# Beyond Soundness: On the Semantic Consistency of Executable Process Models

Ingo Weber, Jörg Hoffmann
SAP Research Karlsruhe, Germany
first.last@sap.com

Jan Mendling
Humboldt-Universität zu Berlin, Germany,
jan.mendling@hu-berlin.de

## Abstract

*Executable business process models 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 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. The paper is written at a semi-formal level using an illustrative example, details can be looked up in a longer technical report.*

## 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 axiomations; 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 tech-

niques 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.

For the sake of readability, we choose a semi-formal presentation style throughout the paper, illustrating our techniques with an example annotated BPMN process model. Formal details are extensively discussed in a technical report (TR) where we define process models and their executional semantics [19]. There, we present our analysis methods in full technical detail and prove correctness relative to the semantics. We also prove formally that our analysis methods run in polynomial time, and that richer classes of processes – with more complex semantic annotations – cannot be analyzed in polynomial time (unless **P=NP**). We include these formal results in this paper but refer the reader to [19] for the proofs.

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.

## 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 [20]. 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.

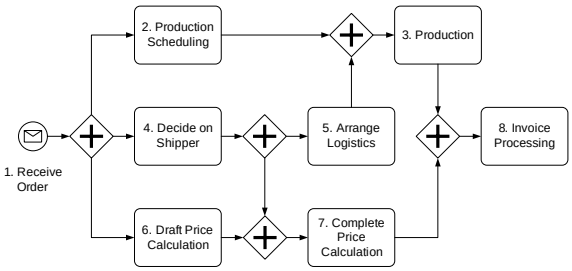


**Figure 1. Example of a sales order process modeled in BPMN.**

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 beed 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 ($sh$).

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

| # | Precondition | Postcondition |
|---|---|---|
| 1 | | orderReceived(o) |
| 2 | orderReceived(o)<br>orderApproved(o) | productionScheduled(o,p) |
| 3 | productionScheduled(o,p)<br>calculationPrepared(o,c) | productionCompleted(o,p)<br>calculationUpdated(o,c) |
| 4 | orderReceived(o) | shipperDecided(o,s) |
| 5 | shipperDecided(o,s)<br>calculationPrepared(o,c) | calculationUpdated(o,c)<br>shipmentApproved(o,sh) |
| 6 | orderReceived(o) | calculationDrafted(o,c) |
| 7 | calculationPrepared(o,c) | calculationCompleted(o,c) |
| 8 | productionCompleted(o,p)<br>calculationCompleted(o,c) | orderCompleted(o) |

| Object | Definition |
|---|---|
| Order | o is at most one of received or completed |
| Production | p is at most one of scheduled or completed |
| Calculation | if c is drafted, then c is prepared<br>if c is updated, then c is prepared<br>c is at most one of drafted, updated, completed<br>c is at most one of prepared or completed |
| Shipment | if shipment sh is approved, then o is approved |

**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.**

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).

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. For a formal presentation please refer to [19].

We define a process model as a *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 *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. Further, each task node $n$ can be annotated using *preconditions* ($\text{pre}_n$) and *effects* ($\text{eff}_n$, also referred to as *postconditions*), which are conjunctions of literals using the process variables. The task can only be executed if $\text{pre}_n$ is true; $n$ is *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. For full details refer to [19].

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 $P = \{orderReceived(x), orderApproved(x), \ldots, shipmentApproved(x, y)\}$. The theory $T$ consists of the formulae shown in Table 2, formalizing their intuitive counterparts from 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 *orderReceived(o)*. By $T$, this has the implicit effect that *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, *orderReceived(o)*, is definitely true. Their effects are *calculationDrafted(o,c)*, which implies *calculationPrepared(o,c)*, $\neg$*calculationUpdated(o,c)*, and $\neg$*calculationCompleted(o,c)*; and *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 *orderReceived(o)* and *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 *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, *calculationPrepared(o,c)* and *shipperDecided(o,s)*, are true in all possible worlds. After the execution, due to the explicit effects all worlds will satisfy *calculationUpdated(o,c)* and *shipmentApproved(o,sh)*. By $T$, via the axiom $\forall x, y$ : *shipmentApproved(x,y)* $\implies$ *orderApproved(x)*, the latter involves the implicit effect *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 *calculationUpdated(o,c)*, the latter has the explicit effect *calculationCompleted(o,c)*.

| Purpose | Definition |
|---|---|
| Order status | $\forall x$ : ¬orderReceived(x) ∨¬orderCompleted(x) |
| Production status | $\forall x, y$ : ¬productionScheduled(x,y) ∨¬productionCompleted(x,y) |
| Calculation status | $\forall x, y$ : calculationDrafted(x,y) $\implies$ calculationPrepared(x,y)<br>$\forall x, y$ : calculationUpdated(x,y) $\implies$ calculationPrepared(x,y)<br>$\forall x, y$ : ¬calculationDrafted(x,y) ∨¬calculationCompleted(x,y)<br>$\forall x, y$ : ¬calculationUpdated(x,y) ∨¬calculationCompleted(x,y)<br>$\forall x, y$ : ¬calculationDrafted(x,y) ∨¬calculationUpdated(x,y)<br>$\forall x, y$ : ¬calculationPrepared(x,y) ∨¬calculationCompleted(x,y) |
| Order approval | $\forall x, y$ : shipmentApproved(x,y) $\implies$ orderApproved(x) |

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

These effects are not in immediate conflict; however, $T$ tells us that $\forall x, y$ : *¬calculationUpdated(x,y) ∨¬calculationCompleted(x,y)*, implying that an implicit effect of *Production* is *¬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* $\mathbf{\Pi_2^p}$*-hard for unrestricted* $T$*, and* **coNP***-hard if* $T$ *consists of Horn clauses only. This holds even if predicate arity is fixed to* 0*.*

As stated, all proofs are available in [19]. 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* **coNP***-hard. This holds even if predicate arity is fixed to* 0*, and* $T$ *is empty.*

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. Polynomial-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, 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 [21]. 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 $\overline{\text{eff}_n}$.

### 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 $M_i^j$, 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 $i < j$. Our TR [19] shows how to generate such a numbering; it also contains 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.

| | Receive Order | Production Scheduling | Decide on Shipper | Draft Price Calculation | Arrange Logistics | Production | Complete Price Calculation | Invoice Processing |
|---|---|---|---|---|---|---|---|---|
| Receive Order | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Production Scheduling | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| Decide on Shipper | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Draft Price Calculation | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| Arrange Logistics | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| Production | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Complete Price Calculation | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| Invoice Processing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 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 $M_i^k = M_j^k$ for all $k < j$. Second, the outgoing edges of a parallel split introduce new parallelism. This is covered simply by setting $M_i^j = 1$ 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 $M_i^j = 0$ 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 $M_i^j$ 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 $M_i^j$ 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-propa-gation 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 $M_i^j = 1$.*

This is proved in our TR [19] along the lines of the arguments above. 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 $\overline{\text{eff}}$ is the union of 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 $n, n'$ with incoming edges $e_i, e_j$: $n$ and $n'$ have a precondition (effect) conflict iff $M_i^j = 1$ and there exists a literal $l$ s.t. $l \in \overline{\text{eff}}_n$ and $\neg l \in \text{pre}_{n'}$ ($\neg l \in \overline{\text{eff}}_{n'}$). 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 *Complete Price Calculation* to come after *Production*, i.e., as a second last step just before *Invoice Processing*.

In what follows, we explain I-propagation at a semi-formal level; as mentioned before, details can be looked up in [19]. 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

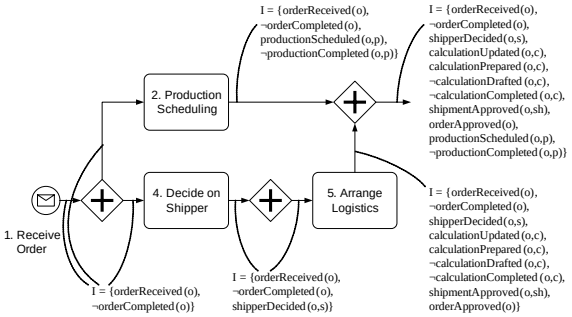


**Figure 3. Outcome of I-propagation on a part of the example process from Fig. 1.**

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 $\overline{\text{eff}}_{n_0}$ for the start node $n_0$, and to $\perp$ 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 *Receive Order* and behind *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 ingoing 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 $l$ 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 ingoing 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 $\overline{\text{eff}}_n$, 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 $\overline{\text{eff}_n}$.

(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 $\neg p, \neg q, \neg r$; $p, q \in I(e)$; $\overline{\text{eff}_n} = \{r\}$. Then, after $n$, neither $p$ nor $q$ are guaranteed to hold (although their opposites are not contained in $\overline{\text{eff}_n}$). 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.

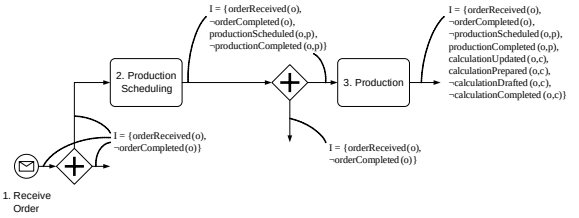


**Figure 4. An illustration of action (3) for task nodes, using a variant of Fig. 3.**

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 $\overline{\text{eff}_n}$ ($l$ cannot be established by the effect of a task node connected to $e''$ since that would be an effect conflict). In this situation, $l$ is *not* guaranteed to hold whenever $e''$ carries a token: $n$ may be fired, leading to $\neg l$. This is best understood using an example. Consider Fig. 4. The task node $n$ we consider is *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 *productionScheduled(o,p)* and *¬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, *productionScheduled(o,p)* and *¬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. See the full proof in [19].

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 [22] 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 [23]. Our approach uses not only annotations in terms of preconditions and effects but also an ontology. In that sense, [22] might be regarded as a special case of our framework. In the area of *access control* the approach of [24] 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 [25] 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 [25] covers some practical special cases. The paper by [26] addresses a.o. *life cycle compliance*. This can be partly reformulated in terms of preconditions, effects, and ontological axioms. Our running example illustrates some constraints related to the life cycle of business objects.

In [27], 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, [27] 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 [28] 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 [28] does not consider ontologies.

The most closely related work to our approach is [29]. 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. [29] 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, [29] 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, the work of [29] proceeds at a rather informal 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 [30]. Data flow analysis builds on compiler theory [31] 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


[1] Aalst, W.: Verification of Workflow Nets. In: Application and Theory of Petri Nets. (1997)

[2] Dehnert, J., Aalst, W.: Bridging The Gap Between Business Models And Workflow Specifications. Int. J. Coop. Inf. Syst. **13** (2004) 289–332

[3] Puhlmann, F., Weske, M.: Investigations on soundness regarding lazy activities. In: Business Process Management (BPM). LNCS 4102 (2006) 145–160

[4] Mendling, J., Aalst, W.: Formalization and Verification of EPCs with OR-Joins Based on State and Context. In: Conf. Adv. Inf. Sys. Engineering (CAiSE). (2007) LNCS 4495.

[5] Verbeek, H., Basten, T., Aalst, W.: Diagnosing Workflow Processes using Woflan. Comp. Journal **44** (2001) 246–279

[6] Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel, D.: Semantic business process management: A vision towards using semantic web services for business process management. In: Proceedings of ICEBE (2005) 535–540

[7] Weber, I., Hoffmann, J., Mendling, J., Nitzsche, J.: Towards a methodology for semantic business process modeling and configuration. In: ICSOC'07 Workshops. LNCS (2008)

[8] A. Ankolekar et al: DAML-S: Web service description for the semantic web. In: ISWC. (2002)

[9] The OWL Services Coalition: OWL-S: Semantic Markup for Web Services (2003)

[10] D. Roman et al.: Web Service Modeling Ontology. Applied Ontology **1** (2005) 77–106

[11] D. Fensel et al: Enabling Semantic Web Services: The Web Service Modeling Ontology. (2006)

[12] Andersson, B., Bergholtz, M., Edirisuriya, A., Ilayperuma, T., Johannesson, P.: A declarative foundation of process models. In: CAiSE. (2005) 233–247

[13] Aalst, W., Hee, K.: Workflow Management. (2002)

[14] Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models though sese decomposition. In: ICSOC. (2007)

[15] Winslett, M.: Reasoning about actions using a possible models approach. In: AAAI. (1988)

[16] Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates, and counterfactuals. Artificial Intelligence **57** (1992) 227–270

[17] Herzig, A., Rifi, O.: Propositional belief base update and minimal change. AI **115** (1999)

[18] Valmari, A.: The state explosion problem. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I. LNCS 1491 (1998) 429–528

[19] Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: On the semantic consistency of executable process models. Technical report (2008), `http://www.imweber.de/texte/tr-ecows08.pdf`.

[20] T. Andrews and et al.: Business Process Execution Language for Web Services (version 1.1) (2003)

[21] Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf. Proc. Letters **8** (1979) 121–123

[22] Ly, L.T., Rinderle, S., Dadam, P.: Semantic correctness in adaptive process management systems. In: BPM. (2006)

[23] Aalst, W., Beer, H., Dongen, B.: Process mining and verification of properties: An approach based on temporal logic. In: OTM Conferences. (2005)

[24] Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM TISSEC **2** (1999)

[25] Namiri, K., Stojanovic, N.: A model-driven approach for internal controls compliance in business processes. In: SBPM. (2007)

[26] Ryndina, K., Küster, J.M., Gall, H.: Consistency of business process models and object life cycles. In: MoDELS Workshops. (2006) 80–90 LNCS 4364.

[27] Meyer, H.: On the semantics of service compositions. In: Web Reasoning and Rule Systems (2007) 31–42

[28] Narayanan, S., McIlraith, S.: Simulation, verification and automated composition of web services. In: World Wide Web Conference. (2002) 77–88

[29] Koliadis, G., Ghose, A.: Verifying semantic business process models in inter-operation. In: SCC. (2007)

[30] Reichert, M., Rinderle, S., Dadam, P.: ADEPT workflow management system: Flexible support for enterprise-wide business processes. In: BPM. (2003)

[31] Aho, A., Sethi, R., Ullman, J.: Compilers: principles, techniques, and tools. Addison-Wesley (1986)