Beyond Soundness: On the Semantic Consistency of Executable Process Models

Ingo Weber, Jörg Hoffmann, Jan Mendling

1 SAP Research Karlsruhe, Germany
first.last@sap.com
2 BPM Group, Faculty of IT, Queensland University of Technology, Australia
j.mendling@qut.edu.au

Abstract. Executable business process models (like BPEL-based Web service orchestrations) build on the specification of process activities, their implemented business functions (e.g., Web services) and the control flow between these activities. Before deploying such a model, it is important to verify control-flow correctness. A process is sound if its control-flow guarantees proper completion and there are no deadlocks. However, a sound control flow is not sufficient to ensure that an executable process model indeed behaves as expected. This is due to business functions requiring certain preconditions to be fulfilled for execution and having an effect on the process (postconditions).

Semantic annotations provide a means for taking such further aspects 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 postconditions specified relative to an ontology that axiomatizes the underlying business domain. Verification then means to determine whether the interaction of control flow and logical states of the process is correct. To this end, we formalize the semantics of annotated processes and point out which kinds of flaws may arise. We then identify a class of processes with restricted semantic annotations where correctness can be verified in polynomial time; and we prove that the semantic annotations cannot be generalized without losing computational efficiency. We introduce our techniques and results at a semi-formal level using an illustrative example; we provide full technical and formal details in two appendices.

1 Introduction

Nowadays, verifying control-flow correctness is understood as an important step before deploying executable business process models as templates for handling individual process instances. In this context, the soundness criterion and its derivatives, e.g. [1,2,3,4], are typically used to check whether proper completion is possible or even guaranteed. Tools like Woflan [5] provide the functionality to efficiently verify soundness based on Petri nets theory. While soundness is indeed a necessary condition for correctness, it covers only the control-flow perspective of the process model. To assure that a process model behaves as expected, it is necessary to take further aspects into account. This is particularly important for Web service composition where third-party services assume preconditions to be true and have effects in terms of postconditions.
For instance, recent work on the composition of executable process models aims to support the designer in finding suitable service implementations based on semantic descriptions [6,7]. While this approach requires services to be semantically formalized in languages like OWL-S [8,9] or the Web Service Modeling Ontology (WSMO) [10,11], it enables several analysis options beyond control-flow verification. In particular, OWL-S and WSMO cover preconditions and postconditions, as well as ontological axiomatizations of the underlying domain. These constructs interfere with control-flow correctness in three ways: first, the state of the process determines which preconditions are true; second, the execution of a service governs which postconditions become effective, and as a result to which state the process changes; third, any state of the process is known to adhere to the domain axioms.

The identification of preconditions and postconditions as well as ontological axioms is, in particular, beneficial for the validation of process models since users and stakeholders can much easier express constraints of the domain than define control-flow [12]. The research problem in this context is the missing combination of ontology reasoning and control-flow analysis. We address this research gap and provide the following contributions. Firstly, we define operational semantics of a process modeling language that captures workflows annotated with preconditions, postconditions, and ontological axiomatizations; to do so, we draw on widely used notions of token passing from the workflow literature [13,14] and on widely accepted notions of logical updates from the AI actions and change community [15,16,17]. Secondly, we identify important correctness properties for such annotated workflows. Finally, we identify a particular class of annotated process models for which we are able to define polynomial-time analysis algorithms. This is of crucial importance since many verification techniques do not scale due to exponential complexity [18]. We prove that in several aspects the class cannot be generalized without losing computational efficiency. The analysis techniques are implemented in a tool.

The paper is organized as follows. Section 2 introduces a BPMN process model that we use as a running example; we illustrate execution problems that arise due to the interaction between control-flow and semantic annotation, and we describe the formalism that we use to reason about these problems. Section 3 explains the analysis techniques that we use to validate the semantically enriched process models and illustrates them using the results of our tool for the running example. Section 4 discusses related work, and Section 5 concludes the paper.

For the sake of readability, the main body of text is written at a semi-formal level, mainly discussing the BPMN running example. Appendix A contains a more technical presentation including formal definitions of our framework, and pseudo-code for our algorithms. Appendix B provides some additional technical details including full proofs.

2 Preliminaries

In this section we introduce our running example, a sales order process. In particular, Section 2.1 discusses the validation that is needed before deploying it. Then, Section 2.2 explains how we formalize the validation problem.
2.1 Motivating Example

We consider a sales order process that is inspired by the BPEL specification [19]. Fig. 1 shows this process in BPMN. The AND-gateways (symbol +) represent parallel execution and synchronization. The receipt of a sales order triggers three concurrent activities: initiating the production scheduling, deciding on the shipper, and drafting a price calculation. Once the shipper has been selected, the price calculation can be completed and the logistics can be arranged. After the latter, the production can be completed. Finally, the invoice is processed. The process model is obviously sound, i.e. proper completion is guaranteed, and in particular, there are no deadlocks. In the following we will take the perspective of a German machine producer (we call it GMP) that manufactures to order. As depicted in the model this involves that production, delivery, and pricing are tailored to the product requirements of the customer.

![Diagram of sales order process in BPMN](image)

Fig. 1. Example of a sales order process modeled in BPMN.

GMP has used this process successfully for years to produce and deliver to the national market. Now the company has made the decision to consider also international orders. Recently, it has received an order of 10 impulse turbines for power generation from a country that currently faces political unrest. According to German legislation such a delivery requires approval from the “Bundesamt für Wirtschaft und Ausfuhrkontrolle” (BAFA), i.e. the authority controlling German foreign trade. In order to deal with the complexity of international trade, GMP has decided to replace some of its production steps with services from experts. The key account manager for international clients has signed agreements with respective service providers. The new set of services including their preconditions and postconditions are summarized in the top part of Table 1. In particular, the Production Scheduling, the Production, and Arrange Logistics tasks are now provided by new services that have more specialized preconditions and postconditions. Further, a domain theory (bottom part of Table 1) has been added, capturing some simple domain constraints that can be easily validated by stakeholders. Finally, Table 1 shows the process variables that capture the state of the involved business objects: order \((o)\), production \((p)\), calculation \((c)\), shipper \((s)\), and shipment \((sh)\).
<table>
<thead>
<tr>
<th>#</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>orderReceived(o)</td>
<td>orderReceived(o)</td>
</tr>
<tr>
<td>2</td>
<td>orderReceived(o)</td>
<td>productionScheduled(o,p)</td>
</tr>
<tr>
<td>3</td>
<td>productionScheduled(o,p)</td>
<td>productionCompleted(o,p)</td>
</tr>
<tr>
<td></td>
<td>calculationPrepared(o,c)</td>
<td>calculationUpdated(o,c)</td>
</tr>
<tr>
<td>4</td>
<td>orderReceived(o)</td>
<td>shipperDecided(o,s)</td>
</tr>
<tr>
<td>5</td>
<td>shipperDecided(o,s)</td>
<td>calculationUpdated(o,c)</td>
</tr>
<tr>
<td></td>
<td>calculationPrepared(o,c)</td>
<td>shipmentApproved(o,sh)</td>
</tr>
<tr>
<td>6</td>
<td>orderReceived(o)</td>
<td>calculationDrafted(o,c)</td>
</tr>
<tr>
<td>7</td>
<td>calculationPrepared(o,c)</td>
<td>calculationCompleted(o,c)</td>
</tr>
<tr>
<td>8</td>
<td>productionCompleted(o,p)</td>
<td>orderCompleted(o)</td>
</tr>
<tr>
<td></td>
<td>calculationCompleted(o,c)</td>
<td>orderCompleted(o)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>o is at most one of received or completed</td>
</tr>
<tr>
<td>Production</td>
<td>p is at most one of scheduled or completed</td>
</tr>
<tr>
<td>Calculation</td>
<td>if c is drafted, then c is prepared</td>
</tr>
<tr>
<td></td>
<td>if c is updated, then c is prepared</td>
</tr>
<tr>
<td></td>
<td>c is at most one of drafted, updated, completed</td>
</tr>
<tr>
<td></td>
<td>c is at most one of prepared or completed</td>
</tr>
<tr>
<td>Shipment</td>
<td>if shipment sh is approved, then o is approved</td>
</tr>
</tbody>
</table>

Table 1. Semantic annotations for the sales order process activities. Top: preconditions and postconditions of the services (numbers referring to the task enumeration in Fig. 1). Bottom: ontology axioms.
In a discussion with the production engineer the key account manager is embarrassed to learn that his new set of services will not work for the production process of GMP, although the control flow of this process is sound. There are three different kinds of problems: executability problems, precondition conflicts, and effect conflicts.

**Executability:** Executability refers to a problem class where the execution of a service is not possible because its precondition is not necessarily true. In order to cover the BAFA export approval, GMP has chosen a shipper whose *Arrange Logistics* service provides a postcondition of *shipmentApproved* if BAFA approves the delivery. Furthermore, GMP selected a *Production Scheduling* service that requires *orderApproved* as a precondition in order to block production until it is clear that the ordered goods can be exported. This alone is fine since, by the axiomatization, an order is approved if its shipment is approved. However, there is now a dependency between arranging logistics and scheduling the production, which determines the order of these two activities. Logistics must be arranged before the production is scheduled. This is not done by the process, and hence the precondition of production scheduling is not fulfilled when trying to execute it. Further executability problems arise with the new services for *Arrange Logistics* and *Production*, because they require the calculation to be prepared, although *Draft Price Calculation* is not guaranteed to be executed beforehand. Also, *Complete Price Calculation* may be done in parallel, which causes problems due to a precondition conflict.

**Precondition conflict:** A precondition conflict exists if one of two concurrent tasks may negate the precondition of the other through its effect. The new *Arrange Logistics* task requires the calculation $c$ to be prepared (drafted or updated) so that it can be updated. However, *Complete Price Calculation* is not ordered with respect to *Arrange Logistics*, and if it is executed first then the status of $c$ changes to *completed*. In this case, the precondition of *Arrange Logistics* is no longer fulfilled: *calculationDrafted* and *calculationCompleted* exclude each other. The same conflict exists between *Production* and *Complete Price Calculation*.

**Effect conflict:** An effect conflict exists if two concurrent tasks overlap in their postcondition such that the execution of one of them overwrites the postcondition of the other. In this case, the final outcome depends on the execution order in a process instance. Such a conflict appears between *Arrange Logistics* and *Complete Price Calculation*, as well as *Production* and *Complete Price Calculation*. The new *Arrange Logistics* and *Production* services of GMP update the price calculation, as indicated by their effect *calculationUpdated*. On the other hand, the parallel task *Complete Price Calculation* establishes the conflicting postcondition *calculationCompleted*. Hence, whether or not the calculation is completed at the end of the process depends on which one of these nodes is executed last – a clearly undesirable behavior (which also causes *Invoice Processing* to not be executable).

---

3 Note that the new process should actually include exception handling for the case where BAFA does *not* approve the export. The shown process allows us to check whether the process is correct under the assumption that this circumstance beyond its control turns out fine.
In the future, GMP wants to be able to analyze such problems during the design phase of the process. Next, we describe how we formalize the behavior of annotated processes, so that reasoning techniques for this kind of analysis become applicable.

2.2 Formalization of Process Behavior

For the workflow part of annotated processes, we assume straightforward execution semantics based on token-passing similar to Petri Nets; in particular, we adapt the definitions from [14]. We extend those with a notion from AI to deal with semantic annotations and their meaning.

We define a process model as a \textit{process graph} with nodes of various types – a single start and end node, task nodes, XOR split/join nodes, and parallel split/join nodes – and directed edges (expressing execution order). The number of incoming (outgoing) edges are restricted as follows: start node 0 (1), end node 1 (0), task node 1 (1), split node 1 (>1), and join node >1 (1). The location of all tokens, referred to as a \textit{marking}, manifests the state of a process execution. An execution of the process starts with a token on the outgoing edge of the start node and no other tokens in the process. 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 a XOR (Parallel) split node consumes the token on its incoming edge and produces a token on one (all) of its outgoing edges, whereas a XOR (Parallel) join node consumes a token on one (all) of its incoming edges and produces a token on its outgoing edge. We assume that a process is sound, i.e., it always completes with one token on the incoming edge of the end node and no tokens elsewhere [1]. Since we use semantics similar to free-choice nets, this also implies that the process is safe, i.e., there is never more than one token on an edge [1].

The semantic annotations are made with respect to a background ontology \( O \) consisting of two parts: the vocabulary as a set of predicates \( P \), and a logical theory \( T \) as a collection of first order formulae over these predicates. Further, there is a set of process variables (\( o, p, c, s, sh \) in the example) over which logical statements can be defined in the form of literals involving these variables. All the mentioned sets are finite. Intuitively, the logical theory is like a rule base stating, e.g., that the approval of a shipment implies that the respective order is also approved or that no calculation can be in two different states simultaneously (such as being both completed and drafted). These rules can be applied to the concrete process variables, i.e., this particular shipment and order, or this particular calculation. \footnote{More formally, the quantifiers in \( T \) are taken to range over the set of process variables, i.e., over the objects actually manipulated by the process. Since this universe is finite, in particular reasoning is decidable. Exploring different interpretations of the quantifiers is an open topic.}

Further, each task node \( n \) can be annotated using \textit{preconditions} (\( \text{pre}_n \)) and \textit{effects} (\( \text{eff}_n \), also referred to as \textit{postconditions}), which are conjunctions of literals using the process variables. The task can only be executed if \( \text{pre}_n \) is true; \( n \) is \textit{executable} if that is the case whenever \( n \)'s incoming edge carries a token, i.e., whenever the control flow reaches \( n \); the process is executable if all its tasks are. If executed, \( n \) changes the state of the world (i.e., the state reached by the process in its execution) according to its postcondition. The postcondition states the explicit effects. Depending on the current state and the axioms in the theory, the task may also have
implicit effects. For example, the Complete Price Calculation activity in the example completes the calculation as its explicit effect; by the theory of the example (because the calculation must have a unique status), as implicit effects we get that the calculation is neither drafted nor updated.

It is important to note that implicit effects are not always as easy to determine as in this example. In particular, the conclusions to be made from \( T \) may be ambiguous, so that several possible outcomes must be taken into consideration. The issue of implicit effects – the question how explicit effects do or do not change the state of the world, in the presence of a domain axiomatization – has been extensively investigated in AI, under the name “frame and ramification problems”. We follow a widely adopted semantics based on a notion of “minimal” change [15]. This is best understood as a kind of local stability: the world does not change of its own accord; properties that were true before are still true unless there is a reason to change them. A little more formally, an outcome state is taken to be possible if there is no other outcome state that makes strictly less changes to the previous state.

Let us illustrate the formalization using our GMP sales order process example. The ontology includes those predicates that are listed as preconditions and postconditions in Table 1, namely \( P = \{\text{orderReceived}(x), \text{orderApproved}(x), \ldots, \text{shipmentApproved}(x, y)\} \). The theory \( T \) consists of the formulae shown in Table 2, formalizing their intuitive counterparts from Table 1.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order status</td>
<td>( \forall x : \neg \text{orderReceived}(x) \lor \neg \text{orderCompleted}(x) )</td>
</tr>
<tr>
<td>Production status</td>
<td>( \forall x, y : \neg \text{productionScheduled}(x, y) \lor \neg \text{productionCompleted}(x, y) )</td>
</tr>
<tr>
<td>Calculation status</td>
<td>( \forall x, y : \text{calculationDrafted}(x, y) \implies \text{calculationPrepared}(x, y) )  [ \forall x, y : \text{calculationUpdated}(x, y) \implies \text{calculationPrepared}(x, y) ]  [ \forall x, y : \neg \text{calculationDrafted}(x, y) \lor \neg \text{calculationCompleted}(x, y) ]  [ \forall x, y : \neg \text{calculationUpdated}(x, y) \lor \neg \text{calculationCompleted}(x, y) ]  [ \forall x, y : \neg \text{calculationPrepared}(x, y) \lor \neg \text{calculationCompleted}(x, y) ]</td>
</tr>
<tr>
<td>Order approval</td>
<td>( \forall x, y : \text{shipmentApproved}(x, y) \implies \text{orderApproved}(x) )</td>
</tr>
</tbody>
</table>

Table 2. Formalization of the ontology axioms in Table 1.

Now, we can formally discuss the behavior of the process. At the start node of the process, Receive Order, the explicit effect is \( \text{orderReceived}(o) \). By \( T \), this has the implicit effect that \( \text{orderCompleted}(o) \) is false. Apart from that, any state of the world is considered possible. Say the process execution next performs the steps Draft Price Calculation and Decide on Shipper. Both are applicable in all possible worlds because their precondition, \( \text{orderReceived}(o) \), is definitely true. Their effects are \( \text{calculationDrafted}(o, c) \), which implies \( \text{calculationPrepared}(o, c) \), \( \neg \text{calculationUpdated}(o, c) \), and \( \neg \text{calculationCompleted}(o, c) \); and \( \text{shipperDecided}(o, s) \), which has no other implications via \( T \). As a result, we may be in any state of the world that complies with \( T \), and where all the mentioned explicit and implicit effect literals are true.
Say we decide to next execute the Production Scheduling activity, whose preconditions are $\text{orderReceived}(o)$ and $\text{orderApproved}(o)$. The former is certainly true, in all the worlds possible at this point. However, the order is not necessarily approved – there are possible worlds in which $\text{orderApproved}(o)$ is false. In other words, at the formal level we see that no assumptions can be made regarding whether or not the order is approved, and hence we conclude that Production Scheduling may not be able to execute. Note that, if Receive Order explicitly stated in its effect that the order is initially not approved, then we would even conclude that Production Scheduling cannot execute in any possible world. From the perspective of whether or not the process is correct, both cases are bad since we need the process to guarantee that Production Scheduling will always be able to execute. On the other hand, it may of course be important to distinguish these two cases. As we will see later, our analysis methods provide this functionality.

Say we decide differently, and execute the Arrange Logistics activity instead of Production Scheduling. This activity is guaranteed to be applicable since its preconditions, $\text{calculationPrepared}(o,c)$ and $\text{shipperDecided}(o,s)$, are true in all possible worlds. After the execution, due to the explicit effects all worlds will satisfy $\text{calculationUpdated}(o,c)$ and $\text{shipmentApproved}(o,sh)$. By $T$, via the axiom $\forall x, y : \text{shipmentApproved}(x,y) \implies \text{orderApproved}(x)$, the latter involves the implicit effect $\text{orderApproved}(o)$. Hence the order is now certain to be approved, and we can safely execute Production Scheduling.

The precondition and effect conflicts in the running example manifest themselves in a straightforward way; the only subtle point is that the conflicts concern implicit, not explicit, effects. Consider the Production and Complete Price Calculation activities. The former has the explicit effect $\text{calculationUpdated}(o,c)$, the latter has the explicit effect $\text{calculationCompleted}(o,c)$. These effects are not in immediate conflict; however, $T$ tells us that $\forall x, y : \lnot \text{calculationUpdated}(x,y) \lor \lnot \text{calculationCompleted}(x,y)$, implying that an implicit effect of Production is $\lnot \text{calculationCompleted}(o,c)$, the negation of the effect of Complete Price Calculation. Hence, we detect the conflict.

Notably, the $T$ formulae in our example are fairly restricted, each being equivalent to the (universally quantified) disjunction of only 2 literals. This is not coincidental:

**Theorem 1.** Assume an annotated process graph. Deciding whether the process is executable is $\Pi_2^P$-hard for unrestricted $T$, and $\text{coNP}$-hard if $T$ consists of Horn clauses only. This holds even if predicate arity is fixed to 0.

Theorem 1 results from the fact that, with more general formulae – even with Horn clauses where implication can be decided in polynomial time – it is computationally hard to reason about the implicit effects of activities. This is a consequence of earlier results in AI [16]. Further, we considered the possibility to annotate conditions at the outgoing edges of XOR splits, i.e., case distinctions directing the execution depending on runtime conditions. However:

**Theorem 2.** Assume an annotated process graph with case distinctions. Deciding whether the process is executable is $\text{coNP}$-hard. This holds even if predicate arity is fixed to 0, and $T$ is empty.

---

5 Predicate arity refers to the number of arguments any predicate may have. When allowing arbitrarily high arity is not realistic since predicates usually have few arguments – the complexity is likely to be even worse.
This can be proved by a reduction from SAT. Hence, both complex $T$ and case distinctions must be dealt with by worst-case exponential analysis methods (unless $P=NP$). Such methods might still be practically feasible, provided the processes do not grow too large; however, given this complexity it is clearly important to look for classes of processes and annotations where analysis is easier. Herein, we explore what we call basic process graphs – e.g., the discussed sales order process graph is basic.

3 Polynomaial-Time Analysis

In basic processes, the formulae in $T$ are restricted to universally quantified disjunctions of at most 2 literals; we do not allow case distinctions (annotated XOR splits), and we do not allow loops. Theorems 1 and 2 show that basic processes are maximally general – one cannot generalize them without losing computational efficiency – regarding the formulae in $T$ and the case distinctions. It is yet an open question whether loops, or structured loops, can be dealt with efficiently; we are currently investigating this. Regarding the restriction on $T$, note that universally quantified disjunctions of at most 2 literals have significant modelling power, and allow us to formulate common things such as the subsumption relations and mutual exclusions used in our running example.

Our analysis method works in two steps. First, it is determined which pairs of activities $n, n'$ in the process are parallel, i.e., have no ordering constraint between them (there are execution paths that do $n$ before $n'$, and there are execution paths that do $n'$ before $n$). Based on this information and the precondition/effect annotations, it is easy to detect precondition and effect conflicts. Once such conflicts have been removed, the second step of the analysis determines whether the activities are executable. We explain the two steps of the analysis in one sub-section each.

Before we start, there is a technical remark to be made regarding deduction in $T$ as present in basic processes. If predicate arity is fixed,\footnote{This means that, quite sensibly, predicates are not allowed to have arbitrarily many arguments.} then $T$ can be put into propositional format, by instantiating the quantifiers with all possible variables, in polynomial time. Further, it is well known that reasoning over 2-clauses, i.e., over propositional disjunctions of at most 2 literals, is polynomial \cite{20}. Hence, given the effect $\text{eff}_n$ of a task node $n$, we can easily determine all literals $l$ that are implied by $\text{eff}_n$ in conjunction with $T$. This ability is important in both steps of the analysis method. We denote the “extended effect”, i.e., the union of $\text{eff}_n$ with its implied literals $l$, by $\text{eff}_n$.

3.1 Precondition and Effect Conflicts

In order to detect precondition and effect conflicts – both of which exist only between parallel tasks – we first need to find out which pairs of tasks actually are parallel. We define an algorithm, called $M$-propagation, which determines this. Since parallelism is not affected by semantic annotations, those need not be considered at this stage. The algorithm populates a matrix $M$ whose rows and columns correspond to the edges of the process and whose entries are Boolean. $M$ contains a 1 in the $i^{th}$ row and $j^{th}$ column, denoted $M_{ij}$, iff $i \neq j$ and $e_i$ and $e_j$ may hold a control flow token at the same time; we...
say in this case that \( e_i \) and \( e_j \) are parallel. Parallelism between task nodes can then be checked simply by verifying whether their incoming edges are parallel. (Note that, by its nature, \( M \) is irreflexive and symmetric.)

The algorithm assumes a numbering of the edges. The numbering must be order-preserving in the sense that, if an edge \( e_i \) always executes before some other edge \( e_j \), then \( i < j \). Appendices A and B show how to generate such a numbering; they also contain full details on the M-propagation algorithm. Thanks to being able to order edges in this way, we do not need to calculate the reachability graph explicitly which would be exponential [18]. Fig. 2 illustrates the outcome of the algorithm on our sales order process example.

\[
\begin{array}{cccccccccc}
\text{Receive Order} & \text{Production Scheduling} & \text{Decide on Shipper} & \text{Draft Price Calculation} & \text{Arrange Logistics} & \text{Production} & \text{Complete Price Calculation} & \text{Invoice Processing} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Fig. 2. The matrix \( M \) for our running example, projected to input edges of task nodes.

M-propagation works by propagating parallelism information over the nodes in the process graph. \( M \) is initialized with 0 on the first diagonal (i.e., the fields \( M_i^i \)), and with the \( \perp \) symbol in all other fields, marking all these values to be as yet unknown. The propagation then commences at the start node, which is receive order in our example. Please note here that our processes have a single start node and that this node is not parallel to any other nodes. We perform propagation steps in an order following the edge numberings, making sure that we only propagate over nodes \( n \) whose incoming edges have already been considered (their \( M \)-values have been determined) and whose outgoing edges have not yet been considered. Each propagation step updates the matrix \( M \) in a “global” sense, i.e., a single matrix \( M \) is maintained and updated by every propagation step. The updates depend on the type of the node \( n \) considered:
Task nodes: For such nodes, we copy the $M$-values up to the number of the outgoing edge from the incoming edge. This works because a task node neither synchronizes nor splits the control flow, and thus has no effect on parallelism; $n$’s outgoing edge is parallel to the same edges as $n$’s incoming edge.

Parallel splits: Here, there are two things to consider. First, similarly as for task nodes, parallelism from the ingoing edge is preserved in the outgoing edges. To account for this, we copy the respective $M$-values: if $e_i$ is the incoming edge and $e_j$ is the outgoing edge with the lowest number, then we set $M^k_i = M^k_j$ for all $k < j$. Second, the outgoing edges of a parallel split introduce new parallelism. This is covered simply by setting $M^j_i = 1$ for all outgoing edges $e_i$ and $e_j$ of $n$, $i \neq j$. In Fig. 2 this can be observed at the first parallel split node: Production Scheduling, Decide on Shipper, and Draft Price Calculation are pairwise parallel.

XOR splits: These are handled exactly as parallel splits, except that we set $M^j_i = 0$ for all outgoing edges $e_i$ and $e_j$ of $n$. Obviously, this reflects the fact that the outgoing edges of an XOR split can never carry a token at the same time (since we assume the workflow to be safe and sound).

Parallel joins Here, matters are slightly more tricky. Say $e_i$ is the outgoing edge. For any $j$ with $j < i$, we set $M^j_i$ to 1 iff there is no incoming edge $e_k$ with $M^k_j = 0$. This is necessary because parallel joins synchronize branches that were previously parallel. Thus, if one of the incoming edges is already synchronized with $e_j$, then the parallel join will transfer this synchronization to $e_i$ as well. This can be observed in Fig. 2 at the parallel join after Production Scheduling. Production Scheduling is parallel to Decide on Shipper and Arrange Logistics, but the latter two are synchronized (i.e., not parallel to one another). Therefore, Production, coming after the parallel join, is not parallel to Decide on Shipper.

XOR joins For this node type, we perform an index-wise logical OR: say $e_i$ is the outgoing edge again; for any $j$ with $j < i$, we set $M^j_i$ to 1 iff an incoming edge $e_k$ exists, with $M^k_j = 1$. This is necessary, because a single incoming edge $e_k$ which can carry a token at the same time as $e_j$ can pass its token to the outgoing edge $e_i$ at any time. Note that in a sound process graph, no two incoming edges of a XOR join may carry a token at the same time.

The propagation ends when $M$ has been determined for the incoming edge of the end node; note that, by definition, this is the edge with the highest number. We have:

Lemma 1. Assume a basic annotated process graph. Then the time taken by $M$-propagation is polynomial in the size of the graph. Assume $M$ is its outcome. Then, for all pairs of task nodes $n_i$, $n_j$ with incoming edges $e_i$, $e_j$: $n_i$ and $n_j$ are parallel iff $M^j_i = 1$.

Given the parallelism information, it is easy to determine any precondition and effect conflict. The following corollary is a consequence of Lemma 1, the definition of precondition/effect conflicts, and the aforementioned fact that reasoning over 2-clauses is polynomial (recall that $\text{eff}$ is the union of $\text{eff}$ with its implications over $T$):

Corollary 1. Assume a basic annotated process graph; assume $M$ is the outcome of $M$-propagation. Then, for all pairs of task nodes $n_i$, $n_j$ with incoming edges $e_i$, $e_j$: $n_i$ and $n_j$ are parallel iff $M^j_i = 1$. Therefore, $n_i$ and $n_j$ are parallel.
and \( n' \) have a precondition (effect) conflict iff \( M^j_i = 1 \) and there exists a literal \( l \) s.t. \( l \in \text{eff}_n \) and \( \neg l \in \text{pre}_{n'} \) \((\neg l \in \text{eff}_{n'}))\). With fixed predicate arity, all conflicts can be found in time polynomial in the size of the graph.

### 3.2 Detecting Non-Executable Activities

Our algorithm for detecting non-executable activities, which we call \( I\)-propagation, currently assumes that no effect conflicts are present in the process. This is not a critical assumption because effect conflicts are errors, and all errors should be removed from the process prior to execution. Absence of effect conflicts can be established by identifying effect conflicts as per Corollary 1, and pointing them out to the process modeler for removal. In our example process from Fig. 1, all effect conflicts can be removed by re-scheduling \textit{Complete Price Calculation} to come after \textit{Production}, i.e., as a second last step just before \textit{Invoice Processing}.\(^7\)

In what follows, we explain \( I\)-propagation at a semi-formal level; as mentioned before, details can be looked up in Appendices A and B. \( I\)-propagation keeps track of sets \( I(e) \) of literals, which are maintained individually for every edge \( e \). The key insight is that, in order to detect non-executable nodes, i.e., nodes \( n \) whose precondition is falsified in at least one execution, it suffices to know a summary of all possible worlds that may be encountered whenever \( n \) is activated. Namely, all we need to know is the set of literals \(- I(e) \) that are necessarily true whenever \( n \)’s incoming edge \( e \) carries a token. Intuitively, \( I(e) \) corresponds to the intersection of the worlds at \( e \). The ability to check executability based on such world-intersections is quite advantageous; it gets us around enumerating all the possible worlds, which would of course be exponentially costly.

Fig. 3 shows the outcome of \( I\)-propagation on part of our example sales order process. Similarly to \( M\)-propagation, \( I\)-propagation performs consecutive propagation steps over nodes \( n \) of the process; in difference to \( M\)-propagation, the modifications are “local”, i.e., as indicated every edge \( e \) has its own \( I(e) \) set. Each propagation step updates the annotated \( I(e) \) sets according to the type of \( n \). Initially, \( I(e) \) is set to \( \text{eff}_{n_0} \) for the start node \( n_0 \), and to \( \bot \) for all other nodes. A node is propagated only if all its incoming edges have been considered, and all its outgoing edges have not yet been considered. The propagation steps, over nodes \( n \), are (we start with the simple ones):

**Splits:** If \( n \) is a parallel split or an XOR split, the propagation simply copies \( I \) from the incoming edge to every outgoing edge. This is because splits do not change the state of the world. See the two parallel splits in Fig. 3 (behind \textit{Receive Order} and behind \textit{Decide on Shipper}) for illustration.

**Parallel joins:** Say \( e' \) is \( n \)’s outgoing edge; we set \( I(e') \) to the union of the sets \( I(e) \) for all of \( n \)’s incoming edges \( e \). This is justified per the assumed absence of 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 \( l \)

\(^7\) If, for some reason, the modeler wishes to consider executability regardless of effect conflicts, then the executability analysis must be able to deal with those. It is not clear whether such an analysis can still be done in polynomial time; we are currently investigating this.
Fig. 3. Outcome of I-propagation on a part of the example process from Fig. 1.

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. 3 (behind Production Scheduling and Arrange Logistics) for illustration: the \( I \) sets of the 2 incoming edges are combined.

**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).

**Task nodes:** These are by far the most complicated propagation steps. Say \( n \) has the incoming edge \( e \) and the outgoing edge \( e' \). Three different actions need to be performed. (1) We write \( \text{eff}_n \), i.e., \( n \)'s explicit and implicit effects, 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 \( \text{eff}_n \).

(1) and (2) are direct consequences of the semantics of annotated task nodes, c.f. Section 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.

It is important to note here that, actually, (1) and (2) can be done in such a simple way only because \( T \) is restricted to disjunctions of at most 2 literals. As pointed out by Theorem 1, minimal change semantics get quite intricate with more complex \( T \). For example, consider this situation: \( T \) contains a single disjunction, of the three literals \( \neg p, \neg q, \neg r; p, q \in I(e); \text{eff}_n = \{r\} \). Then, after \( n \), neither \( p \) nor \( q \) are guaranteed to hold (although their opposites are not contained in \( \text{eff}_n \)). The reason is that, with \( p \) and \( q \) being already true, the effect \( r \) falsifies the disjunction. There
are several possible ways to “repair” this, namely by either falsifying $p$ or $q$; hence after $n$ any of the literals $p$, $q$, $\neg p$, $\neg q$ may be true. At an intuitive level, situations like this (and other more complicated situations) cannot appear when $T$ consists of 2-clauses only; hence for basic process graphs actions (1) and (2) are suffice.

Let us finally consider action (3), dealing with the case where an edge $e''$ parallel to $e'$ inherited a literal $l$ which is in conflict with $\text{eff}_n(l)$ (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 Production. 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 elsewhere. Say $n'$ fires, putting a token on both of the edges. In this situation, due to the effect of Production Scheduling which must have been executed beforehand, we know that $\text{productionScheduled}(o,p)$ and $\neg \text{productionCompleted}(o,p)$ are certain to hold. Accordingly, I-propagation over $n'$ (as explained above) puts these literals into $I(e'')$. However, say $n$ fires next. Then $e''$ still carries a token, but both literals have been inverted. Hence, when $e''$ carries a token, $\text{productionScheduled}(o,p)$ and $\neg \text{productionCompleted}(o,p)$ are not always true. They should be removed from $I(e'')$, which is exactly what action (3) does; the annotation of $e''$ in Fig. 4 shows the outcome.

The propagation ends when $I$ has been determined for the incoming edge of the end node; note that, by definition, this is the last propagation step possible. We have:

**Theorem 3.** Assume a basic annotated process graph without effect conflicts. With fixed predicate arity, the time taken by I-propagation is polynomial in the size of the graph. Assume $I$ is its outcome. Then the process is executable iff, for all task nodes $n$ with incoming edge $e$, $\text{pre}_n \subseteq I(e)$.

Recall here that a process is executable iff all its tasks are (c.f. Section 2.2). The theorem is proved in three steps. First, Lemma 1 shows that the $M$ information exploited
in task node propagations is correct. Second, denote by $\bigcap e$ the set of literals that will always be true when $e$ carries a token. The arguments made above regarding the propagation steps show that $I(e) = \bigcap e$ for all $e$ – i.e. they show that the $I(e)$ sets are correct – when assuming that all task nodes are executable (note that I-propagation ignores preconditions). Third, Theorem 3 now follows with the following trick. If all nodes are executable (as assumed), then, since $\bigcap(e) = I(e)$ for all $e$, we have $\text{pre}_n \subseteq I(e)$ for all task nodes $n$ with incoming edge $e$. Conversely, say $n$ is a task node where $\text{pre}_n \not\subseteq I(e)$, but $\text{pre}_{n'} \subseteq I(e)$ for all of $n$'s predecessors in the graph. Then all predecessors are executable and we know, with the same arguments as before, that $I(e) = \bigcap e$. Hence $n$ cannot be executable. This concludes the argument.

Theorem 3 can be used for executability checking in the obvious way. If $\text{pre}_n \subseteq I(e)$ for all task nodes $n$, we know that the process is correct and we can stop. Else, we can indicate to the user all task nodes $n$ where $\text{pre}_n \not\subseteq I(e)$ but where all predecessors $n'$ have $\text{pre}_{n'} \subseteq I(e)$. These nodes are not executable. Fixing these flaws, the modeler can run I-propagation again and obtain the next “frontier” of non-executable nodes, and so on until all flaws are removed.

Our analysis methods are implemented in Java. The implementation as yet lacks the connection to an interface for specifying the process to be analyzed (i.e., the process is inserted directly into the internal data structures); hence a broad empirical evaluation has not yet been performed. In our running example, the analysis successfully completes within less than a second. Note that, given the low-order polynomial complexity of our algorithms, runtime performance is very likely never going to be an issue.

4 Related Work

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 [13]. There are some contributions beyond pure control-flow verification. In particular, they can be related to semantic checks and data flow analysis.

The approach of [21] 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 [22]. Our approach uses not only annotations in terms of preconditions and effects but also an ontology. In that sense, [21] might be regarded as a special case of our framework. In the area of access control the approach of [23] extends process models with predicates, constants, and variables. However, 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 [24] describes methods to check compliance of a process against rules for role assignment. This is related to our approach in that a theory could (to some extent) be defined to model such rules; but not vice versa since we build on general logics while [24] covers some practical special cases. The paper by [25] addresses a.o. life cycle compliance (essentially whether the process model does not violate the constraints expressed in the life cycles). 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.

In [26], 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 over constants, and the processes are assumed to be sound, acyclic, have a single start and end node, respectively, and the routing constructs are and/xor join/split. However, [26] neither deals with ontological axiomatizations nor with initial state uncertainty. 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 stark contrast to our algorithm which takes polynomial time.

Another exponential-time check is discussed in [27] where semantic web service compositions are completely encoded in Petri Nets, i.e., both the control flow and the preconditions and effects are mapped to states, transitions, and arcs. However, since literals may be used at multiple points in a process, the resulting nets are not free-choice. Further, the choice of annotation restricts the relation between preconditions and effects since precondition tokens are always consumed. The verification properties are then formulated as standard Petri Net properties: reachability, liveness, deadlock-freeness. There is an overlap between these properties and ours, e.g., liveness requires that, for each service, there is an execution sequence in which its precondition is fulfilled whereas our executability requires all token-executions to fulfill the preconditions encountered. Also [27] does not consider ontologies.

The most closely related work to our approach is [28]. 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. [28] 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) which is in stark difference to our polynomial time methods. Further, [28] do not consider preconditions, and they do not consider logical theories constraining the domain behavior. Finally, while we provide a formal execution semantics and prove our methods correct relative to that semantics, the work of [28] proceeds at a rather informal intuitive level.

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 [29]. Data flow analysis builds on compiler theory [30] where data flows are typically examined for sequential programs mostly; it does neither consider theories $T$ nor logical conflicts, and hence explores a direction complementary to ours. Our concepts can be applied in this area by expressing data dependencies as preconditions, effects, and ontological axioms.
5 Conclusion

We introduced a formalism for checking certain correctness properties of semantically annotated process models. It is unique in that it combines notions from the workflow community and from the AI literature resulting in an integrated execution semantics. We showed that this formalism can detect execution problems in annotated process graphs with sound control flow. This is an important contribution with practical implications for the verification of executable process models. Our work identifies a special class of annotated process graphs for which correctness can be checked efficiently, and we provide the algorithms for doing so. We show that the annotations cannot get more complex without introducing computationally hard problems. The contributions of this work aim at the verification beyond soundness for executable process models, e.g., in the form of Web service orchestrations. We assume that this additional verification will lead to fewer errors and a shorter time span for design and deployment of executable process models.

In our next steps we aim to enhance our analysis techniques to deal with loops and to check executability without prior removal of effect conflicts. Yet, it is not clear whether this is possible in polynomial time. In the long term, also computationally hard cases should be addressed. Such analysis techniques will require some form of combinatorial search, and will be of a different nature.

References

A  Formal Presentation

Herein we provide a more formal presentation of our techniques. In particular we provide formal definitions of our framework, and pseudo-code for our algorithms. To make this more readable, a few technical details are moved to Appendix B; also, instead of proofs we give easy-to-read proof sketches, while full proofs are in Appendix B.

We start by defining our framework (Section A.1); then we present our analysis algorithms for the class of basic processes (Section A.2); then we discuss our results on the computational borderline of that class of processes (Section A.3).

A.1  Semantic Business Processes

We introduce our formalism, i.e., our formal execution semantics, for business processes whose tasks are annotated with logical preconditions and effects (Section A.1). We then formally introduce the correctness criteria that we consider (Section A.1).

Annotated Process Graphs  For the sake of readability, we first introduce non-annotated process graphs. This part of our definition is, without any modification, adopted from the workflow literature, following closely the terminology and notation used in [14].

Definition 1. A process graph is a directed graph $G = (N, E)$, 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: for each split node $n$, $|IN(n)| = 1$ and $|OUT(n)| > 1$; for each join node $n$, $|IN(n)| > 1$ and $|OUT(n)| = 1$; for each $n \in N_T$, $|IN(n)| = 1$ and $|OUT(n)| = 1$; for $n_0$, $|IN(n)| = 0$ and $|OUT(n)| = 1$ and vice versa for $n_+$; each node $n \in N$ is on a path from the start to the stop node. If $|IN(n)| = 1$ we identify $IN(n)$ with its single element, and similarly for $OUT(n)$; we denote $OUT(n_0) = e_0$ and $IN(n_+) = e_+$.

The intuitive meaning of these structures should be clear: an execution of the process starts at $n_0$ and ends at $n_+$; a task node is an atomic action executed by the process; parallel splits open parallel parts of the process; xor splits open alternative parts of the process; joins re-unite parallel/alternative branches. The stated requirements are just basic sanity checks any violation of which is an obviously flawed process model.

Formally, the semantics of process graphs is, similarly to Petri Nets, defined as a token game. A state of the process is represented by tokens on the graph edges. Like parallel splits open parallel parts of the process; xor splits open alternative parts of the process; joins re-unite parallel/alternative branches. The stated requirements are just basic sanity checks any violation of which is an obviously flawed process model.

Definition 2. Let $G = (N, E)$ be a process graph. A state $t$ of $G$ is a function $t : E \rightarrow N$; we call $t$ a token mapping. The start state $t_0$ is $t_0(e) = 1$ if $e = e_0$, $t_0(e) = 0$ otherwise. Let $t$ and $t'$ be states. We say that there is a transition from $t$ to $t'$ via $n$, written $t \rightarrow^n t'$, iff one of the following holds:

1. $n \in N_T \cup N_{PS} \cup N_{PJ}$ and $t'(e) = t(e) - 1$ if $e \in IN(n)$, $t'(e) = t(e) + 1$ if $e \in OUT(n)$, $t'(e) = t(e)$ otherwise.
2. $n \in \mathcal{N}_{XS}$ and there exists $e' \in \text{OUT}(n)$ such that $t'(e) = t(e) - 1$ if $e = \text{IN}(n)$.

$t'(e) = t(e) + 1$ if $e = e'$, $t'(e) = t(e)$ otherwise.

3. $n \in \mathcal{N}_{XJ}$ and there exists $e' \in \text{IN}(n)$ such that $t'(e) = t(e) - 1$ if $e = e'$.

$t'(e) = t(e) + 1$ if $e = \text{OUT}(n)$, $t'(e) = t(e)$ otherwise.

An execution path is a transition sequence starting in $t_0$. A state $t$ is reachable if there exists an execution path ending in $t$.

Definition 2 is straightforward: $t(e)$, at any point in time, gives the number of tokens currently at $e$. Task nodes and parallel splits/joins just take the tokens from their IN edges, and move them to their OUT edges; xor splits select one of their OUT edges; xor joins select one of their IN edges. For the remainder of this paper, we will assume that the process graph is sound: from every reachable state $t$, a state $t'$ can be reached so that $t'(e_+) > 0$; for every reachable state $t$, $t(e_+) \leq 1$. This means that the process does not contain deadlocks, and that each completion of a run is a proper termination, with no tokens remaining inside the process. These properties can be ensured using standard workflow validation techniques, e.g., [13,14].

For the semantic annotations, we use standard notions from logics, involving logical predicates and constants (the latter correspond to the entities of interest at process execution time). We denote predicates with $G, H, I$ and constants with $c, d, e$. Facts are predicates grounded with constants, Literals are possibly negated facts. A clause is a conjunction of Horn (binary) clauses. Note that binary clauses can be used to specify many common ontology properties such as subsumption relations $\forall x. G(x) \Rightarrow H(x)$ ($\phi \Rightarrow \psi$ abbreviates $\neg \phi \vee \psi$), attribute image type restrictions $\forall x, y. G(x, y) \Rightarrow H(y)$, and role symmetry $\forall x, y. G(x, y) \Rightarrow G(y, x)$. An example of a property that is Horn (but not binary) is role transitivity, $\forall x, y, z. G(x, y) \land G(y, z) \Rightarrow G(x, z)$.

An ontology $O$ is a pair $(P, T)$ where $P$ is a set of predicates ($O$’s formal terminology) and $T$ is a theory over $P$ (constraining the behaviour of the application domain encoded by $O$). Annotated process graphs are defined as follows.

Definition 3. An annotated process graph is a tuple $G = (\mathcal{N}, \mathcal{E}, O, A)$. $\mathcal{N}$ is a process graph, $O = (P, T)$ is an ontology, and $A$, the annotation, is a partial function mapping $n \in \mathcal{N}_T \cup \{n_0, n_{+}\}$ to $(\operatorname{pre}(n), \operatorname{eff}(n))$ where $\operatorname{pre}(n), \operatorname{eff}(n) \subseteq P$, and mapping $e \in \text{OUT}(n)$ for $n \in \mathcal{N}_{XS}$ to $(\operatorname{con}(e), \operatorname{pos}(e))$, where $\operatorname{con}(e) \subseteq P$.
pos(e) ∈ {1, ..., |OUT(n)|}. We require that: there does not exist an n so that T ∧ eff(n) is unsatisfiable or T ∧ pre(n) is unsatisfiable; there does not exist an e so that T ∧ con(e) is unsatisfiable; there do not exist n, e, and e’ so that e, e’ ∈ OUT(n), A(e) and A(e’) are defined, and pos(e) = pos(e’).

We refer to cycles in (N, E) as loops; we refer to edges e for which A(e) is defined as case distinctions (sometimes, process graphs without loops and/or case distinctions will be of interest). We refer to pre(n) as n’s precondition, eff(n) as n’s effect (sometimes called postcondition in the literature), con(e) as e’s condition, and pos(e) as e’s position. The annotation of tasks – atomic actions that will on the IT level correspond to Web service executions – in terms of logical preconditions and effects closely follows Semantic Web service approaches such as OWL-S (e.g. [8,32]) and WSMO (e.g. [11]). All the involved sets of literals (pre(n), eff(n), con(e)) are interpreted as conjunctions.\footnote{It is easy to extend our formalism to allow arbitrary formulas for pre(n), eff(n), con(e); extending the actual analysis leads to harder decision problems, and remains future work.}

Similarly to Definition 1, the requirements stated in Definition 3 are just basic sanity checks. If a precondition/effect/condition contradicts the theory, then the respective task node/edge will never be applicable. The requirement on edge positions is a minor technical detail, stating that no two edges are assigned the same position. This closely corresponds to standard process languages such as BPEL [31], where the positions are assigned based on the order of appearance in the input file. This ensures that the outcome of an xor split is deterministic – if the xor split edges are annotated. Note here that Definition 3 allows A to be a partial function. That is, an arbitrary subset of the process may be not annotated. The rationale behind this is that analysis will take place during modelling. In such a context, it is useful to allow analysis of partially modelled – partially annotated – processes.

Example 1. Consider the annotated process graph depicted in Figure 5 (using the slightly extended BPNM notation from [33]). In short, data objects depict the entities of interest, and associations link them to activities. Preconditions and effects are displayed as text on the associations, where the preconditions are denoted subsequent to “<” and the effects after “>”.

![Fig. 5. Basic example of a semantic process model (extended BPMN diagram).](image-url)
In terms of the formal notations from Definition 3, this process graph is defined as follows (by the number of "\(\cdot\)" symbols in the definition of logical predicates we indicate their arity):

\[ P := \{ \text{PurchaseOrder}(\cdot), \text{isCancelled}(\cdot), \text{isSent}(\cdot) \} \]

\[ C := \{ PO \} \]

\[ T := \emptyset \]

\[ N_T := \{ n_1, n_3 \} \]

\[ N_{XS} := \{ n_2 \} \]

\[ N_{XJ} := \{ n_4 \} \]

\[ E := \{(n_0, n_1), (n_1, n_2), (n_2, n_3), (n_2, n_4), (n_4, n_+)\} \]

The annotation function is given by the following:

\[ n_1 (\text{"Send PO"):} \]

\[ \text{pre}(n_1) := \{ \text{PurchaseOrder}(PO) \} \]

\[ \text{eff}(n_1) := \{ \text{isSent}(PO) \} \]

\[ n_3 : \text{"Send cancellation"} \]

\[ \text{pre}(n_3) := \{ \text{PurchaseOrder}(PO), \text{isSent}(PO) \} \]

\[ \text{eff}(n_3) := \{ \text{isCancelled}(PO) \} \]

This simplistic process sends out a purchase order, followed by an optional cancellation. This kind of process model combines a formalized view on both the process structure and the semantics of the individual activities. Traditional workflow validation techniques focus only on the former. The semantics of individual activities is specified by capturing under which circumstances the execution of an activity in a process instance will change "the world" in which way – where "the world" is the relevant business domain as formalized by the underlying ontology.

The formal execution semantics is defined as follows.

**Definition 4.** Let \( \mathcal{G} = (N, E, O, A) \) be an annotated process graph. Let \( C \) be the set of all constants appearing in any of the annotated \( \text{pre}(n), \text{eff}(n), \text{con}(n) \). A state \( s \) of \( \mathcal{G} \) is a pair \((t_s, i_s)\) where \( t \) is a token mapping and \( i \) is an interpretation \( i : P[C] \to \{0, 1\} \). A start state \( s_0 \) is \((t_0, i_0)\) where \( t_0 \) is as in Definition 2, and \( i_0 \models T[C] \), and \( i_0 \models T[C] \land \text{eff}(n_0) \) in case \( A(n_0) \) is defined. 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. \( n \in N_{PS} \cup N_{PT} \cup N_{XJ}, i_s = i_{s'}, \text{ and } t_s \xrightarrow{n} t_{s'} \) according to Definition 2.
2. \( n \in N_{XS}, i_s = i_{s'}, \text{ and } t'(e) = t(e) - 1 \text{ if } e \in IN(n), t'(e) = t(e) + 1 \text{ if } e = e', t'(e) = t(e) \text{ otherwise, where either } e' \in OUT(n) \text{ and } A(e') \text{ is undefined, or } e' = \text{argmin}\{\text{pos}(e) \mid e \in OUT(n), A(e) \text{ is defined}, i_s \models \text{con}(e)\} \).
3. \( n \in N_T \cup \{n_+\}, t_s \xrightarrow{n} t_{s'} \) according to Definition 2, and either: \( A(n) \) is undefined and \( i_s = i_{s'} \); or \( i_s \models \text{pre}(n) \) and \( i_{s'} \models \text{min}(i_s, T[C] \land \text{eff}(n)) \) where \( \text{min}(i_s, T[C] \land \text{eff}(n)) \) is defined to be the set of all \( i \) that satisfy \( T[C] \land \text{eff}(n) \) and that are minimal with respect to the partial order defined by \( i_1 \leq i_2 \) iff \( \{p \in P[C] \mid i_1(p) = i_2(p)\} \supseteq \{p \in P[C] \mid i_2(p) = i_s(p)\} \).

An execution path is a transition sequence starting in a start state \( s_0 \). A state \( s \) is reachable if there exists an execution path ending in \( s \).
Given an annotated process graph \((\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})\), we will use the term execution path of \((\mathcal{N}, \mathcal{E})\) to refer to an execution over tokens that acts as if \(\mathcal{A}\) was completely undefined (in which case Definition 4 essentially simplifies to Definition 2).

The part of Definition 4 dealing with \(n \in \mathcal{N}_{P_S} \cup \mathcal{N}_{P_J} \cup \mathcal{N}_{X_J}\) parallels Definition 2, and should be easy to understand: the tokens pass as usual, and the interpretation remains unchanged. For \(n \in \mathcal{N}_{X_S}\), the definition says that an output edge \(e'\) is selected where either \(e'\) is not annotated, or \(e'\) has the smallest position among those edges whose condition 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 (Section 11.2 in [31]). If no edges are annotated, then the analysis must foresee that an arbitrary case distinction may be created later on during the modelling, so no assumptions can be made on the form of that case distinction, so any possibility must be taken into account. Definition 2 just generalises these two extremes in the straightforward way.

Let us finally consider the “semantic” part of Definition 4, dealing with task nodes and their semantic annotations. First, note that we interpret the quantifiers in \(T\) over the constants \(C\) that are used in the annotation. The rationale is that the process should execute based on those entities that are actually available (it remains open to examine whether it makes sense to drop this assumption). Consider now the start states, of which there may be many, namely all those that comply with \(T\), as well as \(\text{eff}(n_0)\) if that is annotated. This models the fact that, at design time, we don’t know the precise situation in which the process will be executed. All we know is that, certainly, this situation will comply with the domain behaviour given in the ontology, and with the properties guaranteed as per the annotation of the start node (if any). The semantics of task node executions is a little more intricate. If \(\mathcal{A}(n)\) is undefined, then the logical state \(i\) remains of course unchanged. If \(\mathcal{A}(n)\) is defined, then \(\text{pre}(n)\) is required to hold. The tricky bit lies in the definition of the possible outcome states \(i'\) in the latter case. Our semantics defines this to be the set of all \(i'\) that comply with \(T\) and \(\text{eff}(n)\), and that differ minimally from \(i\). This is where we draw on the AI actions and change literature, for a solution to the frame and ramification problems. The latter problem refers to the need to make additional inferences from \(\text{eff}(n)\), as implied by \(T\); 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 a web service makes a booking via account A, then the balance of account B should remain unaffected; this is reflected in the requirement that \(i'\) differs minimally from \(i\). More precisely, \(i'\) is allowed to change \(i\) only where necessary, in the sense that there is no \(i''\) that makes do with fewer changes. This semantics follows the possible models approach (PMA) [15]; while this approach is not uncontroversial, it underlies most of the recent work on formal semantics for execution of Semantic Web services (e.g. [34,35,36]). Alternative semantics from the AI literature (see [37] for an excellent overview) could be used in principle; this is a topic for future research.

**Example 2.** Let us consider an example to illustrate the minimal change semantics of task node executions. Consider a process with a task node \(n\) that cancels a purchase order PO. Suppose that cancellation is annotated in terms if the effect
\[ \text{eff}(n) = \{ \text{isCancelled}(PO) \} \]. Suppose further that the ontology contains the predicate \( \text{isCancelled}(.) \), as well as the axiom specifying that any order can be only confirmed, or cancelled, but not both, i.e.,

\[
\phi_1 = (\forall x : \neg\text{isCancelled}(x) \lor \neg\text{isConfirmed}(x))
\]

is an axiom in \( T \). Now, say we execute \( n \) in a state \( s \) where \( PO \) is confirmed, i.e., \( i_s(\text{isConfirmed}(PO)) = 1 \). Which are the possible resulting states \( s' \), with \( s \xrightarrow{n} s' \)? By the definition of \( \min(i, T[C] \land \text{eff}(n)) \) in Definition 4, any such state must satisfy

\[
(\forall x : \neg\text{isCancelled}(x) \lor \neg\text{isConfirmed}(x))[PO] \land \text{isCancelled}(PO)
\]

which is the same as

\[
(\neg\text{isCancelled}(PO) \lor \neg\text{isConfirmed}(PO)) \land \text{isCancelled}(PO)
\]

which means of course that \( s' \) must satisfy \( i_{s'}(\text{isConfirmed}(PO)) = 0 \). That is, the value of \( \text{isConfirmed}(PO) \) is changed as a side-effect of applying \( n \).

Suppose now that we also have the predicates \( \text{inStock}(.) \) and \( \text{isPaid}(.) \), and that the (somewhat hypothetical) ontology specifies that any order which is both in stock and paid for is automatically confirmed, i.e.,

\[
\phi_2 = (\forall x : \neg\text{inStock}(x) \lor \neg\text{isPaid}(x) \lor \text{isConfirmed}(x))
\]

is an additional axiom in \( T \). Suppose about our state \( s \) that \( i_s(\text{inStock}(PO)) = 1 \) and \( i_s(\text{isPaid}(PO)) = 1 \). Now, upon executing \( n \), as pointed out above \( PO \) is no longer confirmed and so \( \phi_2 \) is no longer true and we must “repair” it. More formally, any state \( s' \) that complies with \( T[C] \land \text{eff}(n) \) will have to change \( i_s \) in a way so that it complies with \( \phi_2 \). Since, in difference to \( \phi_1 \), \( \phi_2 \) is not binary, this spawns a non-trivial behavior of the minimal change semantics. There are three options to “repair” \( \phi_2 \): falsify \( \text{inStock}(PO) \), falsify \( \text{isPaid}(PO) \), or falsify both.\(^{10} \) The first two options each yield a resulting state \( s' \); the latter does not yield a resulting state \( s' \) because that option is not a minimal change. One needs not assume that \( PO \) is neither in stock nor paid. It suffices to assume one of those. The intuitive meaning of this semantics is that, since \( PO \) was cancelled (by \( n \)), something bad must have happened to \( PO \), namely either it must have run out of stock or the payment must have been cancelled. While, of course, both may be the case, this seems an unlikely assumption and is hence not considered.

The case of the binary clause in Example A.1 should be clear; binary clauses specify certain consequences that \( \text{must} \) be implied by particular effects. In that way, binary clauses are a convenient modelling construct, and their semantics is “uncritical” in that there is no ambiguity about their implications. This is not so for clauses with more than 2 literals, as exemplified by the axiom \( \phi_2 \) in Example A.1; there, the implications of the theory are much more subtle. We remark that, as has been argued in the AI literature,

\(^{10} \) Making \( \text{isConfirmed}(PO) \) true is not an option because \( \neg\text{isConfirmed}(PO) \) follows logically from the effect of \( n \) and \( \phi_1 \).
there are cases where the PMA semantics behaves counter-intuitively. Many alternative solutions have been proposed – see the aforementioned [37] as an entry point – but there is no single one that is considered “best”. As stated, exploring some of the alternative solutions is a topic for future work.

Correctness Properties  We now formally define the correctness properties that we consider in this paper. We start with a definition of parallel – un-ordered – task nodes.

Definition 5. Let \( \mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A) \) be an annotated process graph, \( n_1, n_2 \in \mathcal{N} \). We say that \( n_1 \) and \( n_2 \) are parallel, written \( n_1 \parallel n_2 \), if there exist execution paths \( t_1 \) and \( t_2 \) of \((\mathcal{N}, \mathcal{E})\) so that \( n_1 \) is executed before \( n_2 \) on \( t_1 \), and \( n_2 \) is executed before \( n_1 \) on \( t_2 \).

Precisely, \( t_1 = s_{t_1}^0 \rightarrow n_1^0 \rightarrow n_1^1 \rightarrow \ldots \rightarrow n_1^k = s_{t_1}^k \), \( t_2 = s_{t_2}^0 \rightarrow n_2^0 \rightarrow n_2^1 \rightarrow \ldots \rightarrow n_2^l = s_{t_2}^l \), and ex. \( i_1 < j_1 \) and \( i_2 < j_2 \) such that \( n_{i_1}^1 = n_1, n_{i_1}^1 = n_2, n_{i_2}^2 = n_2, \) and \( n_{i_2}^2 = n_1 \).

Note that \( t_1 \) and \( t_2 \) here are token executions, ignoring the semantic annotations; i.e., Definition 5 refers only to the workflow structure.

Definition 6. Let \( \mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A) \) be an annotated process graph. Let \( n \in \mathcal{N}_T \cup \{n_+\} \). Then, \( n \) is called
- reachable iff there exists a reachable state \( s \) so that \( t_s(IN(n)) > 0 \);
- executable iff, for all reachable states \( s \) with \( t_s(IN(n)) > 0 \), we have that \( s \models pre(n) \).

\( \mathcal{G} \) is reachable (executable) iff all \( n \in \mathcal{N}_T \cup \{n_+\} \) are reachable (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 \( T \land eff(n_1) \land pre(n_2) \) is not satisfiable;
- \( n_1 \) and \( n_2 \) have an effect conflict if \( T \land eff(n_1) \land eff(n_2) \) is not satisfiable.

Consider first the notions of reachable and executable task nodes \( n \). Reachability is important because, if \( n \) is not reachable, then it is completely useless; this certainly indicates a malfunction of the process model. 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, e.g. based on BPEL, then the implementation of \( n \) will be executed anyway, which may lead to 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.

Reachability and executability are both temporal properties on the behaviour of the process, and of course it may be of interest to allow arbitrary validation properties via a suitable temporal logic (see e.g. [38,22]). 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 (perhaps more so than the fully general case allowing arbitrarily complex quantification which may never be used in practice).

Example 3. To illustrate the notion of executability, consider the handling of a purchase order (PO) in a business process model, e.g., in Figure 6. Here, a purchase order is sent
to the supplier, and after receiving a confirmation the process continues with either of two possible branches, one of them leading to a task where a cancellation is sent. Formally the activities are represented in the following way:

Formally the activities are represented in the following way:

\[ P := \{ \text{PurchaseOrder}(.), \text{isConfirmed}(.), \text{isCancelled}(.), \text{isSent}(.) \} \]
\[ C := \{ \text{PO} \} \]
\[ N_T := \{ n_1, n_2, n_3 \} \]

\[ n_1 \text{ ("Send PO"):} \]
\[ \text{pre}(n_1) := \{ \text{PurchaseOrder(PO)} \} \]
\[ \text{eff}(n_1) := \{ \text{isSent(PO)} \} \]

\[ n_2 \text{ ("Receive Confirmation"):} \]
\[ \text{pre}(n_2) := \{ \text{PurchaseOrder(PO), isSent(PO)} \} \]
\[ \text{eff}(n_2) := \{ \text{isConfirmed(PO)} \} \]

Regarding the third activity, the cancellation, there are different cases leading to different answers of the executability question:

- The precondition of the cancellation requires only that the purchase order has been sent - formally:

\[ n_3 \text{ ("Send Cancellation"):} \]
\[ \text{pre}(n_3) := \{ \text{PurchaseOrder(PO), isSent(PO)} \} \]
\[ \text{eff}(n_3) := \{ \text{isCancelled(PO)} \} \]

Then, executability is given.

- The precondition of the cancellation is that the purchase order has been sent, but not confirmed - formally:

\[ n_3 \text{ ("Send Cancellation"):} \]
\[ \text{pre}(n_3) := \{ \text{PurchaseOrder(PO), isSent(PO), \neg isConfirmed(PO)} \} \]
\[ \text{eff}(n_3) := \{ \text{isCancelled(PO)} \} \]

Then, executability of the cancellation task is not given.

\[ \text{Fig. 6. Example of a process with non-executable task nodes (clipping of a BPMN diagram).} \]
The question which of the cases applies is domain and case specific. While it is common for mail-order or online retailers to accept cancellation even after delivery, the situation is a different one with perishable goods. Now, the domain specific aspects can be modelled as semantic annotations and executability is one criterion if the usage in a process model is consistent.

Note that, while the example in Fig. 6 is simplistic and the potential flaw is easy to spot, that process models in practice typically contain many more activities, gateways, and edges. It can be assumed that the ease of spotting such flaws decreases as (i) the model grows; (ii) the number of activities that are influential to the flaw grows; and (iii) the background ontology gets more complex.

Consider now the precondition and effect conflicts from Definition 6. Such conflicts indicate that the semantic annotations of different task nodes may be in conflict; \( n_1 \) may jeopardise the precondition of \( n_2 \), or \( n_1 \) and \( n_2 \) may jeopardise each other’s effects.\(^{11}\) 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 \( n_1 \parallel n_2 \). Apart from that, it is debatable to some extent whether such conflicts represent flaws, or whether they are a natural phenomenon of the modelled process. Our standpoint is that they are flaws, because in a parallel execution it may happen that the conflicting nodes appear at the same time.

**Example 4.** To illustrate the notion of precondition and effect conflicts, consider the handling of a purchase order (PO) as above and in Figure 7. After the purchase order is sent to the supplier, two process branches are pursued in parallel: one of them deals with receiving a confirmation, while the other one eventually leads to the cancellation of the purchase order. Formally:

\[P := \{ \text{PurchaseOrder()}, \text{isConfirmed()}, \text{isCancelled()}, \text{isSent}() \}\]
\[C := \{ \text{PO} \}\]
\[\mathcal{N}_T := \{ n_1, n_2, n_3 \}\]

\(^{11}\) For illustration it is useful to consider the special case where \( T \) is empty: then, a precondition conflict means there exists \( l \in \text{eff}(n_1) \cap \neg \text{pre}(n_2) \); similarly for effect conflicts.
\( n_1 \) (“\textit{Send PO}”):
\[
\begin{align*}
\text{pre}(n_1) & := \{ \text{PurchaseOrder}(PO) \} \\
\text{eff}(n_1) & := \{ \text{isSent}(PO) \}
\end{align*}
\]
\( n_2 \) : “\textit{Receive Confirmation}”
\[
\begin{align*}
\text{pre}(n_2) & := \{ \text{PurchaseOrder}(PO), \text{isSent}(PO) \} \\
\text{eff}(n_2) & := \{ \text{isConfirmed}(PO) \}
\end{align*}
\]

As in the previous example, the question if this process contains flaws depends on the actual annotation of the activities, in particular the cancellation. Further, the ontology plays a role.

Say the precondition of a cancellation is that the purchase order has been sent, but not confirmed:

\( n_3 \) : “\textit{Send Cancellation}”
\[
\begin{align*}
\text{pre}(n_3) & := \{ \text{PurchaseOrder}(PO), \text{isSent}(PO), \neg \text{isConfirmed}(PO) \} \\
\text{eff}(n_3) & := \{ \text{isCancelled}(PO) \}
\end{align*}
\]

Then, \( n_2 \) and \( n_3 \) have a precondition conflict.

Now, say the predicates “\textit{isCancelled}” and “\textit{isConfirmed}” are modelled to be mutually exclusive in the ontology:

\[
T := \{ \forall x : (\neg \text{isSent}(x) \lor \neg \text{isConfirmed}(x)) \}
\]

Then, \( n_2 \) and \( n_3 \) have an effect conflict.

Again, the added value lies in the flexibility of the models, as well as the ability to spot conflicts of this type in complex process models.

Note that reachability can be established as a side-effect of executability:

**Lemma 2.** Let \( \mathcal{G} = (N, E, O, A) \) be an annotated process graph without case distinctions where \( (N, E) \) is sound and all \( n \in N_T \) are executable. Then all \( n \in N_T \) are reachable.

**Proof Sketch:** By definition there exists a sequence \( e \) of edges from \( n_0 \) to \( n \). By soundness and executability, and because none of the edges in \( e \) is annotated with a condition, one can easily use \( e \) to construct an execution path that reaches \( n \).

Below, we provide analysis methods only for checking executability. With Lemma 2, once that is established, we know that reachability is also given. Note that Lemma 2 does not hold if \( \mathcal{G} \) has case distinctions: a node may not be reachable because an edge condition on \( e \) may never become true.

**A.2 Polynomial-Time Analysis**

We now specify efficient analysis algorithms for a particular class of processes, namely:

**Definition 7.** Let \( \mathcal{G} = (N, E, O, A) \), \( O = (P, T) \), be an annotated process graph. \( \mathcal{G} \) is basic if it contains neither loops nor case distinctions, and \( T \) is binary.
Note that our example from the previous Section is a basic annotated process graph. For complexity considerations, we will in the following assume fixed arity, i.e., a fixed upper bound on the arity of the predicates $P$. This is a realistic assumption because predicate arities are typically very small in practice (e.g., in Description Logics the maximum arity is 2). Given a process graph whose annotations mention the constants $C$, and a set $L$ of literals (such as a task node effect), in the following we denote $T := \{l \in P[C] \mid T \wedge L \models l\}$, i.e., $T$ is the closure of $L$ under implications in the theory $T$. Since $T$ is binary, $T$ can be computed in polynomial time given fixed arity [20].

Note that, with binary $T$, an effect conflict can be easily detected as the (negative) overlap of the closure over the effect sets, i.e., $T \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ is not satisfiable iff $\text{eff}(n_1) \cap \neg \text{eff}(n_2) \neq \emptyset$, and similarly for precondition conflicts.

Our analysis algorithm performs three steps: (1) Determine a numbering $\#$ of the edges $E$ so that, whenever task node $n_1$ is ordered before task node $n_2$ in every process execution, then $\#(IN(n_1)) < \#(IN(n_2))$. (2) Using $\#$, determine all pairs of parallel task nodes; with that, find all precondition and effect conflicts. (3) Determine (making use also of parallelity information), for each edge $e$, the set of literals that is always true when $e$ is active; with that, find all non-executable task nodes. In what follows, we explain in detail steps (2) and (3), in that order (Sections A.2 and A.2). Step (1) is relatively straightforward, and can be looked up in Appendix B.

**Detecting Precondition and Effect Conflicts** Step (2) propagates matrix functions $M$ along the edges of the process graph. $M$ contains one entry for every pair of edges in $E$; $\#$ is used for indexing into $M$. The propagation steps are defined below. We use the following helper notations: $\#^{-1}$ is the inverse function of $\#$, i.e., $\#^{-1}(i) = e$ iff $\#(e) = i$; $M_i$ is the $i$th row of $M$; given a node $n$, $\#IN_{\text{max}}(n) := \max\{\#(e) \mid e \in IN(n)\}$ is the maximum number of any incoming edge, and similarly for $\#OUT_{\text{min}}(n)$ and $\#OUT_{\text{max}}(n)$.

**Definition 8.** Let $G = (\mathcal{N}, \mathcal{E}, O, A)$ be an annotated process graph. A matrix $M$ is a function $M : \{0, \ldots, |\mathcal{E}| - 1\} \times \{0, \ldots, |\mathcal{E}| - 1\} \mapsto \{0, 1, \bot\}$. We define the matrix $M_0$ as $(M_0)^i_j = 0$ if $i = j$, $(M_0)^i_j = \bot$ otherwise. Let $M$ and $M'$ be matrices, $n \in \mathcal{N}$. We say that $M'$ is the propagation of $M$ at $n$ iff we have:

1. For all $i, j \in \{0, \ldots, \#IN_{\text{max}}(n)\}$, we have $M_i^j \neq \bot$.
2. For all $e \in OUT(n)$ and $j \in \{0, \ldots, |\mathcal{E}| - 1\} \setminus \{\#(e)\}$, we have $M_{\#(e)}^j = \bot$.

As well as one of the following:

3. $n \in \mathcal{N}_T$ and $M'$ is given by $M'^{i_j} = M_{\#(IN(n))}^{i_j}$ if $\#(OUT(n)) = j$ and $i < j$.
   $M'^{i_j} = M_j^i$ otherwise.
4. $n \in \mathcal{N}_{PS}$ and

\[
M'^{i_j} = \begin{cases} 
M_{\#(IN(n))}^i & \#^{-1}(j) \in OUT(n) \text{ and } i < \#OUT_{\text{min}}(n) \\
1 & \#^{-1}(j) \in OUT(n) \text{ and } i \neq j \\
M_j^i & \#OUT_{\text{min}}(n) \leq i \leq \#OUT_{\text{max}}(n) \\
\end{cases}
\]
5. \( n \in N_{XS} \) and
\[
M'^i_j = \begin{cases} 
M^i_{\#(IN(n))} & \#^{-1}(j) \in OUT(n) \text{ and } i < \#OUT_{\min}(n) \\
0 & \#^{-1}(i) \in OUT(n) \text{ and } i \neq j \\
M^i_j & \text{and } \#OUT_{\min}(n) \leq i \leq \#OUT_{\max}(n) \\
\end{cases}
\]

6. \( n \in N_{PJ} \) and
\[
M'^i_j = \begin{cases} 
1 & \#(OUT(n)) = j \text{ and } i < j \text{ and for all } e \in IN(n) : M^i_{\#(e)} = 1 \\
0 & \#(OUT(n)) = j \text{ and } i < j \text{ and for all } e \in IN(n) : M^i_{\#(e)} = 0 \\
M^i_j & \text{otherwise.} \\
\end{cases}
\]

7. \( n \in N_{XJ} \) and
\[
M'^i_j = \begin{cases} 
1 & \#(OUT(n)) = j \text{ and } i < j \text{ and } \exists e \in IN(n) : M^i_{\#(e)} = 1 \\
0 & \#(OUT(n)) = j \text{ and } i < j \text{ and for all } e \in IN(n) : M^i_{\#(e)} = 0 \\
M^i_j & \text{otherwise.} \\
\end{cases}
\]

If \( M^* \) results from starting in \( M_0 \), and stepping on to propagations until no more propagations exist, then we call \( M^* \) an \( M \)-propagation result.

Definition 8 is hard to read; however, the underlying key ideas are simple. The matrix \( M \) annotated at edge \( e \), at any point in time, provides complete information about all edges preceding \( e \) according to \( \# \); precedence according to \( \# \) is meaningful because \( \# \) respects task node orderings. The definition of \( M_0 \) is obvious, likewise case 3 which handles task nodes. In a parallel split \( n \) (case 4), \( n \)'s OUT edges copy the information from \( n \)'s IN edge, except that the OUT edges are marked to be parallel with respect to each other. For xor splits (case 5), the OUT edges are marked to be not parallel with respect to each other. In a parallel join (case 6), an OUT edge is parallel to a preceding edge iff all IN edges are. Finally, xor joins (case 7) are only executed if all IN edges agree on parallelism: if they don’t, then the underlying workflow is unsound; if they do, then the OUT edge simply copies the information from the IN edges.

Note that due to the absence of cycles and multiple instantiations, no node can be executed in parallel to itself - in other words, the parallelity relation here is irreflexive, and the matrix fields on the diagonal always have the value 0 \( (M^i_j = 0) \). However, if an execution of edge \( e \) may occur in parallel to an execution of edge \( e' \), then this is true vice versa - the parallelity relation is symmetric, and so is the matrix \( (M^i_{\#(e')} = M^i_{\#(e)}) \).

An exemplary matrix is shown in Figure 8.

The following lemma has been stated in Section 3.1 already. Note that the wording of the claims is not identical; our wording here uses our formal terminology and is hence more precise. The same is true of all the formal claims in the remainder of this paper.

Lemma 1 Let \( \mathcal{G} = (N, E, O, A) \) be an annotated process graph. There exists exactly one \( M \)-propagation result \( M^* \), and for all \( n_1, n_2 \in N_T \) we have \( n_1 \parallel n_2 \) iff \( M^*_{\#(IN(n_2))} = 1 \). The time required to compute \( M^* \) is polynomial in the size of \( \mathcal{G} \).
Fig. 8. An example matrix. Due to being irreflexive and symmetric, the upper right half (shown with a light blue background) does not have to be represented, internally.

**Proof Sketch:** Uniqueness of $M^*$ follows because 1 (2) requires all IN (OUT) edges to be determined (not determined), and any propagation affects only OUT edges.

Parallelism between two nodes is determined by the routing constructs between the start node and these two nodes. Namely, we have $n_1 \parallel n_2$ iff $n_1$ and $n_2$ have a common ancestor $n \in \mathcal{N}_{PS}$ with no corresponding join node in between, and $n_1$ and $n_2$ do not lie on different sides of an xor-split. By construction of cases 4–7, which propagate exactly this information, these conditions hold true iff $M^*#(IN(n_2)) = 1$.

An obvious upper bound on the time required to compute $M^*$ is $O(|\mathcal{N}| \cdot |\mathcal{E}|^2)$: in each of $|\mathcal{N}|$ propagation steps, maximally all pairs of edges need to be considered. ■

Given the parallelism information, it is easy to determine any precondition and effect conflicts. One simply loops over all pairs of parallel task nodes and checks whether they are in conflict; c.f. the discussion of Corollary 1 in Section 3.1.

**Detecting Non-Executable Activities** Having completed the computation of $M^*$, and after all effect conflicts have been pointed out to the user and removed, we can proceed to step (3) of our analysis algorithm. This determines, for each edge $e$, the set of literals that is always true when $e$ is active. Again, this computation is based on propagation steps; this time, the propagations update sets of literals that are assigned to the edges. In the fixpoint, these literal sets are exactly the desired ones. The information from $M^*$ is used to determine the “side effects” that any task node may have, on edges other than its own OUT edge.

**Definition 9.** Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A)$ be a basic annotated process graph without effect conflicts, and with constants $C$. We define the function $I_0 : \mathcal{E} \mapsto 2^{P[C]} \cup \{\bot\}$ as $I_0(e) = eff(n_0)$ if $e = OUT(n_0)$, $I_0(e) = \bot$ otherwise. Let $I, I' : \mathcal{E} \mapsto 2^{P[C]} \cup \{\bot\}, n \in \mathcal{N}$. We say that $I'$ is the propagation of $I$ at $n$ iff one of the following holds:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
1. $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{XS}$, and $I(IN(n)) \neq \bot$, and for all $e \in OUT(n)$ we have $I(e) = \bot$, and $I'$ is given by $I'(e) = I(IN(n))$ if $e \in OUT(n)$, $I'(e) = I(e)$ otherwise.

2. $n \in \mathcal{N}_{P,J}$, and for all $e \in IN(n)$ we have $I(e) \neq \bot$, and $I(OUT(n)) = \bot$, and $I'$ is given by $I'(e) = \bigcup_{e' \in I(N)} I(e')$ if $e = OUT(n)$, $I'(e) = I(e)$ otherwise.

3. $n \in \mathcal{N}_{X,J}$, and for all $e \in IN(n)$ we have $I(e) \neq \bot$, and $I(OUT(n)) = \bot$, and $I'$ is given by $I'(e) = \bigcap_{e' \in I(N)} I(e')$ if $e = OUT(n)$, $I'(e) = I(e)$ otherwise.

4. $n \in \mathcal{N}_T$, and $I(IN(n)) \not\in \bot$, and $I(OUT(n)) = \bot$, and

$$I'(e) = \begin{cases} \text{eff}(n) \cup \{I(IN(n)) \setminus \text{eff}(n)\} & e = OUT(n) \\ I(e) \setminus \text{eff}(n) & M^*(#(e)) = 1 \text{ and } I(e) \neq \bot \\ I(e) & \text{otherwise} \end{cases}$$

If $\mathcal{A}(n)$ is not defined then $\text{eff}(n) := \emptyset$ in the above.

If $I^*$ results from starting in $I_0$, and stepping on to propagations until no more propagations exist, then we call $I^*$ an $I$-propagation result.

Definition 9 is a hard to read but, as before, relies on straightforward key ideas. The definition of $I_0$ is obvious. For split nodes (case 1), the OUT edges simply copy their sets from the IN edge. For parallel joins (case 2), the OUT edge assumes the union of $I(e)$ for all IN edges $e$; for xor joins (case 3), the intersection is taken instead. The handling of task nodes (case 4) is somewhat more subtle. First, although there are no effect conflicts it may happen that a parallel node has inherited (though not established itself, due to the postulated absence of effect conflicts) a literal which the task node effect contradicts; hence line 2 of case 4. Second, we must determine how the effect of $n$ may affect any of the possible interpretations prior to executing $n$. This is non-trivial due to the complex semantics of task executions, based on the PMA [15] definition of minimal change for solving the frame problem, c.f. Section A.1. Our key observation is:

(*) With binary $T$, if executing a task makes literal $l$ false in at least one possible interpretation, then $\neg l$ is necessarily true in all possible interpretations.

Due to this observation, it suffices to subtract $\neg \text{eff}(n)$ in the top and middle lines of the definition of $I'(e)$: $l$ does not become false in any interpretation, unless $\neg l$ follows logically from $\text{eff}(n)$.

Importantly, (*) does not hold for more general $T$. To see this, consider the following example where $T$ consists of a single clause with 3 literals, namely the clause $\neg G() \lor \neg H() \lor I()$. The three predicates used all have arity 0, and the clause is Horn, corresponding to the implication $G \land H \Rightarrow I$. Let’s say the start node $n_0$ is annotated with $\text{eff}(n_0) = \{G(), H(), I()\}$, i.e., we know that all the predicates are true when we start to execute the process, and by Definition 4 there is just a single start state $s_0$, accordingly. Now, say $n_0$ connects to a task node $n$ where $\text{eff}(n) = \{\neg I()\}$. Consider the possible transitions from $s_0$ to a state $s'$ via $n$. Definitely, $s'$ must satisfy $\neg I()$, because

---

12 The interactions of parallel nodes with conflicting effects may be quite subtle, and require a much more complicated propagation algorithm. We are currently working on such an extended algorithm, which will allow the user to tolerate effect conflicts if desired.
that is dictated by the annotation of \(n\). However, since \(\neg G() \lor \neg H() \lor I()\) must also hold, \(G()\) and \(H()\) cannot both remain true – i.e., given that we know the implication \(G \land H \Rightarrow I\) always holds, it must be the case that falsifying \(I\) in \(s_0\) has a side effect on \(G\) and/or \(H\). By the possible models approach assumed in Definition 4, we get two possible resulting states \(s'_1\) and \(s'_2\), where \(s'_1\) makes \(G\) false and keeps \(H\) true, and \(s'_1\) makes \(H\) false and keeps \(G\) true (making both \(G\) and \(H\) false is not a minimal change).

Hence, after executing \(n\), \(\neg I()\) is the only literal that holds true in all possible interpretations. In particular, \(G()\), which was true before executing \(n\), disappeared although \(\neg G()\) does not follow logically from \(T \land \text{eff}(n)\); same for \(H()\). This is in contrast to (*). Intuitively, restricting \(T\) to binary clauses ensures that the side effects it incurs are always “deterministic”. We have:

**Theorem 3** Let \(G = (N, E, O, A)\) be a basic annotated process graph without effect conflicts. There exists exactly one \(I\)-propagation result \(I^*\). \(G\) is executable if, for all \(n \in N_T \cup \{n_+\}\), \(\text{pre}(n) \subseteq I^*(IN(n))\). With fixed arity, the time required to compute \(I^*\) is polynomial in the size of \(G\).

**Proof Sketch:** Uniqueness of \(I^*\) is obvious due to the conditions on incoming edges being defined and outgoing edges being not yet defined.

Assume that all \(n \in N_T \cup \{n_+\}\) are executable. For any edge \(e\), denote by \(\bigcap e\) the set of literals that are always – in any reachable state \(s\) – true whenever \(s_e(IN(n)) > 0\).

Clearly, we have \(\text{pre}(n) \subseteq \bigcap IN(n)\), for all \(n \in N_T\). Now, one can prove that, given all \(n \in N_T \cup \{n_+\}\) are executable, \(I^*(e) = \bigcap e\), for all \(e\). This is obvious for \(OUT(n_0)\), as well as the outgoing edges of split nodes (case 1). If we join parallel branches, then all their results will be true (case 2). If we join alternative branches, then only their common results will be true (case 3). For task nodes (case 4), the above (*) shows that, with binary \(T\), every literal \(l\) true at \(IN(n)\) remains true at \(OUT(n)\) unless its opposite \(\neg l\) can be derived from \(n\)’s effect. (Effect conflicts must be excluded because those may lead to interactions between parallel nodes, which are not handled by the algorithm.)

We get that, if all \(n \in N_T \cup \{n_+\}\) are executable, then, for all \(n \in N_T \cup \{n_+\}\), \(\text{pre}(n) \subseteq I^*(IN(n))\). The other direction also follows immediately: let \(n \in N_T \cup \{n_+\}\) be a node with \(\text{pre}(n) \nsubseteq I^*(IN(n))\), so that all of \(n\)’s predecessors \(n’\) have \(\text{pre}(n’) \subseteq I^*(IN(n))\). By the same arguments as above, it follows that all \(n’\) are executable and hence that \(\bigcap IN(n) = I^*(IN(n))\). Hence \(n\) is not executable.

With fixed arity, the number of different literals \(|P[C]|\) is polynomial in the size of \(G\); with binary \(T\), \(L\) for any set \(L\) of literals can be computed in \(O(|P[C]|^3)\), so \(\text{eff}(n)\) can be pre-computed for every relevant \(n\) in time \(O(|N_T| \ast |P[C]|^2)\). Hence an upper bound on the required time required for \(I^*\) is \(O(|N_T| \ast |P[C]|^3 + |N| \ast |P[C]| \ast |E|)\).”

A.3 Computational Borderline

Since the class of basic processes can be validated in polynomial time, it is interesting to determine the computational borderline of this positive result: *What happens if we generalise this class?* We give negative results for allowing case distinctions, and for allowing more general ontologies. It is an open question whether or not loops, or structured loops, can be dealt with efficiently. We are currently investigating this.
Theorem 2 Assume an annotated process graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A)$ that is basic except that $A(e)$ may be defined for some $e \in \mathcal{E}$. Deciding whether $\mathcal{G}$ is reachable is $\text{NP}$-hard. Deciding whether $\mathcal{G}$ is executable is $\text{coNP}$-hard.

Proof Sketch: Both results can be proved by a reduction from SAT. Assume a propositional CNF formula with $n$ variables and $m$ clauses. After the start node, $n$ consecutive non-annotated xor splits/joins allow to “choose a value” for each variable, by a corresponding task node effect in each of the two branches in each xor construct. Afterwards, $m$ consecutive annotated xor splits branch into a final xor join if none of the respective clause’s literals is true; else, the xor branches into the next xor split. The process is reachable – all of these xor splits and joins can be reached – iff the formula is satisfiable. A similar construction works for executability checking.

Theorem 1 Assume an annotated process graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A)$ that is basic except that $T$ is not binary. Deciding whether $\mathcal{G}$ is reachable is $\Sigma_2^p$-hard in general, and $\text{NP}$-hard if $T$ is Horn. Deciding whether $\mathcal{G}$ is executable is $\Pi_2^p$-hard in general, and $\text{coNP}$-hard if $T$ is Horn.

Proof Sketch: This is a consequence of the complexity of “belief update” [16]. A belief update corresponds to the execution of a task node $n$: the “belief to be updated” is the set of interpretations that may arise when $n$’s incoming edge is activated; the “update” is $n$’s effect. A difference to [16] is that the latter uses complex “update formulas” instead of complex $T$. That difference is overcome by a construction where $T$ is the update formula conditioned on a new fact $q$, and $n$’s effect makes $q$ true and hence “switches $T$ on”. The $(\mathcal{N}, \mathcal{E})$ part of the construction is straightforward for both reachability and executability checking. The proofs of [16] can be adapted to show all the claims.

Basic process graphs are generous in that they allow predicates with large arity, and in that they do not require the start node effect $\text{eff}(n_0)$ to be complete. One might wonder if sacrificing this generality could buy us anything. This is not the case: the proofs of Lemmas 2 and 1 do not exploit this feature. Our main result regarding the computational borderline follows directly:

Corollary 2. Assume an annotated process graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, O, A)$. Unless $P = NP$, neither reachability nor executability can be decided in polynomial time if one of the following holds: $\mathcal{G}$ is basic except that it may contain case distinctions; $\mathcal{G}$ is basic except that $T$ may involve Horn clauses. This holds even if predicate arity is fixed to 0, and $\text{eff}(n_0)$ is complete, i.e., if for every $p \in P[C]$: either $p \in \text{eff}(n_0)$ or $\neg p \in \text{eff}(n_0)$. 
B  Technical Details

Herein we provide some missing technical details, as well as full formal proofs. We start with the polynomial time analysis methods from Section A.2, then proceed to the computational borderline investigated in Section A.3.

B.1 Polynomial-Time Analysis

We first specify how to compute the enumeration functions \( \# \) used in Section A.2.

**Definition 10.** Let \( G = (N, E, O, A) \) be an annotated process graph. Let \( \text{enum} \) and \( \text{enum}' \) be functions \( \text{enum}, \text{enum}': E \mapsto \{0, 1, \ldots, |E| - 1\} \cup \{\perp\} \) mapping each edge to an integer or the \( \perp \) symbol, where \( \perp < 0 \) in any comparison. Let \( n \in N \) be a node. We say that \( \text{enum}' \) is the propagation of \( \text{enum} \) at \( n \) iff one of the following holds:

1. \( n \in N_T, \text{enum}(IN(n)) \neq \perp, \text{enum}(OUT(n)) = \perp, \) and
   \[
   \text{enum}'(e) = \begin{cases} 
   \max\{\text{enum}(e) \mid e \in E\} + 1 & e = OUT(n) \\
   \text{enum}(e) & \text{otherwise}
   \end{cases} \tag{1}
   
2. \( n \in N_{PJ} \cup N_{XJ}, \) for all \( e \in IN(n) \) \( \text{enum}(e) \neq \perp, \text{enum}(OUT(n)) = \perp, \) and
   \[
   \text{enum}'(e) = \begin{cases} 
   \max\{\text{enum}(e) \mid e \in E\} + 1 & e = OUT(n) \\
   \text{enum}(e) & \text{otherwise}
   \end{cases} \tag{2}
   
3. \( n \in N_{PS} \cup N_{XS}, \text{enum}'(IN(n)) \neq \perp, \) for all \( e \in OUT(n) \) \( \text{enum}(e) = \perp, \) and
   \[
   \text{enum}(e) = \begin{cases} 
   \max\{\text{enum}(e) \mid e \in E\} + i & e_i \in OUT(n) \\
   \text{enum}(e) & \text{otherwise}
   \end{cases} \tag{3}
   
where the outgoing edges are arbitrarily ordered: \( OUT(n) = \{e_1, \ldots, e_j\} \).

We define the function \( \text{enum}_0 \) as

\[
\text{enum}_0(e) := \begin{cases} 
0 & e = OUT(n_0), \\
\perp & \text{otherwise}.
\end{cases} \tag{4}
\]

If the function \( \text{enum}^* \) results from starting in \( \text{enum}_0 \), and stepping on to propagations until no more propagations exist, then we call \( \text{enum}^* \) an enumeration result. For ease of legibility, we abbreviate the enumeration result with \( \# := \text{enum}^* \). Also, we use the inverse of this function \( \#^{-1} \), which maps from an index number to the corresponding edge.

To give an intuition behind the enumeration, consider a process graph where there are no XOR-Splits or XOR-Joins. Then, the edge order in the enumeration function \( \# \) corresponds to one possible firing order of an execution of the process – like an execution sequence of the edges. Or, to put it differently: on an arbitrary process graph the enumeration function \( \# \) corresponds to the order in one particular firing sequence of tokens in terms of the formalism given in Definition 2.

Before proving the formal results on the analysis methods, we define when nodes are sequential or mutually exclusive w.r.t. execution paths.
Definition 11. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph, $n_1, n_2 \in \mathcal{N}$. We say that $n_1$ and $n_2$ are sequential, written $n_1 < n_2$, if for any execution path $t$ which both contains $n_1$ and $n_2$, $n_1$ is always executed before $n_2$, and there exists at least one such $t$. Precisely, if $n_1 < n_2$ then there is no execution path $t' = s^0 \rightarrow n^0 \ldots \rightarrow n^k \rightarrow s^k$ and ex. $i < j$ such that $n^i = n_1$, $n^j = n_2$.

Definition 12. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph, $n_1, n_2 \in \mathcal{N}$. We say that $n_1$ and $n_2$ are mutex (or mutually exclusive), written $n_1 \times n_2$, if any execution path $t$ contains at most either $n_1$ or $n_2$, but never both. In other words, if $n_1$ and $n_2$ are mutex, then there is no execution path $t' = s^0 \rightarrow n^0 \ldots \rightarrow n^k \rightarrow s^k$ and ex. $i, j$ such that $n^i = n_1$, $n^j = n_2$.

Note that the sequential relation is not symmetric, but the parallel and mutex relations are.

Lemma 3. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph, $n_1, n_2 \in \mathcal{N}$. Then, $n_1$ and $n_2$ are either sequential, or parallel, or mutex.

Proof: Both, sequentiality and parallelism require there to be at least one execution path $t$ which contains $n_1$ and $n_2$ – if such a $t$ does not exist, the nodes are mutex. If $t$ exists, and if there is another execution path $t'$ in which $n_1$ and $n_2$ are executed in the opposite order to their order in $t$, then $n_1$ and $n_2$ are parallel; if there is no such $t'$, they are sequential.

Enumeration results respect sequential pairs of nodes:

Lemma 4. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph. Let $n_1, n_2 \in \mathcal{N}_T$ such that $n_1 < n_2$. Then any enumeration result # for $\mathcal{G}$ orders the incoming edges of $n_1$ and $n_2$ accordingly: $\#(IN(n_1)) < \#(IN(n_2))$.

Proof. First, it is important to note that any two sequential nodes $n_1, n_2$ with $n_1 < n_2$, are connected through a (non-empty) set of directed paths from $n_1$ to $n_2$, out of which at least one is executed in any execution sequence which contains both $n_1$ and $n_2$. If there was an execution sequence in which no such path existed, then there would be another execution sequence in which $n_2$ was executed before $n_1$, because $n_2$ would not have to wait for a token coming from $n_1$, and $n_1 < n_2$ would not hold anymore.

Now, Definition 10 follows the principle of replacing the $\bot$ symbol as the enumeration of each edge, based on the edges’ connections to the nodes. Hereby, for any node $n'$ the outgoing edges are only enumerated if (i) they still carry the $\bot$ symbol, and (ii) none of the edges in $IN(n')$ carries the $\bot$ symbol anymore – formally: $\forall e \in IN(n') : \text{enum}(e) \neq \bot, \forall e' \in OUT(n') : \text{enum}(e') = \bot$. For our token-sequential nodes $n_1, n_2$ this means that $n_1$ is always enumerated before $n_2$, because there is at least one directed path from $n_1$ to $n_2$, and the incoming edges of $n_2$ cannot be enumerated until the $\bot$ symbol has been replaced on $OUT(n_1)$.

Furthermore, the numbers assigned as enumeration exceed the existing enumeration numbers: $\max\{\text{enum}(e) \mid e \in \mathcal{E}\} + x$ with $x > 0$. Taken together, the numbers assigned to the edges around $n_2$ must be higher than the ones assigned to the edges of $n_1$, in particular $\#(IN(n_1)) < \#(IN(n_2))$. 
Lemma 1 Let $G = (N, E, O, A)$ be an annotated process graph. There exists exactly one $M$-propagation result $M^*$, and for all $n_1, n_2 \in N_T$ we have $n_1 \parallel n_2$ iff $M^{*\#(IN(n_1))}_{\#(IN(n_2))} = 1$. The time required to compute $M^*$ is polynomial in the size of $G$.

Proof:

First, we show that for an arbitrary but fixed enumeration function $\#$ the $M$-propagation result is unique. This is already stated in the proof sketch: Case 1 (2) requires all IN (OUT) edges to be determined (not determined), and any propagation affects only OUT edges. Since any edge is an incoming (outgoing) edge to exactly one node, and since the propagation over a node can only take place once, the propagation result must be unique.

The property we exploit for showing that $M^*$ is the anticipated outcome is that task nodes $n_1$ and $n_2$ are parallel if there is a state $s$ on an execution path $t$ where their incoming edges carry a token at the same time: $t_s(IN(n_1)) > 0, t_s(IN(n_2)) > 0$. Then, the nodes may be executed in either order – which, by definition, means $n_1 \parallel n_2$.

What we show here is (i) if $n_1$ and $n_2$ are sequential or mutex, then $M^{*\#(IN(n_1))}_{\#(IN(n_2))} = 0$ and (ii) if $n_1$ and $n_2$ are parallel, then $M^{*\#(IN(n_2))}_{\#(IN(n_1))} = 1$. Since, following Lemma 3, any pair of nodes is either token-sequential or token-mutex or token-parallel, the above points prove both directions of the second statement in Lemma 1.

First, we show $(n_1 < n_2) \Rightarrow (M^{*\#(IN(n_2))}_{\#(IN(n_1))} = 0)$. Note that each of the $M$-propagation cases copies the values containing 0 from the predecessor links up to the column index of the edge under discussion. Two cases are worthwhile noting: in case of $n \in N_P$, the values concerning OUT$(n)$ are set to 0 if one of the incoming edges carries a 0, and to 1 otherwise; if $n \in N_X$, then no pair of incoming edges $e_i, e_j \in IN(n)$ can be parallel in a sound process graph without loops, since that would allow the outgoing edge to carry two tokens at the same time.

Note further, that the fields on the diagonal always contain 0, and that all incoming edges have a smaller enumeration then the outgoing links. Together, this means that the 0-value from the diagonal is copied to all successors of an edge - in other words: if there is a path from edge $e$ to edge $e'$, then $M^{*\#(e)}$ is 0. Further, $n_1 < n_2$ implies that there is at least one path from $n_1$ to $n_2$ otherwise their order cannot be fixed and thus there is also a path $n_1 \to n_2$. Therefore we can conclude $(n_1 < n_2) \Rightarrow (M^{*\#(IN(n_2))}_{\#(IN(n_1))} = 0)$.

Before we continue, it is important to note that split nodes usually have to be matched by join nodes of the corresponding type, following definition 2 and the soundness criteria: a XOR split matched by a Parallel join would cause a deadlock situation; and a Parallel split matched by an XOR join would cause multiple instantiation after the join. However, there can be a set of split nodes matched by a set of corresponding join nodes, all being of the same type, as depicted in Fig. 9.

Now, let us show $n_1 \times n_2 \Rightarrow M^{*\#(IN(n_2))}_{\#(IN(n_1))} = 0$. For $n_1, n_2$ to be mutex, there must be at least one XOR split node $n'$ before $n_1$ and $n_2$, such that there is a path $p_1$ from $n'$ to $n_1$ and there is a path $p_2$ from $n'$ to $n_2$ such that $p_1$ and $p_2$ do not overlap. There can be other XOR split nodes with same properties, but there cannot be a Parallel split node in the mentioned position as argued above. Also, there can be no path from $n_1$ to $n_2$ (or vice versa), because that would result in an execution path containing both $n_1$
Fig. 9. Example: sound process with groups of matching joins and splits. (BPMN notation, thin circle for $n_0$, bold circle for $n_+$, diamond with + depicts parallel splits / joins, task nodes omitted.)

and $n_2$ – due to the non-determinism even in the presence of XOR splits. Now, let $e_1$ be the first edge on $p_1$ and let $e_2$ be the first edge on $p_2$. Following definition 8, we know that $M^\ast \#(e_1) = 0$. And for the same reasons as in the case of token-sequentiality this 0-value is propagated to $n_1$ and $n_2$. This is true for any of the XOR split nodes like $n'$, and there are no other nodes between the XOR splits and $n_1$ and $n_2$ which could change the value to 1. That is, of course, with the exception of a XOR join node that synchronizes $p_1$ and $p_2$; however, since $p_1$ and $p_2$ are assumed to contain no common node, there can be no such join node.

The analogous argument holds for $n_1 \parallel n_2$, where the split nodes are of type Parallel, setting the initial values on the outgoing edges of the splits to 1. If there was a Parallel join node between the splits and $n_1, n_2$, synchronizing the paths from the splits to $n_1, n_2$, then $n_1, n_2$ would not be parallel.

As for the computational complexity, cases 1 and 2 can be accounted for by a suitable for-each loop over the nodes, e.g., straight forward following the enumeration $\#(in)$ (in $O(|N|)$). Cases 3 - 5 make one pass over the values of the input edge in $O(|E|)$ again. Cases 6 and 7 in turn make a pass over all input edges, resulting in $O(|E|^2)$. Together, the algorithm runs in $O(|N| \ast |E|^2)$ in the worst case.

Given a process graph $G$, we denote, for any edge $e \in E$, $\bigcap_{G}^{e} := \bigcap_{s: \text{reachable}, t_{b}(e) > 0} \{l \mid l \in P[C], s \models l\}$.

That is, $\bigcap_{G}^{e}$ is the set of literals that are always true when $e$ is activated in $G$. If the context is clear, we will sometimes drop the subscript $G$ to simplify notation. We have:

**Lemma 5.** Let $G = (N, E, O, A)$, $O = (P, T)$ be an executable basic annotated process graph. Denote by $C$ the set of all constants appearing in any of the annotated $\text{pre}(n), \text{eff}(n)$. Let $G' = (N, E', O', A')$ be the modification of $G$ where $O' = (P, 1)$ and $A' \equiv A$ except that, for all $n \in N_T$, $\text{eff}(n) := \{l \in P[C] \mid T[C] \wedge \text{eff}(n) \models l\}$ if $\text{eff}(n)$ is defined, and $\text{eff}(n) := \{l \in P[C] \mid T[C] \models l\}$ otherwise. Then, for any $e \in E$, we have $\bigcap_{G}^{e} = \bigcap_{G'}^{e}$.

**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
where \( n \), in \( G \) exactly one state \( s' \) can be reached by executing \( n \) in \( s \), namely the state \( s' := (s \setminus \neg \text{eff}(n)) \cup \text{eff}(n) \).

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

First, for any \( s' \in \min(s, T[C] \land \text{eff}(n)) \) it is clear by definition that \( \text{eff}(n) \subseteq s' \).

The definition of \( s' \) as given above changes only those values. It suffices to show that \( s' \models T[C] \); then, we have \( s' \models T[C] \land \text{eff}(n) \), and clearly the set of changed values is a proper subset of any other state with the same property. Assume to the contrary of \( \neg s' \models T[C] \) and \( s' \not\models l \lor l' \), i.e., \( \neg l \in s' \) and \( \neg l' \in s' \); note here that \( T \) is binary and hence every clause has at most two literals. If \( \neg l \in \text{eff}(n) \), then \( l' \in \text{eff}(n) \) — because, given the clause \( l \lor l' \), \( l' \) is a logical consequence of \( \neg l \). With \( \text{eff}(n) \subseteq s' \), we obtain a contradiction, proving that \( \neg l \) cannot be contained in \( \text{eff}(n) \).

Similarly, we can disprove \( \neg l' \in \text{eff}(n) \). Hence, by construction of \( s' \), \( \{\neg l, \neg l'\} \subseteq s \). But then, \( s \not\models T[C] \) 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 \( G \) is exactly the same as the transition induced in \( G' \).

Hence, obviously since the graph structure is not changed in any other way, any possible difference in the sets \( \bigcap_n s_n \) would have to be due to different start states. So let us consider the start states in \( G \) and \( G' \).

The start states in \( G \) are all those with \( s_0 \models T[C] \), and \( s_0 \models T[C] \land \text{eff}(n_0) \) in case \( A(n_0) \) is defined. In \( G' \), by construction the start states are all those where \( s_0 \models 1 \land \text{eff}(n_0) \), with \( \text{eff}(n_0) = \{ l \in P[C] \mid T[C] \models l \} \) in case \( A(n_0) \) is undefined, and \( \text{eff}(n_0) = \{ l \in P[C] \mid T[C] \land \text{eff}(n_0) \models l \} \) in case \( A(n_0) \) is defined.

Obviously, this means that the set of start states of \( G' \) is a superset of the set of start states of \( G \) — any start state of \( G \) is a start state of \( G' \), 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 \( \bigcap_{G} s_0 = \bigcap_{G'} s_0 \).

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 \( G' \) 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 \( G \) and \( G' \). Let’s call this set of paths \( P \). By prerequisite, every task node is executable, and there are no conditions at the outgoing edges of xor splits. Thus we know that every path \( p \in P \) can be executed from every possible start state \( s_0 \), in both \( G \) and \( G' \). The change that \( p \) makes to \( s_0 \) is the accumulated effect of the task nodes executed on \( p \). From the above, we know that this is the same in both \( G \) and \( G' \). We can write the resulting state \( s \) as \( s = (s_0 \setminus \neg \text{eff}(p)) \cup \text{eff}(p) \), where \( \text{eff}(p) \) denotes the accumulated effect of \( p \). What exactly that latter effect is does not play a role in our argument below; the important point is that \( \text{eff}(p) \) is a function, i.e., is well-defined.

Consider now the sets \( \bigcap_G s_0 \) and \( \bigcap_{G'} s_0 \). With the above, we know that

\[
\bigcap_G e = \bigcap_{s_0,p} ((s_0 \setminus \neg \text{eff}(p)) \cup \text{eff}(p)),
\]
where \( s_0 \) ranges over the start states of \( G \) and \( p \) ranges over \( P \). Now, first, we can separate the “positive effects” – which occur irrespectively of the start state – out and get

\[
\bigcap_e G_e \subseteq \bigcap_{s_0} (s_0 \setminus \text{eff}(p)) \cup \bigcap_p \text{eff}(p).
\]

Further, we can re-write \( \bigcap_{s_0,p} (s_0 \setminus \text{eff}(p)) \) to \( \bigcap_{s_0,p} (s_0 \cap L(p)) \) where \( L(p) \) is the complement of \( \text{eff}(p) \). We can re-write \( \bigcap_{s_0,p} (s_0 \cap L(p)) \) to \( (\bigcap_{s_0} s_0) \cap (\bigcap_p L(p)) \).

Hence, overall, we have derived that

\[
\bigcap_e G_e \subseteq (\bigcap_{s_0} s_0) \cap (\bigcap_p L(p)) \cup \bigcap_p \text{eff}(p).
\]

In the same way, we can derive

\[
\bigcap_e G'_e \subseteq (\bigcap_{s_0} s_0') \cap (\bigcap_p L(p)) \cup \bigcap_p \text{eff}(p),
\]

where \( s'_0 \) ranges over the start states of \( G' \). We need to prove that \( \bigcap_{s_0} s_0 = \bigcap_{s'_0} s'_0 \). Replacing both sides of the equation with the expressions we have just derived, the terms concerning \( p \) 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.

\[\text{Lemma 6.}\] Let \( G = (N, E, O, A) \) be an executable basic annotated process graph without effect conflicts. Let \( E_0 \subseteq E \) be a set of edges so that there exists a reachable state \( s \) 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 reachable state \( s' \) where \( s' \neq l \) and \( t_{s'}(e) > 0 \). Then, there exists a reachable state \( s_0 \) where \( s_0 \neq l \) and, for all \( e \in E_0 \), \( t_{s_0}(e) > 0 \).

\textbf{Proof:} Let \( l \) be an arbitrary literal. We prove that the claim holds for all possible \( E_0 \), by induction over the nodes \( n \in N \). As the induction base case, we prove that the claim holds for every set \( E_0 \) that contains the outgoing edge of the start node, \( e_0 \). As the inductive step, we prove that, for every node \( n \), if the claim holds for every \( E_0 \) containing one of \( n \)’s ingoing edges, then the claim holds for every \( E_0 \) containing one of \( n \)’s outgoing edges. This proves the overall claim: assume there exists a set \( E_0 \) for which the claim does not hold (with \( l \)). Then, by the definition of process graphs and by contraposition to the inductive step, there must exist a node \( n \) so that \( \text{OUT}(n) \cap E_0 \neq \emptyset \) and there exists a set \( E_1 \) for which the claim does not hold and where \( \text{IN}(n) \cap E_1 \neq \emptyset \) (if that was not the case, then we could prove the claim for \( E_0 \) using the inductive step). Iterating the argument, we get a contradiction to the induction base case.

\textbf{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.

\textbf{Inductive case.} As stated, the inductive hypothesis is that the claim holds for every \( E_0 \) containing one of \( n \)’s ingoing edges. We prove that, under this hypothesis, the
claim holds for every $E_0$ containing one of $n$’s outgoing edges. To avoid clumsiness of language, we will use the following conventions. Whenever we write “$E_0$”, we mean a set of edges with $E_0 \cap IN(n) \neq \emptyset$ for which the prerequisite of the claim holds: there exists a reachable state $s$ where, for all $e \in E_0$, $t_s(e) > 0$; and, for each single $e \in E_0$, there exists a reachable state $s'$ where $s' \not\in l$ and $t_{s'}(e) > 0$. Similarly, whenever we write “$E_0'$”, we mean a set of edges with $E_0' \cap OUT(n) \neq \emptyset$ for which the prerequisite of the claim holds. We distinguish the different kinds of nodes:

1. $n \in \mathcal{N}_T$. We distinguish three cases:

   (a) $l \in \text{eff}(n)$. This case is trivial because no $E_0'$ exists. Assume the opposite was the case. Then there exists a reachable state $s'$ where $t_{s'}(OUT(n)) > 0$ and $s' \not\in l$. 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) $\neg l \in \text{eff}(n)$. Let $E_0'$ be an arbitrary set of edges, with $OUT(n) \in E_0'$, and so that there exists a reachable state $s$ where $t_s(e) > 0$ for every $e \in E_0'$. In order to reach $s$, $n$ must be executed. Now, parallel nodes can be executed in an arbitrary order; this follows trivially from the proof of Lemma 1. Hence, on the execution path leading to $s$, we can re-order the execution of the nodes with outgoing edges $E_0'$ (if necessary) so that $n$ comes last. By prerequisite, $n$ is executable, and so we can execute it at this point. Obviously, in the resulting state $s_0$, $l$ is false. Hence the claim holds for $E_0'$, and we are done.

   (c) $\{l, \neg l\} \cap \text{eff}(n) = \emptyset$. For this case, we prove that there is a 1-to-1 correspondence between the sets $E_0$ and the 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 trivially: by induction hypothesis, we know that there exists a reachable state $s_0$ where $s_0 \not\in l$ 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 1-to-1 correspondence. Let $E_0'$ be a set of edges with $E_0' \cap OUT(n) \neq \emptyset$ so that: there exists a reachable state $s$ where, for all $e \in E_0'$, $t_s(e) > 0$; and, for each single $e \in E_0'$, there exists a reachable state $s'$ where $s' \not\in l$ 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 state $s$ is obvious. Regarding the existence of the state $s'$ with $s' \not\in l$ and $t_{s'}(OUT(n)) > 0$, there are two possible reasons for that. First, there exists a state $s''$ with $s'' \not\in l$ 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.

2. $n \in \mathcal{N}_{XS}$. There is a 1-to-1 correspondence between the sets $E_0'$ and the 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 is obvious regarding parallelity, 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 \( s_0 \) 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.

3. \( n \in N_{PS}. \) In this case, new parallelity is introduced, so that a 1-to-1 corre-
dence between the sets \( E_0' \) and the sets \( E_0 \) no longer exists. Instead, we have a
1-to-n correspondence: every set \( E_0' \) can be constructed from a set \( E_0 \) as
\( E_0' = E_0 \setminus \{IN(n)\} \cup E', \) where \( E' \subseteq OUT(n). \) This correspondence can be proved
exactly as before. By induction hypothesis we know that there exists a reachable
state \( s_0 \) 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.

4. \( n \in N_{XJ}. \) Like for xor splits, we have a 1-to-1 correspondence between the sets
\( E_0' \) and the 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 dual to parallel splits: now there is a n-to-1 corre-
dence between the sets \( E_0' \) and the sets \( E_0; \) 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. This concludes the argument.

Lemma 7. Let \( G = (N, E, O, A) \) be a basic annotated process graph without effect
conflicts. There exists exactly one \( I \)-propagation result \( I^*. \) With fixed arity, the time
required to compute \( I^* \) is polynomial in the size of \( G. \)

Proof: We first show uniqueness of \( I^*. \) Assume to the contrary that \( I^* \) is not
unique for \( \{I_1, I_2\}, \) where \( I_1 \) and \( I_2 \) are \( I \)-propagation results. Say \( e \) is the outgoing edge of
node \( n. \) Since \( I \)-propagation sets the value of an edge exactly once, and since the value is a
function of \( n \) ’s annotation (in case of task nodes) and the values of the ingoing edges, we
can conclude that, for some edge \( e' \in IN(n), \) we have \( I_1(e') \neq I_2(e'). \) Obviously, we
can iterate this argument. Since the graph is acyclic, we hence obtain \( I_1(e_0) \neq I_2(e_0), \)
which is a contradiction and disproves the assumption.

With fixed arity, the number of different literals \( |P(n)| \) is polynomial in the size of
\( G; \) with binary \( T, \) \( T \) for any set \( L \) of literals can be computed in \( O(\|P(n)\|^2), \) so \( \epsilon(n) \)
can be pre-computed for every relevant \( n \) in time \( O(|N| + |P(n)|^2). \) Hence an upper
bound on the time required for computing \( I^* \) is \( O(|N| + |P(n)|^2 + |N| + |P(n)| + |E|). \)

Lemma 8. Let \( G = (N, E, O, A) \) be an executable basic annotated process graph
without effect conflicts. Then, for all \( e \in E, I^*(e) = \emptyset. \)
**Proof:** Since \( G \) is executable and basic, we can apply Lemma 5. That is, we can compile the binary ontology into extended action effects without affecting the sets \( \bigcap \). Hence in what follows we can assume without loss of generality that the ontology is empty.

The proof is by induction over the nodes \( N \). The base case of the induction shows the claim for the outgoing edge of the start node \( n_0 \). The inductive case shows, for any node \( n \), that the claim holds for \( n \)’s outgoing edges provided it holds for \( n \)’s incoming edges. Obviously, this proves the overall claim.

The induction base case is obvious since \( I^*(e_0) = \text{eff}(n_0) \). For the inductive case, we distinguish the different kinds of nodes. We use the notation \( L^e := \bigcup_{n' \in N'} \neg \text{eff}(n') \) where \( N' = \{ n' \in N_T \mid M^* \#(e) \#(I_N(n')) = 1 \} \). By Lemma 1, we know that \( N' \) is exactly the set of task nodes \( n' \) for which there exists a reachable state with a token on both \( e \) and \( I_N(n') \).

1. \( n \in N_{PS} \cup N_{XS} \). By construction we have that, for all \( e \in OUT(n), I^*(e) = I^*(I_N(n)) \setminus L^e \). We show that, for all \( e \in OUT(n), \bigcap = \bigcap I_N(n) \setminus L^e \). With the induction hypothesis, this shows the claim. Let \( e \in OUT(n) \) be arbitrary.

First, say \( l \in \bigcap I_N(n) \setminus L^e \). This means that any reachable state with a token on \( I_N(n) \) satisfies \( l \), and that no task node disvalidating \( l \) can be activated in a state with a token on \( e \). Clearly, it follows that any reachable state with a token on \( e \) satisfies \( l \). Hence \( \bigcap \subseteq \bigcap I_N(n) \setminus L^e \).

For the other direction, let’s first assume that \( l \notin \bigcap I_N(n) \). Then there exists a reachable state \( s \) with a token on \( I_N(n) \) and \( s \not\models l \). By executing \( n \), we can obtain a state with \( s \models l \), and with a token on \( e \) for parallel splits the latter is obvious, for xor splits it is true because \( G \) is basic and hence no edge conditions are annotated.

We get \( \bigcap \subseteq \bigcap I_N(n) \). Finally, assume that \( l \in L^e \). Then there exists a state \( s \) with tokens on both \( e \) and \( I_N(n') \) where \( \neg l \in \text{eff}(n') \). Since \( n' \) is executable by prerequisite (its precondition is guaranteed to be satisfied in \( s \)), we can execute it in \( s \), and thus obtain a state with a token on \( e \) and \( s \not\models l \). Hence \( \bigcap \subseteq \bigcap I_N(n) \setminus L^e \).

2. \( n \in N_c \). By construction we have that \( I^*(OUT(n) = \text{eff}(n) \cup (I^*(I_N(n)) \setminus \neg \text{eff}(n)) \). We show that \( \bigcap OUT(n) = \text{eff}(n) \cup (\bigcap I_N(n) \setminus \neg \text{eff}(n)) \). With the induction hypothesis, this shows the claim.

First, say \( l \in \text{eff}(n) \cup (\bigcap I_N(n) \setminus \neg \text{eff}(n)) \). Say \( s \) is a reachable state with a token on \( OUT(n) \). If \( s \) is directly reached from a state \( s' \) with a token on \( I_N(n) \), then it is obvious from the execution semantics that \( s \models l \). If other transitions were executed in between, then the case must be considered that \( l \) was true in \( s' \), but was falsified by one of the other transitions. However, for that to happen there must exist a task node \( n' \) parallel to \( n \) that falsifies \( l \). This would be an effect conflict, in contradiction to our prerequisite. Hence \( \bigcap OUT(n) \supseteq \text{eff}(n) \cup (\bigcap I_N(n) \setminus \neg \text{eff}(n)) \).

For the other direction, let’s first assume that \( l \in \neg \text{eff}(n) \). Obviously, directly after executing \( n \), \( l \) is false. If another task node \( n' \) were to make \( l \) true while there still is a token on \( OUT(n) \), then \( n' \) would have to be parallel to \( n \), implying an effect conflict in contradiction to the prerequisite. Hence \( \bigcap OUT(n) \cap \neg \text{eff}(n) = \emptyset \).

Second, assume that \( l \) is neither contained in \( \text{eff}(n) \), nor contained in \( \bigcap I_N(n) \). Due to \( l \notin \bigcap I_N(n) \), we have a reachable state \( s \) with a token on \( I_N(n) \) and \( s \not\models l \).
Due to $l \not\in \text{eff}(n)$, if we execute $n$ in $s$ (which we are guaranteed to be able to do because by prerequisite $n$ is executable) then we reach a state $s'$ with a token on $\text{OUT}(n)$ and $s' \not\models l$. Hence $\bigcap_{e \in \text{OUT}(n)} \text{eff}(n) \cup \bigcup_{e \in \text{IN}(n)}$, and altogether $\bigcap_{e \in \text{OUT}(n)} \subseteq \text{eff}(n) \cup (\bigcap_{e \in \text{IN}(n)} \neg \text{eff}(n))$—note here that, because effects are not self-contradictory, $\text{eff}(n) \cap \neg \text{eff}(n) = \emptyset$.

3. $n' \in \mathcal{N}_{P,I}$. By construction we have that $I^*(\text{OUT}(n)) = \bigcup_{e \in \text{IN}(n)} I^*(e) \setminus \text{OUT}(n)$. We show that $\bigcap_{e \in \text{OUT}(n)} = \bigcup_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$. With the induction hypothesis, this shows the claim. (Note that $\bigcup_{e \in \text{IN}(n)} \bigcap_e$ cannot contain any contradictory literals, since all the incoming edges of $n$ may carry a token at the same time: a state where this is true satisfies the conjunction of all the $e$.)

First, let $e \in \text{IN}(n)$ be arbitrary and let $l$ be a literal so that $l \in \bigcap_e \setminus \text{OUT}(n)$. This means that any reachable state with a token on $e$ satisfies $l$, and that no task node disvalidating $l$ can be activated in a state with a token on $\text{OUT}(n)$. Clearly, it follows that any reachable state with a token on $\text{OUT}(n)$ satisfies $l$. Hence $\bigcap_{e \in \text{IN}(n)} \supseteq \bigcup_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$.

For the other direction, let’s first assume that $l \in \bigcup_{e \in \text{OUT}(n)}$. Then there exists a state $s$ with tokens on both $\text{OUT}(n)$ and $\text{IN}(n')$ where $\neg l \in \text{eff}(n')$. Since $n'$ is executable by prerequisite, we can execute it in $s$, whereby we obtain a state with a token on $\text{OUT}(n)$ and $s' \not\models l$. Hence $\bigcap_{e \in \text{OUT}(n)} \cap \text{OUT}(n) = \emptyset$. Finally, let’s assume that, for all $e \in \text{IN}(n), l \not\in e$. Then, for each $e \in \text{IN}(n)$, there exists a reachable state $s'$ with a token on $e$ and $s' \not\models l$. We can thus apply Lemma 6, and obtain a state $s_0$ with $s_0 \not\models l$ and tokens on all $e \in \text{IN}(n)$. By executing $n$ in $s_0$, we reach a state $s''$ with a token on $\text{OUT}(n)$ and $s'' \not\models l$. Hence $\bigcap_{e \in \text{OUT}(n)} \subseteq \bigcup_{e \in \text{IN}(n)} \bigcap_e$, and altogether $\bigcap_{e \in \text{OUT}(n)} \subseteq \bigcup_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$.

4. $n' \in \mathcal{N}_{X,I}$. By construction we have that $I^*(\text{OUT}(n)) = \bigcap_{e \in \text{IN}(n)} I^*(e) \setminus \text{OUT}(n)$. We show that $\bigcap_{e \in \text{OUT}(n)} = \bigcap_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$. With the induction hypothesis, this shows the claim.

First, let $l$ be a literal so that $l \in \bigcap_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$. This means that any reachable state with a token on any $e \in \text{IN}(n)$ satisfies $l$, and that no task node disvalidating $l$ can be activated in a state with a token on $\text{OUT}(n)$. Clearly, it follows that any reachable state with a token on $\text{OUT}(n)$ satisfies $l$. Hence $\bigcap_{e \in \text{OUT}(n)} \supseteq \bigcap_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$.

For the other direction, let’s first assume that $l \in \bigcup_{e \in \text{OUT}(n)}$. Then there exists a state $s$ with tokens on both $\text{OUT}(n)$ and $\text{IN}(n')$ where $\neg l \in \text{eff}(n')$. Since $n'$ is executable by prerequisite, we can execute it in $s$, whereby we obtain a state with a token on $\text{OUT}(n)$ and $s' \not\models l$. Hence $\bigcap_{e \in \text{OUT}(n)} \cap \text{OUT}(n) = \emptyset$. Finally, let’s assume that $l \not\in \bigcap_{e \in \text{IN}(n)} \bigcap_e$, i.e., there exists $e \in \text{IN}(n)$ with $l \not\in e$. This means that there exists a reachable state $s$ with a token on $e$ and $s' \not\models l$. By executing $n$, we obtain a state with a token on $\text{OUT}(n)$ and $s' \not\models l$. Hence $\bigcap_{e \in \text{OUT}(n)} \subseteq \bigcap_{e \in \text{IN}(n)} \bigcap_e$, and altogether $\bigcap_{e \in \text{OUT}(n)} \subseteq \bigcup_{e \in \text{IN}(n)} \bigcap_e \setminus \text{OUT}(n)$. This concludes the argument.
Theorem 3 Let $G = (N, E, O, A)$ be a basic annotated process graph without conflict. There exists exactly one I-propagation result $I^*$. $G$ is executable iff, for all $n \in N \cup \{n_+\}$, $\text{pre}(n) \subseteq I^*(\text{IN}(n))$. With fixedarity, the time required to compute $I^*$ is polynomial in the size of $G$.

Proof: Recall that a node $n$ is executable iff, for all reachable states $s$ so that $tb_s(\text{IN}(n)) > 0$, 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. $G$ is executable if all its nodes are executable.

Obviously, a node $n$ is executable iff $\text{pre}(n) \subseteq \bigcap I^*(\text{IN}(n))$.

Say $G$ is executable. By Lemma 8 we know that $I^*(e) = \bigcap e$ for all $e \in E$. Hence it follows that $\text{pre}(n) \subseteq I^*(\text{IN}(n))$, showing the direction from left to right.

For the other direction, let $N \subseteq N'$ be the nodes in $G$ that are not executable. If $N = \emptyset$ then there is nothing to show. Else, let $n \in N$ be a node where all predecessors of $n$ in $G$, i.e., all nodes $n'$ with a path to $n$, are executable. Since the graph is acyclic, such a node $n$ exists. By the same arguments as used for proving Lemma 8, it follows that $I^*(e) = \bigcap e$ for all edges $e$ connected to the predecessors $n'$, in particular $I^*(\text{IN}(n)) = \bigcap I^*(\text{IN}(n))$. Hence, if $\text{pre}(n) \subseteq I^*(\text{IN}(n))$, then $\text{pre}(n) \subseteq \bigcap I^*(\text{IN}(n))$ and $N$ is executable in contradiction. Hence $\text{pre}(n) \not\subseteq I^*(\text{IN}(n))$, which proves the claim.

The rest of the claim follows trivially by Lemmas 5 and 7. ■

B.2 Computational Borderline

Theorem 1 Assume an annotated process graph $G = (N, E, O, A)$ that is basic except that $T$ is not binary. Deciding whether $G$ is reachable is $\Sigma^P_2$-hard in general, and $\text{NP}$-hard if $T$ is Horn. Deciding whether $G$ is executable is $\Pi^P_2$-hard in general, and $\text{coNP}$-hard if $T$ is Horn.

Proof: Let us first consider the general case, with no restrictions on $T$. The proofs are by reduction of validity of a QBF formula $\forall X.\exists Y.\phi[X, Y]$, where $\phi$ is in CNF. The process graphs $G$ 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 $\text{pre}(n_t) = \emptyset$. The main trick of the proof lies in the definitions of $O$, $\text{eff}(n_0)$, and $\text{eff}(n_t)$. Those are adapted from the constructions used in the proofs of Lemma 6.2 from [16]. The predicates $P$ of $O$ are all 0-ary, i.e., they have no arguments and are hence logical facts. 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 $\text{eff}(n_0)$ 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 $s_0$. We define $\text{eff}(n_t)$ to be $\{t\}$. The complex part of the construction lies in the theory $T$ of $O$. We define $T :=$

$$(\bigwedge_{i=1}^m (\neg t \lor x_i \lor z_i)) \land (\bigwedge_{i=1}^m (\neg t \lor \neg x_i \lor \neg z_i)) \land (\bigwedge_{C \in \phi} (\neg t \lor \neg q \lor C)) \land (\bigwedge_{i=1}^n (\neg t \lor \neg y_i \lor q))$$

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

$t \Rightarrow ((\bigwedge_{i=1}^m x_i \equiv \neg z_i) \land (q \Rightarrow \phi) \land ((\bigvee_{i=1}^n y_i) \Rightarrow q))$
Note that \(\text{eff}(n_t)\) is consistent with the theory: any interpretation that sets \(r\) and all \(y_i\) to 0 satisfies \(T \land \text{eff}(n_t)\). 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 \(a\) to \(x\), there exists a state \(s \in S\) that complies with \(a_x\) if \(a_X\) is satisfiable together with \(T \land \text{eff}(n_t)\), 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 \(\phi\) 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 \((\bigvee_{i=1}^m y_i) \Rightarrow q\), 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 \(i_{s'} \models T \land \text{eff}(n_t)\). But then, \(s'\) is closer to \(s_0\) than \(s\), and hence \(s \not\in S\) in contradiction. This concludes the argument for \((^*)\).

To prove \(\Pi_2^n\)-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, the claim follows.

To prove \(\Sigma_2^n\)-hardness of deciding reachability, an only slightly more complex construction is required. We introduce another node \(n_{\neg q} \in \mathcal{N}\), 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 \(T\) is restricted to be Horn. The graphs \((\mathcal{N}, \mathcal{E})\) 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 [16]. 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 \(\text{eff}(n_t)\) to contain all \(x_i\), all \(y_i\), all \(\neg z_i\), \(\neg q\), and \(\neg t\). The theory is:

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

where \(\phi\) 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:

\[
\forall t \in \mathcal{N} \quad (\exists i \in \mathbb{N} : n_i \in \mathcal{N}) \quad t \Rightarrow \bigg[ \big( \bigwedge_{i=1}^{n} z_i \big) \Rightarrow q \bigg] \land \left( \bigg( \bigwedge_{i=1}^{n} (\neg x_i \lor y_i) \land (x_i \Rightarrow z_i) \land (y_i \Rightarrow z_i) \bigg) \right) \land \bigg( \bigwedge_{C \in \phi} C[-Y/ + X] \bigg)
\]

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 \( T \land \text{eff}(n_t) \). Hence \( n_t \) complies with Definition 3.

The key in this transformation is that \( \phi \) 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 if \( C \) 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 \( \phi \) is satisfiable. Let \( L_0 \) be the set of literals on whose interpretation \( s \) agrees with \( s_0 \). We can conclude that \( T \land \text{eff}(n_t) \land \bigwedge_{l \in L_0} \neg q \) is unsatisfiable, since otherwise we can construct a state \( s' \) that has \( s' \models L_0 \land \neg q \) and that is hence closer to \( s_0 \) than \( s \). The only part of \( T \land \text{eff}(n_t) \) that forces implication of \( q \) is \( \big( \bigwedge_{i=1}^{n} z_i \big) \Rightarrow q \). Thus we infer that \( T \land \text{eff}(n_t) \land \bigwedge_{l \in L_0} \neg q \models \bigwedge_{i=1}^{n} z_i \). The only part of \( T \land \text{eff}(n_t) \) 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 \( T \land \text{eff}(n_t) \land \bigwedge_{l \in L_0} \neg q \). 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 \( \phi \).

For the other direction, assume that \( \phi \) 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 \( i \), \( i_s \models x_i \) iff \( a_X(x_i) = 1 \). Then, we set that for all \( y_i \), \( i_s \models y_i \) iff \( a_X(y_i) = 0 \). 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 \( i_s \models T \land \text{eff}(n_t) \). \( \phi \) 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 \( z_i \), 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 \( \Pi^2_2 \)-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, the claim follows.

To prove \( \Sigma^2_2 \)-hardness of deciding reachability, we introduce another node \( n_q \in \mathcal{N} \), 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.
Theorem 2 Assume an annotated process graph \( \mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A}) \) that is basic except that \( \mathcal{A}(e) \) may be defined for some \( e \in \mathcal{E} \). Deciding whether \( \mathcal{G} \) is reachable is \( \text{NP-hard} \). Deciding whether \( \mathcal{G} \) is executable is \( \text{coNP-hard} \).

Proof: The proof for reachability checking is by the following reduction from 3SAT. Assume a CNF \( \phi \) with \( n \) propositions \( p_1, \ldots, p_n \), and \( k \) clauses \( c_1, \ldots, c_k \) where \( c_i = \{l_{i1}, l_{i2}, l_{i3}\} \). We obtain a basic annotated process graph with choice, \( (\mathcal{N}, \mathcal{E}, \mathcal{P}, \mathcal{A}) \), as follows.

The set of nodes, \( \mathcal{N} \), and their annotation, is:
1. start node \( n_0 \); \( \mathcal{A}(n_0) = (\emptyset, \emptyset) \)
2. parallel split node \( n_{ps} \)
3. xor-split nodes \( n_{xs_1} \ldots n_{xs_n} \)
4. for \( 1 \leq i \leq n \): task nodes \( n_{p_i} and n_{notp_i}; \mathcal{A}(n_{p_i}) = (\emptyset, \{p_i\}), \mathcal{A}(n_{notp_i}) = (\emptyset, \{notp_i\}) \)
5. xor-join nodes \( n_{xj_1} \ldots n_{xj_n} \)
6. parallel join node \( n_{pj} \)
7. for \( 1 \leq i \leq k \): xor-split node \( n_{xs_i}' \)
8. for \( 1 \leq i \leq k - 1 \): xor-join node \( n_{xj_i}' \)
9. task node \( n_g; \mathcal{A}(n_g) = (\emptyset, \emptyset) \)
10. xor-join node \( n_{xj_1}' \)
11. stop node \( n_+; \mathcal{A}(n_+) = (\emptyset, \emptyset) \)

The set of edges, \( \mathcal{E} \), and their annotation, is:
1. \( (n_0, n_{ps}) \)
2. for \( 1 \leq i \leq n \): \( (n_{ps}, n_{xs_i}) \)
3. for \( 1 \leq i \leq n \): \( (n_{xs_i}, n_{p_i}) and (n_{xs_i}, n_{notp_i}) \); these edges are not annotated
4. for \( 1 \leq i \leq n \): \( (n_{p_i}, n_{xj_i}) \) and \( (n_{notp_i}, n_{xj_i}) \)
5. for \( 1 \leq i \leq n \): \( (n_{xj_i}, n_{pj}) \)
6. \( (n_{pj}, n_{os_1}) \)
7. for $1 \leq i \leq k$: \((nsx'_i, nxj')\); \(A((nsx'_i, nxj')) = \{-l_{i1}, -l_{i2}, -l_{i3}\}\)

8. for $1 \leq i \leq k - 1$: for $1 \leq j \leq 3$: \((nsx'_i, nxj'_i); A((nsx'_i, nxj'_i)) = \{l_{ij}\}\)

9. for $1 \leq i \leq k - 1$: \((nxj'_i, nsx'_{i+1})\)

10. for $1 \leq j \leq 3$: \((nsx'_k, ng); A((nsx'_k, ng)) = \{l_{kj}\}\)

11. \((ng, noj)\)

12. \((noj, n_+)\)

Obviously, \(G\) is reachable – i.e., all nodes including in particular \(ng\) are reachable – iff \(\phi\) is satisfiable. Also, the annotation of the start node can be set to be \(A(n_0) = (\emptyset, \{\neg p_1, \neg not p_1, \ldots, \neg p_n, \neg not p_n\}\), and hence to be complete. Finally, 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 \(\phi\), let \(p\) be a new proposition; obtain \(\phi'\) by inserting \(p\) into every clause of \(\phi\). Then construct, for \(\phi'\), the process graph as above, with the only difference being that \(ng\) has the annotation \(A(ng) = (\{p\}, \emptyset)\). With this construction, we have that 1. \(ng\) is reachable (trivially, by making \(p\) true and choosing the \(p\)-branch for every clause); and 2. \(ng\) is executable iff every satisfying assignment to \(\phi'\) makes \(p\) true. The latter is, obviously, the case iff \(\phi\) is unsatisfiable. Since \(ng\) is the only node with a precondition, all other nodes are trivially executable and hence the claim follows. \(\blacksquare\)