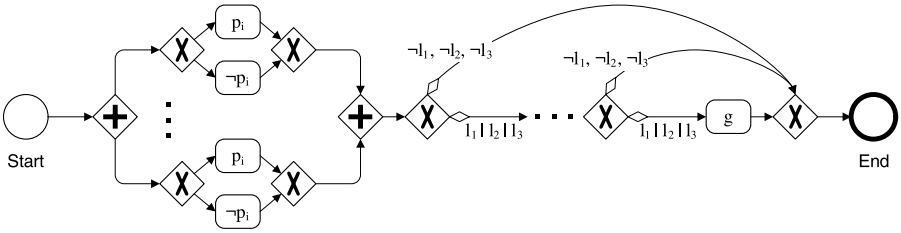


Fig. 9 Schematic illustration of 3SAT reduction for Lemma 9, reachability checking

iff $a_X(x_i) = 1$. Then, we set that for all y_i, $i_s \models y_i$ iff $a_X(x_i) = 0$. We set that for all z_i, $i_s \models z_i$. Finally, we set $i_s \models q$ and $i_s \models t$. It is easily verified that $i_s \models \mathcal{T} \wedge \text{eff}(n_t)$: ϕ is satisfied because the values of x_i and y_i are different, for each i. Further, s is maximally close to s_0. This can be seen as follows. First, we cannot change any of the values of a z_i or of q, because those are implied by the distinct values of each x_i and y_i. Second, we cannot set any x_i or y_i to true in isolation, because that would be in conflict with the respective other value. So any change we make to the setting of x_i and y_i would involve switching one x_i or y_i to false, and hence being further away from s_0 in that proposition. This concludes the argument for (**).

To prove $\mathbf{\Pi_2^p}$-hardness of deciding executability, as before connect n_t via an edge (n_t, n_+) to the stop node. We set $\text{pre}(n_+) = \{\neg q\}$. By (**), n_+ is executable iff $\phi[X]$ is unsatisfiable; since the other nodes have no preconditions and are trivially executable, and since all nodes are trivially reachable, the claim follows.

To prove $\mathbf{\Sigma_2^p}$-hardness of deciding reachability, we introduce another node $n_q \in N$, and connect (n_t, n_q) as well as (n_q, n_+). We set $\text{pre}(n_q) = \{q\}$, and $\text{eff}(n_q) = \text{pre}(n_+) = \emptyset$. Then, by (*), n_+ is reachable iff $\phi[X]$ is satisfiable; the other nodes are trivially reachable; this concludes the argument. □

Lemma 9 *Assume a sound atomic annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ}$, $\text{eff}(n_0)$ is a complete assignment, all predicates have arity 0, and $\mathcal{P}$ is basic except that con(e) may be defined for some $e \in E$. Even if $\mathcal{P}$ is known to be reachable, deciding whether $\mathcal{P}$ is executable, or whether some $n \in N$ is executable, is* **coNP***-hard. Even if $\mathcal{P}$ is known to be executable, deciding whether $\mathcal{P}$ is reachable, or whether some $n \in N$ is reachable.*

Proof The proof for reachability checking is by the following reduction from 3SAT. Assume a CNF ϕ with n propositions $p_1, \ldots, p_n$, and k clauses $c_1, \ldots, c_k$ where $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$. We obtain an atomic basic annotated process graph with some annotated edges, $(N, E, \lambda, \Omega, \alpha)$, as follows. The ontology contains only 0-ary predicates, namely $P := \{p_1, notp_1, \ldots, p_n, notp_n\}$; we identify literal $\neg p_i$ with proposition $notp_i$. The construction is illustrated in Fig. 9.

The set of nodes N and their annotation (of which we show only the non-empty ones) is:

1. start node n_0; $\text{eff}(n_0) = \emptyset$
2. parallel split node nps
3. xor-split nodes $nxs_1 \cdots nxs_n$

4. for $1 \leq i \leq n$: task nodes np_i and $nnotp_i$; $\text{eff}(np_i) = \{p_i\}$, $\text{eff}(np_i) = \{notp_i\}$
5. xor-join nodes $nxj_1 \cdots nxj_n$
6. parallel join node npj
7. for $1 \leq i \leq k$: xor-split node nxs'_i
8. for $1 \leq i \leq k-1$: xor-join node nxj'_i
9. task node ng
10. xor-join node nxj'
11. stop node n_+

The set of edges E and their annotation is given below. Again, empty annotation is not shown; also, the position of the annotated edges does not matter and is hence not specified.

1. (n_0, nps)
2. for $1 \leq i \leq n$: (nps, nxs_i)
3. for $1 \leq i \leq n$: (nxs_i, np_i) and $(nxs_i, nnotp_i)$
4. for $1 \leq i \leq n$: (np_i, nxj_i) and $(nnotp_i, nxj_i)$
5. for $1 \leq i \leq n$: (nxj_i, npj)
6. (npj, nxs_1)
7. for $1 \leq i \leq k$: (nxs'_i, nxj'); $\text{con}((nxs'_i, nxj')) = \{\neg l_{i1}, \neg l_{i2}, \neg l_{i3}\}$
8. for $1 \leq i \leq k-1$: for $1 \leq j \leq 3$: (nxs'_i, nxj'_i); $\text{con}((nxs'_i, nxj'_i)) = \{l_{ij}\}$
9. for $1 \leq i \leq k-1$: (nxj'_i, nxs'_{i+1})
10. for $1 \leq j \leq 3$: (nxs'_k, ng); $\text{con}((nxs'_k, ng)) = \{l_{kj}\}$
11. (ng, nxj')
12. (nxj', n_+)

Since all preconditions are empty, it is obvious that $\mathcal{P}$ is executable. By construction, $\mathcal{P}$ is reachable iff ng is reachable. The latter is the case iff ϕ is satisfiable. The annotation of the start node can be set to be $\text{eff}(n_0) = \{\neg p_1, \neg notp_1, \ldots, \neg p_n, \neg notp_n\}$, and hence to be complete. The parallel split/join can be replaced by a simple sequencing of all the xors setting proposition values.

For executability checking, we can use a similar reduction. Given a CNF ϕ, let p be a new proposition; obtain ϕ' by inserting p into every clause of ϕ. Then construct, for ϕ', the process graph as above, with the only difference being that ng has the annotation $\text{pre}(ng) = \{p\}$. With this construction, we have that (1) ng is reachable (trivially, by making p true and choosing the p-branch for every clause); (2) with that, clearly all nodes are reachable; and (3) ng is executable iff every satisfying assignment to ϕ' makes p true. The latter is, obviously, the case iff ϕ is unsatisfiable. Since ng is the only node with a precondition, all other nodes are trivially executable and hence the claim follows. □

Lemma 10 *Assume a sound annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ}$, for all $n \in N_L$ we have $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$, $\text{eff}(n_0)$ is a complete assignment, all predicates have arity 0, and $\mathcal{P}$ is basic except that $\text{con}(n)$ may be defined for some $n \in N_L$. Even if $\mathcal{P}$ is known to be reachable, deciding whether $\mathcal{P}$ is executable, or whether some $n \in N$ is executable, is* **coNP***-hard. Even if $\mathcal{P}$ is known to be executable, deciding whether $\mathcal{P}$ is reachable, or whether some $n \in N$ is reachable.*

Proof This can be proved via a 3SAT reduction very similar to that used for proving Lemma 9. We simply replace each edge condition with a loop node n where $\lambda(n)$ points to an empty sub-process—consisting only of start and end node. The idea is to only allow exiting the loop if the edge condition holds true. The only tricky bit here lies in the interpretation of edge conditions and repetition conditions. An edge condition $\text{con}(e)$ means that the edge can be taken when $\text{con}(e)$ is true. A repetition condition $\text{con}(n)$ means that the loop is repeated if $\text{con}(n)$ is true. Our construction necessitates us to say the opposite, i.e., we want to state a condition under which the loop may be exited. The solution is, of course, to use $\neg\text{con}(e)$ as the repetition condition. If $\text{con}(e)$ contains several literals, then $\neg\text{con}(e)$ is a disjunction, which is not supported by repetition conditions. However, we can obtain the desired effect by creating, in this case, one loop node for every literal in $\text{con}(e)$.

In detail, the reduction works as follows. We assume a CNF ϕ with n propositions $p_1, \ldots, p_n$, and k clauses $c_1, \ldots, c_k$ where $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$. The construction follows the same scheme as depicted in Fig. 9, and the reader is advised to consider this figure when reading the following. The set of nodes N and their annotation (of which we show only the non-empty ones) is:

1. start node n_0; $\text{eff}(n_0) = \emptyset$
2. parallel split node nps
3. xor-split nodes $nxs_1 \cdots nxs_n$
4. for $1 \leq i \leq n$: task nodes np_i and $nnotp_i$; $\text{eff}(np_i) = \{p_i\}$, $\text{eff}(np_i) = \{notp_i\}$
5. xor-join nodes $nxj_1 \cdots nxj_n$
6. parallel join node npj
7. for $1 \leq i \leq k$: xor-split node nxs'_i
8. for $1 \leq i \leq k$: loop node nl_i with $\text{con}(nl_i) = \{\neg l_{i1}, \neg l_{i2}, \neg l_{i3}\}$; for $1 \leq j \leq 3$: loop node nl_i^j with $\text{con}(nl_i^j) = \{l_{ij}\}$
9. for $1 \leq i \leq k-1$: xor-join node nxj'_i
10. task node ng
11. xor-join node nxj'
12. stop node n_+

As stated, λ points to an empty sub-process for every loop node. The set of edges E is:

1. (n_0, nps)
2. for $1 \leq i \leq n$: (nps, nxs_i)
3. for $1 \leq i \leq n$: (nxs_i, np_i) and $(nxs_i, nnotp_i)$
4. for $1 \leq i \leq n$: (np_i, nxj_i) and $(nnotp_i, nxj_i)$
5. for $1 \leq i \leq n$: (nxj_i, npj)
6. (npj, nxs_1)
7. for $1 \leq i \leq k-1$: (nxs'_i, nl_i) and (nl_i, nxj'_i)
8. for $1 \leq i \leq k$: (nxj'_i, nl_i^1), (nl_i^1, nl_i^2), (nl_i^2, nl_i^3), (nl_i^3, nxj')
9. for $1 \leq i \leq k-1$: (nxj'_i, nxs'_{i+1})
10. (nxs'_k, nl_k) and (nl_k, ng)
11. (ng, nxj')
12. (nxj', n_+)

Since all preconditions are empty, it is obvious that $\mathcal{P}$ is executable. By construction, $\mathcal{P}$ is reachable iff ng is reachable. The repetition conditions at nodes nl_i ensure that one can exit the loop iff clause i is satisfied. The repetition conditions at the sequenced nodes nl_i^1, nl_i^2, nl_i^3 ensure that one can traverse the entire sequence iff clause i is violated. Hence ng is reachable iff ϕ is satisfiable. The annotation of the start node can be set to be $\text{eff}(n_0) = \{\neg p_1, \neg notp_1, \ldots, \neg p_n, \neg notp_n\}$, and hence to be complete. The parallel split/join can be replaced by a simple sequencing of all the xors setting proposition values.

For executability checking, we can use a similar reduction. Given a CNF ϕ, let p be a new proposition; obtain ϕ' by inserting p into every clause of ϕ. Then construct, for ϕ', the process graph as above, with the only difference being that ng has the annotation $\text{pre}(ng) = \{p\}$. With this construction, we have that (1) ng is reachable (trivially, by making p true and choosing the p-branch for every clause); (2) with that, clearly all nodes are reachable; and (3) ng is executable iff every satisfying assignment to ϕ' makes p true. The latter is, obviously, the case iff ϕ is unsatisfiable. Since ng is the only node with a precondition, all other nodes are trivially executable and hence the claim follows. □

Lemma 11 *Assume a basic sound atomic annotated process graph* $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ *without effect conflicts, where* $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ}$, $\text{eff}(n_0)$ *is a complete assignment, and all predicates have arity* 0. *Even if* $\mathcal{P}$ *is known to be reachable, deciding whether* $n \in N$ *is executable is* **coNP***-hard. Deciding whether* $\mathcal{P}$ *is reachable, or whether* $n \in N$ *is reachable, is* **NP***-hard.*

Proof This can be proved via a 3SAT reduction very similar to that used for proving Lemma 9. The difference to that lemma is that $\mathcal{P}$ is basic, so we cannot make use of edge conditions or of repetition conditions. The main property underlying the situations considered is that non-executable task nodes are allowed. We can simply use those just like edge conditions, to filter the set of execution paths that may traverse a certain branch of the process. Note here that, for executability, we consider only the decision problem asking whether *a particular node* (rather than the entire process) is executable. For reachability, it is noteable that we can not restrict consideration to executable processes—if the process is executable then it is also reachable, cf. Proposition 2.

In detail, the reduction works as follows. We assume a CNF ϕ with n propositions $p_1, \ldots, p_n$, and k clauses $c_1, \ldots, c_k$ where $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$. The construction follows the same scheme as depicted in Fig. 9, and the reader is advised to consider this figure when reading the following.

The set of nodes N and their annotation (of which we show only the non-empty ones) is:

1. start node n_0; $\text{eff}(n_0) = \emptyset$
2. parallel split node nps
3. xor-split nodes $nxs_1 \cdots nxs_n$
4. for $1 \leq i \leq n$: task nodes np_i and $nnotp_i$; $\text{eff}(np_i) = \{p_i\}$, $\text{eff}(np_i) = \{notp_i\}$
5. xor-join nodes $nxj_1 \cdots nxj_n$
6. parallel join node npj

7. for $1 \leq i \leq k$: xor-split node nxs'_i
8. for $1 \leq i \leq k$: for $1 \leq j \leq 3$: task node $nxst_{ij}$; $\text{pre}(nxst_{ij}) = \{l_{ij}\}$
9. for $1 \leq i \leq k$: task node $nxst'_i$; $\text{pre}(nxst'_i) = \{\neg l_{i1}, \neg l_{i2}, \neg l_{i3}\}$
10. for $1 \leq i \leq k-1$: xor-join node nxj'_i
11. task node ng
12. xor-join node nxj'
13. stop node n_+

The set of edges E and their annotation is:

1. (n_0, nps)
2. for $1 \leq i \leq n$: (nps, nxs_i)
3. for $1 \leq i \leq n$: (nxs_i, np_i) and $(nxs_i, nnotp_i)$
4. for $1 \leq i \leq n$: (np_i, nxj_i) and $(nnotp_i, nxj_i)$
5. for $1 \leq i \leq n$: (nxj_i, npj)
6. (npj, nxs_1)
7. for $1 \leq i \leq k$: $(nxs'_i, nxst'_i)$ and $(nxst'_i, nxj')$
8. for $1 \leq i \leq k-1$: for $1 \leq j \leq 3$: $(nxs'_i, nxst_{ij})$ and $(nxst_{ij}, nxj'_i)$
9. for $1 \leq i \leq k-1$: (nxj'_i, nxs'_{i+1})
10. for $1 \leq j \leq 3$: $(nxs'_k, nxst_{kj})$ and $(nxst_{kj}, ng)$
11. (ng, nxj')
12. (nxj', n_+)

Obviously, $\mathcal{P}$ is reachable iff ng is reachable iff ϕ is satisfiable. The annotation of the start node can be set to be $\text{eff}(n_0) = \{\neg p_1, \neg notp_1, \ldots, \neg p_n, \neg notp_n\}$, and hence to be complete. The parallel split/join can be replaced by a simple sequencing of all the xors setting proposition values.

For executability checking, we use a similar reduction. Given a CNF ϕ, let p be a new proposition; obtain ϕ' by inserting p into every clause of ϕ. Then construct, for ϕ', the process graph as above, with the only difference being that ng has the annotation $\text{pre}(ng) = \{p\}$. With this construction, ng is executable iff every satisfying assignment to ϕ' makes p true. The latter is, obviously, the case iff ϕ is unsatisfiable. Since all nodes are reachable (ng can be reached by setting p to be true), this proves the claim. □

Theorem 2. *Assume a sound annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ} \cup N_L$, $\text{eff}(n_0)$ is a complete assignment, all predicates have arity 0, and either $\mathcal{P}$ is atomic or for all $n \in N_L$ we have $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$. The following problem is $\mathbf{\Pi_2^p}$-hard even if $\mathcal{P}$ is known to be reachable*:

– *Is $\mathcal{P}$ executable, or is $n \in N$ executable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary clauses*?

The following problems are **coNP***-hard even if $\mathcal{P}$ is known to be reachable*:

– *Is $\mathcal{P}$ executable, or is $n \in N$ executable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary Horn clauses*?
– *Is $\mathcal{P}$ executable, or is $n \in N$ executable, given that $\mathcal{P}$ is basic except that* $\text{con}(e)$ *may be defined for some $e \in E$*?

– *Is $\mathcal{P}$ executable, or is $n \in N$ executable, given that $\mathcal{P}$ is basic except that* con(n) *may be defined for some $n \in N_L$*?
– *Is $n \in N$ executable, given that $\mathcal{P}$ is basic*?

Proof Follows directly from Lemmas 8, 9, 10, and 11. □

Theorem 3. *Assume a sound annotated process graph $\mathcal{P} = (N, E, \lambda, \Omega, \alpha)$ without effect conflicts, where $N \setminus \{n_0, n_+\} \subseteq N_T \cup N_{XS} \cup N_{XJ} \cup N_L$,* eff$(n_0)$ *is a complete assignment, all predicates have arity* 0*, and either $\mathcal{P}$ is atomic or for all $n \in N_L$ we have $N^{\lambda(n)} = \{n_0^{\lambda(n)}, n_+^{\lambda(n)}\}$. The following problem is $\mathbf{\Sigma_2^p}$-hard*:

– *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary clauses*?

The following problems are **NP***-hard*:

– *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic except that $\mathcal{T}$ may involve arbitrary Horn clauses*?
– *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is executable, and basic except that* con(e) *may be defined for some $e \in E$*?
– *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is executable, and basic except that* con(n) *may be defined for some $n \in N_L$*?
– *Is $\mathcal{P}$ reachable, or is $n \in N$ reachable, given that $\mathcal{P}$ is basic*?

Proof Follows directly from Lemmas 8, 9, 10, and 11. □

References

1. Ankolekar, A., et al.: DAML-S: Web service description for the semantic web. In: ISWC, 2002
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison–Wesley/Longman, Boston (1986)
3. Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf. Process. Lett. **8**, 121–123 (1979)
4. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
5. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: first results. In: AAAI, 2005
6. Beckstein, C., Klausner, J.: A planning framework for workflow management. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999
7. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL implementation secrets. In: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02), pp. 3–22, 2002
8. Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets 1986 Part I: Petri Nets, Central Models and Their Properties. Lecture Notes in Computer Science, vol. 254, pp. 360–376. Springer, Berlin (1987)
9. Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur. **2**(1), 65–104 (1999)
10. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison–Wesley, Reading (2005)
11. Born, M., Dörr, F., Weber, I.: User-friendly semantic annotation in business process modeling. In: Hf-SDDM-07: Proceedings of the Workshop on Human-friendly Service Description, Discovery and Matchmaking—in Workshop Proceedings at WISE-07, December 2007

12. Born, M., Dörr, F., Weber, I.: User-friendly semantic annotation in business process modeling. In: Hf-SDDM'07: Workshop on Human-friendly Service Description, Discovery and Matchmaking at WISE'07, Nancy, France, December 2007
13. Born, M., Hoffmann, J., Kaczmarek, T., Kowalkiewicz, M., Markovic, I., Scicluna, J., Weber, I., Zhou, X.: Semantic annotation and composition of business processes with Maestro. In: European Semantic Web Conference (ESWC) Demo Track, June 2008
14. Born, M., Hoffmann, J., Kaczmarek, T., Kowalkiewicz, M., Markovic, I., Scicluna, J., Weber, I., Zhou, X.: Supporting execution-level business process modeling with semantic technologies. In: Database Systems for Advanced Applications (DASFAA-09) Demo Track, 2009
15. Brewka, G., Hertzberg, J.: How to do things with worlds: on formalizing actions and plans. J. Log. Comput. **3**(5), 517–532 (1993)
16. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**, 677–691 (1986)
17. Burch, J., Clarke, E., Mcmillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^20 states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 1–33, 1990
18. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
19. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
20. Conradi, R., Liu, C., Hagaseth, M.: Planning support for cooperating transactions in EPOS. Inf. Syst. **20**(4), 317–336 (1995)
21. Fensel, D., et al.: Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer, Berlin (2006)
22. Da Rold, C.: European IT services survey signals irreversible changes. Technical Report Markets Note, M-20-0616, Gartner Research, 19 June 2003
23. De Giacomo, G., Lenzerini, M., Poggi, A., Rosati, R.: On the update of description logic ontologies at the instance level. In: AAAI, 2006
24. Dehnert, J., van der Aalst, W.M.P.: Bridging the gap between business models and workflow specifications. Int. J. Cooperative Inf. Syst. **13**(3), 289–332 (2004)
25. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, New York (1995)
26. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit-state model checking in the validation of communication protocols. Int. J. Softw. Tools Technol. (2004)
27. Een, N., Sörensson, N.: An extensible SAT solver. In: Giunchiglia, E. (ed.) Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT-03), Portofino, Italy, May 2003
28. Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates, and counterfactuals. Artif. Intell. **57**(2–3), 227–270 (1992)
29. Garcia-Valles, F., Colom, J.M.: Implicit places in net systems. In: Petri Nets and Performance Models, 1999. Proceedings. The 8th International Workshop on, pp. 104–113 (1999)
30. Governatori, G., Hoffmann, J., Sadiq, S., Weber, I.: Detecting regulatory compliance for business process models through semantic annotations. In: BPD-08: 4th International Workshop on Business Process Design, September 2008
31. Hepp, M., Hinkelmann, K., Karagiannis, D., Klein, R., Stojanovic, N. (eds.) Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM 2007), Innsbruck, Austria, June 2007
32. Herzig, A., Rifi, O.: Propositional belief base update and minimal change. Artif. Intell. **115**(1), 107–138 (1999)
33. Hoffmann, J., Weber, I., Scicluna, J., Kaczmarek, T., Ankolekar, A.: Combining scalability and expressivity in the automatic composition of semantic web services. In: ICWE'08: 8th International Conference on Web Engineering, Yorktown Heights, NY, USA, July 2008
34. Hoffmann, J., Weber, I., Governatori, G.: On compliance checking for clausal constraints in annotated process models. Information Systems Frontiers, Special Issue on Governance, Risk, and Compliance, 2009
35. Holzmann, G.: The Spin Model Checker—Primer and Reference Manual. Addison–Wesley, Reading (2003)
36. Holzmann, G., Peled, D.: An improvement in formal verification. In: Formal Description Techniques, pp. 197–211 (1994)
37. Horn, A.: On sentences which are true of direct unions of algebras. J. Symb. Log. (1951)

38. Howell, R., Rosier, L.: Problems concerning fairness and temporal logic for conflict-free Petri nets. Theor. Comput. Sci. **64**(3), 305–329 (1989)
39. IBM. Insurance Application Architecture (IAA), v 7.1 (2004). http://www-03.ibm.com/industries/financialservices/doc/content/solution/278918103.html, accessed: 28.10.2008
40. Jaccheri, M.L., Conradi, R.: Techniques for process model evolution in EPOS. IEEE Trans. Softw. Eng. **19**(12), 1145–1156 (1993)
41. Keller, G., Nüttgens, M., Scheer, A.-W.: Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89, Saarbrücken, Germany, January 1992. http://www.iwi.uni-sb.de/iwi-hefte/heft089.pdf
42. Kindler, E.: Model-based software engineering and process-aware information systems. Trans. Petri Nets Other Models Concurr. II **2**, 27–45 (2009). Special Issue on Concurrency in Process-Aware Information Systems
43. Koliadis, G., Ghose, A.: Verifying semantic business process models in inter-operation. In: IEEE Intl. Conf. Services Computing (SCC 2007), pp. 731–738, 2007
44. Kovalyov, A., Esparza, J.: A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In: Proc. of the International Workshop on Discrete Event Systems, WODES'96, pp. 1–6, Edinburgh, Scotland, UK, 1996
45. Kumaran, S., Liu, R., Wu, F.Y.: On the duality of information-centric and activity-centric models of business processes. In: Proc. Conf. on Advanced Information Systems Engineering (CAiSE-08), pp. 32–47, 2008
46. Lin, F., Reiter, R.: State constraints revisited. J. Log. Comput. **4**(5), 655–678 (1994)
47. Lutz, C., Sattler, U.: A proposal for describing services with DLs. In: DL, 2002
48. Ly, L.T., Rinderle, S., Dadam, P.: Semantic correctness in adaptive process management systems. In: BPM06: Proc. 4th Int'l Conf. on Business Process Management, pp. 193–208, Vienna, Austria, 2006
49. Ly, L.T., Rinderle, S., Dadam, P.: Integration and verification of semantic constraints in adaptive process management systems. Data Knowl. Eng. **64**(1), 3–23 (2008)
50. Markovic, I., Karrenbrock, M.: Semantic web service discovery for business process models. In: Hf-SDDM'07: Workshop on Human-friendly Service Description, Discovery and Matchmaking at WISE'07, Nancy, France, December 2007
51. Marques-Silva, J., Sakallah, K.A.: GRASP—a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
52. Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. Lecture Notes in Business Information Processing, vol. 6. Springer, Berlin (2008)
53. Mendling, J., van der Aalst, W.M.P.: Formalization and verification of EPCs with OR-joins based on state and context. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) Proceedings of the 19th Conference on Advanced Information Systems Engineering (CAiSE 2007), Trondheim, Norway. Lecture Notes in Computer Science, vol. 4495, pp. 439–453. Springer, Berlin (2007)
54. Meyer, H.: On the semantics of service compositions. In: Web Reasoning and Rule Systems, First International Conference (RR-07), pp. 31–42, 2007
55. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Conference on Design Automation (DAC-01), Las Vegas, Nevada, USA, 2001. IEEE Computer Society, Los Alamitos (2001)
56. Namiri, K., Stojanovic, N.: A model-driven approach for internal controls compliance in business processes. In: SBPM-07: Proc. Workshop on Semantic Business Process and Product Lifecycle Management, Innsbruck, Austria, June 2007. ISSN 1613-0073
57. Narayanan, S., McIlraith, S.: Simulation, verification and automated composition of web services. In: 11th International World Wide Web Conference (WWW-02), pp. 77–88, 2002
58. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007
59. OMG. Business Process Modeling Notation, V1.1. http://www.bpmn.org/, January 2008. OMG Available Specification, Document Number: formal/2008-01-17
60. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science, pp. 46–57. IEEE Computer Society Press, Providence (1977)
61. Puhlmann, F., Weske, M.: Investigations on soundness regarding lazy activities. In: Dustdar, S., Fiadeiro, J.L., Sheth, A. (eds.) Business Process Management, 4th International Conference, BPM 2006. Lecture Notes in Computer Science, vol. 4102, pp. 145–160. Springer, Berlin (2006)
62. Reichert, M., Rinderle, S., Dadam, P.: ADEPT workflow management system: flexible support for enterprise-wide business processes. In: BPM, 2003

63. Reichert, M., Rinderle, S., Dadam, P.: ADEPT workflow management system: flexible support for enterprise-wide business processes (tool presentation). In: BPM03: Proc. Int'l Conf. on Business Process Management, Eindhoven, Netherlands, June 2003, pp. 370–379. Springer, Berlin (2003)
64. Reichert, M., Rinderle, S., Kreher, U., Dadam, P.: Adaptive process management with ADEPT2. In: ICDE, 2005
65. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distrib. Parallel Databases **16**(1), 91–116 (2004)
66. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Appl. Ontol. **1**(1) (2005)
67. Ryndina, K., Küster, J.M., Gall, H.: Consistency of business process models and object life cycles. In: MoDELS Workshops. Lecture Notes in Computer Science, vol. 4364, pp. 80–90. Springer, Berlin (2006)
68. Sadiq, S., Orlowska, M., Sadiq, W.: Specification and validation of process constraints for flexible workflows. J. Inf. Syst. **30**(5), 349–378 (2005)
69. Scheer, A.-W.: ARIS Business Process Modelling. Springer, Berlin (2000)
70. Sinur, J., Hill, J.B.: Align BPM and SOA Initiatives Now to Increase Chances of Becoming a Leader by 2010. Gartner Predicts 2007, 10 November 2006
71. Strichman, O.: Accelerating bounded model checking of safety formulas. Form. Methods Syst. Des. **24**(1), 5–24 (2004)
72. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services (2003)
73. Valmari, A.: A stubborn attack on state explosion. In: Proceedings of the 2nd Workshop on Computer Aided Verification (CAV'90), pp. 156–165, 1990
74. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) Application and Theory of Petri Nets 1997. Lecture Notes in Computer Science, vol. 1248, pp. 407–426. Springer, Berlin (1997)
75. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Inf. Softw. Technol. **41**(10), 639–650 (1999)
76. van der Aalst, W.M.P.: Interorganizational workflows: an approach based on message sequence charts and Petri nets. Syst. Anal. Model. Simul. **34**(3), 335–367 (1999)
77. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Inf. Syst. **30**(4), 245–275 (2005)
78. van der Aalst, W.M.P., van Hee, K.: Workflow Management: Models, Methods, and Systems (Cooperative Information Systems). MIT Press, Cambridge (2002)
79. van der Aalst, W.M.P., Hirnschall, A., Verbeek, H.M.W.: An alternative way to analyze workflow graphs. In: Banks-Pidduck, A., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.): Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02). Lecture Notes in Computer Science, vol. 2348, pp. 535–552. Springer, Berlin (2002)
80. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: an approach based on temporal logic. In: Meersman, R., Tari, Z., Hacid, M.-S., Mylopoulos, J., Pernici, B., Babaoglu, Ö., Jacobsen, H.-A., Loyall, J.P., Kifer, M., Spaccapietra, S. (eds.) OTM Conferences (1). Lecture Notes in Computer Science, vol. 3760, pp. 130–147. Springer, Berlin (2005)
81. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models though SESE decomposition. In: Krämer, B., Lin, K.J., Narasimhan, P. (eds.): 5th International Conference on Service-Oriented Computing (ICSOC). Lecture Notes in Computer Science, vol. 4749, pp. 43–55. Springer, Berlin (2007)
82. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2–4, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5240, pp. 100–115. Springer, Berlin (2008)
83. Verbeek, H.M.W., Basten, T., van der Aalst, W.M.P.: Diagnosing workflow processes using Woflan. Comput. J. **44**(4), 246–279 (2001)
84. Weber, I., Hoffmann, J., Mendling, J., Nitzsche, J.: Towards a methodology for semantic business process modeling and configuration. In: Proceedings of the ICSOC 2007 Workshops. Lecture Notes in Computer Science. Springer, Berlin (2008)
85. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the semantic consistency of executable process models. In: ECOWS-08: Proceedings of the 6th IEEE European Conference on Web Services, pp. 102–111, November 2008
86. Weber, I., Hoffmann, J., Mendling, J.: Semantic business process validation. In: SBPM-08: 3rd International Workshop on Semantic Business Process Management at ESWC-08, June 2008

87. Weber, I., Markovic, I., Drumm, C.: A conceptual framework for semantic business process configuration. J. Inf. Sci. Technol. (JIST) **5**(2), 3–20 (2008)
88. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the verification of semantic business process models. Technical report, 2009. Available at http://www.imweber.de/texte/tr-dpd.pdf
89. Weber, I., Barros, A., May, N., Hoffmann, J., Kaczmarek, T.: Composing services for third-party service delivery. In: ICWS-09: IEEE International Conference on Web Services, Application and Industry Track, Los Angeles, CA, July 2009
90. Weber, I., Governatori, G., Hoffmann, J.: Approximate compliance checking for annotated process models. In: Advances in Enterprise Engineering—Proceedings of the GRCIS workshop at CAiSE'08, June 2008
91. Winslett, M.: Reasoning about actions using a possible models approach. In: AAAI, 1988
92. Zhao, W., Hauser, R., Bhattacharya, K., Bryant, B.R., Cao, F.: Compiling business processes: untangling unstructured loops in irreducible flow graphs. Int. J. Web Grid Serv. **2**(1), 68–91 (2006)
93. zur Muehlen, M., Recker, J.: How much language is enough? Theoretical and practical use of the business process modeling notation. In: Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08), 2008