Semantic Business Process Validation

Ingo Weber[1], Jörg Hoffmann[2], and Jan Mendling[3]

[1] SAP Research, Karlsruhe, Germany
ingo.weber@sap.com
[2] University of Innsbruck, STI, Austria
joerg.hoffmann@sti2.at
[3] BPM Group, Queensland University of Technology, Australia
j.mendling@qut.edu.au

Abstract. The use of formal semantics for the support of Business Process Management is an emerging branch of research, with substantial economic potential. In particular, business processes modelled in graphical notations such as BPMN can be semantically annotated to specify more precisely what the individual tasks in the process will be responsible for. This raises the need for, and opens up the opportunity to apply, *semantic validation* techniques: techniques that take the annotations and the underlying ontology into account in order to determine whether the tasks are consistent with respect to each other, and with respect to the underlying workflow structure. To this end, we introduce a formalism for semantic business processes, which combines definitions from the workflow community with definitions from AI; we introduce some validation tasks that are of interest in this context. We then make first technical contributions towards solving this kind of problem. We identify a class of processes where the validation tasks can be solved in polynomial time, by propagating certain pieces of information through the process graphs. We show that this class of processes is maximal in the sense that, with more general semantic annotations, the validation tasks become computationally hard. We outline how the validation information gathered can serve to automatically suggest bug fixes.

1 Introduction

One of the promises of service-oriented architectures is increased flexibility through loose coupling of components. In enterprise applications, Web services can be used to encapsulate business functionality. The process flow can then, to a large degree, be separated from the implementation of the process activities. This increased flexibility can be a key advantage when acting in rapidly changing markets. In such an environment, a business expert models a business process on a high level of abstraction, like the Business Process Modelling Notation (BPMN) [18]. Semantic techniques can then be used to help bridge between the high-level process and the actual IT infrastructure. Namely, the business expert can annotate the tasks in the process (the steps that are atomic from the business expert's point of view) with semantic information relative to an ontology that formalises the underlying business domain. Then, based on a Semantic Web Services (SWS) infrastructure, advanced support – discovery, composition, mediation – for *binding* the process, i.e., for implementing it with concrete services, becomes possible.

Since modelling is an error-prone activity, the above scenario calls for validation techniques: prior to trying to bind the process, validation should help to ensure that the process model as such makes sense. A validation method may be useful *after* binding the process, as well: being implemented at a more detailed level, the bound process may reveal flaws that were not visible at the BPMN level of abstraction. Possible validation questions are, for example: Is there an *effect conflict*, i.e., two tasks with conflicting semantic effects and no ordering constraint between them (simple example: two write-operations in a database)? Is a particular task *reachable*, i.e., is there any execution of the process that involves the task? Is a particular task *executable*, i.e., is its semantic precondition guaranteed to be true when trying to execute it? Answers to these questions are certainly useful as a help for the modeller, and for binding the process; we will see that one can even design techniques that automatically suggest (potential) bug fixes.

Traditional validation techniques, as considered in the workflow community (e.g. [24]) and the model checking community (e.g. [7]), do not deal with ontological axiomatizations, which are at the heart of the semantic annotations required in the above scenario. Hence mapping our validation problem into existing methodologies is, to say the least, problematic. Thus we suggest a different approach, *Semantic Business Process Validation (SBPV)*, which is explicitly designed to deal with this kind of validation tasks. We introduce a formal execution semantics for semantic business processes in the context of axiomatizations, and we identify associated validation tasks. We make first technical contributions towards solving this kind of problem.

Our SBPV formalism is a natural combination of definitions from the workflow community (e.g. [24,25]), and the AI actions and change community (e.g. [27,10,12]). The former deal with the semantics of workflow structures, via a notion of token passing adopted from Petri Nets. The latter deal with the semantics of logical preconditions and effects in the presence of a logical theory – the axioms of the ontology – constraining the behaviour of the underlying domain. In particular, this distinguishes us from all other works [15,6,22,20,21] extending Business Process validation beyond workflows (see Section 6 for details). We formalise the validation tasks outlined above. Thereafter:

- We identify a class of processes, termed *basic* processes, where the validation tasks can be solved in polynomial time. Basic processes have no cycles, no branching conditions, and an ontology consisting of only binary clauses. Naturally, this deliberately restricted class covers only a subset of real-world processes. We specify validation algorithms inspired by the token passing that underlies the workflow semantics; instead of passing only tokens, logical information is propagated. Although the key ideas are straightforward, some of the details are quite intricate.
- We show that the class of basic processes is maximal in the sense that, when allowing more general semantic annotations, then the validation tasks become computationally hard. In particular, this holds already for very simple branching conditions, and for Horn clauses instead of binary clauses.
- We outline methods suggesting bug fixes, based on the information propagated by our validation algorithms. This suggests that, in general, one should try to obtain this information (rather than only "yes/pass" answers to the validation questions).

Section 2 introduces our SBPV formalism. Sections 3 and 4 present our polynomial-time algorithms for basic processes, and our results their computational borderline.

Section 5 outlines how bug fixes can be suggested, Section 6 discusses related work, Section 7 concludes. For space reasons, the presentation is very concise; many technical details (including all proofs), and more illustrative examples, are moved to [26].

2 Semantic Business Processes

We introduce a formalism, i.e., a formal execution semantics, for business processes whose tasks are annotated with logical preconditions and effects (Section 2.1). We then present a number of validation tasks that are of interest for such processes (Section 2.2). We illustrate the formalism with an example annotated BPMN process (Section 2.3).

2.1 Annotated Process Graphs

Our business processes consist of different kinds of nodes (task nodes, split nodes, ...) connected with edges. We will henceforth refer to this kind of graphs as *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 [25].

Definition 1. *A* process graph *is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the disjoint union of $\{n_0, n_+\}$ (*start node*, *stop node*), $\mathcal{N}_T$ (*task nodes*), $\mathcal{N}_{PS}$ (*parallel splits*), $\mathcal{N}_{PJ}$ (*parallel joins*), $\mathcal{N}_{XS}$ (*xor splits*), and $\mathcal{N}_{XJ}$ (*xor joins*). For $n \in \mathcal{N}$, $IN(n)$/$OUT(n)$ 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 $n \in \mathcal{N}_T$, $|IN(n)| = 1$ and $|OUT(n)| = 1$; for n_0, $|IN(n)| = 0$ and $|OUT(n)| = 1$ and vice versa for n_+; each node $n \in \mathcal{N}$ 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 the notation, the following definition closely follows [25].

Definition 2. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a process graph. A* state *t of $\mathcal{G}$ is a function $t : \mathcal{E} \mapsto \mathbf{N}$; 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. *$n \in \mathcal{N}_T \cup \mathcal{N}_{PS} \cup \mathcal{N}_{PJ}$ 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 $e' \in OUT(n)$ such that $t'(e) = t(e) - 1$ if $e = IN(n)$, $t'(e) = t(e) + 1$ if $e = e'$, $t'(e) = t(e)$ otherwise.*

3. $n \in \mathcal{N}_{XJ}$ *and there exists* $e' \in IN(n)$ *such that* $t'(e) = t(e) - 1$ *if* $e = e'$, $t'(e) = t(e) + 1$ *if* $e = OUT(n)$, $t'(e) = t(e)$ *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. [24,25].

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).[1] We denote predicates with G, H, I and constants with c, d, e. *Facts* are predicates grounded with constants, *Literals* are possibly negated facts. If l is a literal, then $\neg l$ denotes l's opposite ($\neg p$ if $l = p$ and p if $l = \neg p$); if L is a set of literals then $\neg L$ denotes $\{\neg l \mid l \in L\}$. We identify sets L of literals with their conjunction $\bigwedge_{l \in L} l$. Given a set $\mathcal{P}$ of predicates and a set C of constants, $\mathcal{P}[C]$ denotes the set of all literals based on $\mathcal{P}$ and C; if arbitrary constants are allowed, we write $\mathcal{P}[]$.

A *theory* $\mathcal{T}$ is a closed (no free variables) first-order formula. Given a set C of constants, $\mathcal{T}[C]$ denotes $\mathcal{T}$ with quantifiers interpreted over C. For the purpose of our formal semantics, $\mathcal{T}$ can be arbitrary. For computational purposes, we will consider the following standard restrictions. A *clause* is a universally quantified disjunction of atoms, e.g., $\forall x.\neg G(x) \vee \neg H(x)$. A clause is *Horn* if it contains at most one positive literal. A clause is *binary* if it contains at most two literals. A theory is Horn (binary) if it is a conjunction of Horn (binary) clauses. 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, $\forall x, y, z.G(x, y) \wedge G(y, z) \Rightarrow G(x, z)$.

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

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

[1] Hence our constants correspond to BPEL "data variables" [17]; note that the term "variables" in our context is reserved for variables as used in logics, quantifying over constants.

$\mathcal{T} \wedge con(e)$ is unsatisfiable; there do not exist n, e, and e' so that $e, e' \in OUT(n)$, $\mathcal{A}(e)$ and $\mathcal{A}(e')$ are defined, and pos(e) = pos(e').

We refer to cycles in $(\mathcal{N}, \mathcal{E})$ as *loops*; we refer to edges e for which $\mathcal{A}(e)$ 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. [1,8]) and WSMO (e.g. [9]). All the involved sets of literals (pre(n), eff(n), con(e)) are interpreted as conjunctions.[2]

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 [17], 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 $\mathcal{A}$ to be a partial function. That is, *an arbitrary subset of the process may be not annotated*. The rationale behind this is that validation will take place during modelling. In such a context, it is useful to allow validation of partially modelled – partially annotated – processes. The formal semantics is:

Definition 4. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph. Let C be the set of all constants appearing in any of the annotated pre(n), eff(n), 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 *$i : \mathcal{P}[C] \mapsto \{0, 1\}$. A* start state *$s_0$ is (t_0, i_0) where t_0 is as in Definition 2, and $i_0 \models \mathcal{T}[C]$, and $i_0 \models \mathcal{T}[C] \wedge$ eff(n_0) in case $\mathcal{A}(n_0)$ is defined. Let s and s' be states. We say that there is a* transition *from s to s' via n, written $s \rightarrow^n s'$, iff one of the following holds:*

1. *$n \in \mathcal{N}_{PS} \cup \mathcal{N}_{PJ} \cup \mathcal{N}_{XJ}$, $i_s = i_{s'}$, and $t_s \rightarrow^n t_{s'}$ according to Definition 2.*
2. *$n \in \mathcal{N}_{XS}$, $i_s = i_{s'}$, and $t'(e) = t(e) - 1$ if $e \in IN(n)$, $t'(e) = t(e) + 1$ if $e = e'$, $t'(e) = t(e)$ otherwise, where either $e' \in OUT(n)$ and $\mathcal{A}(e')$ is undefined, or $e' = argmin\{$pos$(e) \mid e \in OUT(n), \mathcal{A}(e)$ is defined,$i_s \models$ con$(e)\}$.*
3. *$n \in \mathcal{N}_T \cup \{n_+\}$, $t_s \rightarrow^n t_{s'}$ according to Definition 2, and either: $\mathcal{A}(n)$ is undefined and $i_s = i_{s'}$; or $i_s \models$ pre(n) and $i_{s'} \in min(i_s, \mathcal{T}[C] \wedge$ eff$(n))$ where $min(i_s, \mathcal{T}[C] \wedge$ eff$(n))$ is defined to be the set of all i that satisfy $\mathcal{T}[C] \wedge$ eff(n) and that are minimal with respect to the partial order defined by $i_1 \leq i_2$:iff $\{p \in \mathcal{P}[C] \mid i_1(p) = i_s(p)\} \supseteq \{p \in \mathcal{P}[C] \mid i_2(p) = i_s(p)\}$.*

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).

[2] It is easy to extend our formalism to allow arbitrary formulas for pre(n), eff(n), con(e); extending the actual validation leads to harder decision problems, and remains future work.

The part of Definition 4 dealing with $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{PJ} \cup \mathcal{N}_{XJ}$ parallels Definition 2, and should be easy to understand: the tokens pass as usual, and the interpretation remains unchanged. For $n \in \mathcal{N}_{XS}$, 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 [17]). If no edges are annotated, then the validation 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 $\mathcal{T}$ 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 $\mathcal{T}$, as well as 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 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 $\mathcal{T}$ and 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 eff(n), as implied by $\mathcal{T}$; this is reflected in the requirement that i' complies with both. The frame problem refers to the need to not change the previous state arbitrarily – e.g. if 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) [27]; while this approach is not uncontroversial, it underlies most of the recent work on formal semantics for execution of Semantic Web services (e.g. [14,5,11]). Alternative semantics from the AI literature (see [12] for an excellent overview) could be used in principle; this is a topic for future research.

2.2 Validation Tasks

We now formally define the validation properties that we consider in this paper. We start with a definition of *parallel* – un-ordered – task nodes; some of the validation tasks will be about potential conflicts between such nodes.

Definition 5. *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* parallel*, written $n_1 \parallel n_2$, if there exist execution paths $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ of $(\mathcal{N}, \mathcal{E})$ so that n_1 is executed before n_2 on $\boldsymbol{t}_1$, and n_2 is executed before n_1 on $\boldsymbol{t}_2$.*

We will show in Section 3 how parallel nodes can be detected. Note that $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ here are token executions, ignoring the semantic annotations; i.e., Definition 5 refers only to the workflow structure. We are interested in validating the following properties:

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

Let $n_1, n_2 \in \mathcal{N}_T$, $n_1 \parallel n_2$. Then, n_1 has a precondition conflict *with n_2 if $\mathcal{T} \wedge eff(n_1) \wedge pre(n_2)$ is not satisfiable; n_1 and n_2 have an* effect conflict *if $\mathcal{T} \wedge eff(n_1) \wedge eff(n_2)$ 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. [19,23]). 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).

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.[3] 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. Obviously, once parallelity is clarified, precondition and effect conflicts can be found by a simple loop over all pairs of parallel task nodes. In the remainder of the paper, we will assume that such a loop has completed successfully, i.e., without finding any conflicts. Note that reachability can be established as a side-effect of executability:

[3] For illustration it is useful to consider the special case where $\mathcal{T}$ 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.

Lemma 1. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph without case distinctions where $(\mathcal{N}, \mathcal{E})$ is sound and all $n \in \mathcal{N}_T$ are executable. Then all $n \in \mathcal{N}_T$ are reachable.*

We consider process graphs without case distinctions in our validation technique described in Section 3 below. We provide methods only for checking executability. With Lemma 1, once that is established, we know that reachability is also given. Note that Lemma 1 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.

2.3 An Example Process

We consider a sales order process that is inspired by the BPEL specification [3]. Figure 1 shows this process in BPMN, where as usual the symbol + represents parallel splits and joins. The receipt of a sales order triggers three concurrent activities: initiating the production scheduling, deciding on the shipper, and drafting a prize calculation. Once the shipper has been decided, the price calculation can be completed and the logistics can be arranged. After the latter, also the production scheduling can be completed. Finally, the invoice is processed. The process model is obviously *sound* in the workflow sense. However, let us take the perspective of a German machine producer (we call it GMP) that manufactures to order. This involves that production, delivery, and pricing are tailored to the product requirements of the customer.

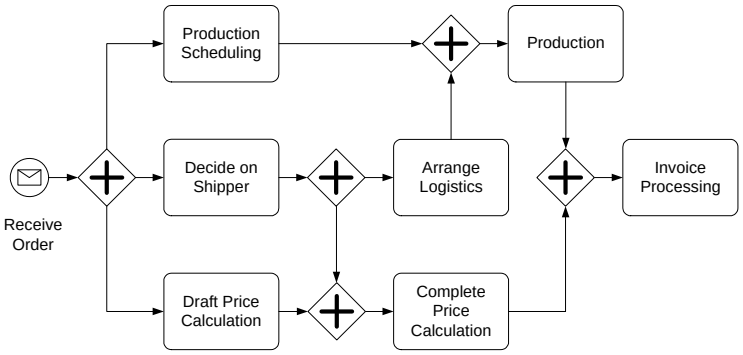

Fig. 1. Example of a sales order process modeled in BPMN.

GMP has used the depicted process successfully and without encountering any problems for years to produce and deliver to the national market. Now the company has made the decision to consider also international orders and delivery. 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 that controls 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, depicted in the bottom part of Table 1, has beed added, capturing some simple domain constraints that can be easily validated by stakeholders if a systems analyst translates them into natural language. Finally, Table 1 introduces thee process variables that capture the state of the involved business objects: order (*o*), production (*p*), calculation (*c*), shipper (*s*), and shipment (*sh*).

Activity	Precondition	Postcondition
Receive Order		orderReceived(o)
Production Scheduling	orderReceived(o) orderApproved(o)	productionScheduled(o,p)
Production	productionScheduled(o,p) calculationPrepared(o,c)	productionCompleted(o,p) calculationUpdated(o,c)
Decide on Shipper	orderReceived(o)	shipperDecided(o,s)
Arrange Logistics	shipperDecided(o,s) calculationPrepared(o,c)	calculationUpdated(o,c) shipmentApproved(o,sh)
Draft Price Calculation	orderReceived(o)	calculationDrafted(o,c)
Complete Price Calculation	calculationPrepared(o,c)	calculationFinalized(o,c)
Invoice Processing	productionCompleted(o,p) calculationFinalized(o,c)	orderCompleted(o)

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) $\forall x, y$: calculationUpdated(x,y) $\implies$ calculationPrepared(x,y) $\forall x, y$: ¬calculationDrafted(x,y) ∨¬calculationFinalized(x,y) $\forall x, y$: ¬calculationUpdated(x,y) ∨¬calculationFinalized(x,y) $\forall x, y$: ¬calculationPrepared(x,y) ∨¬calculationFinalized(x,y) $\forall x, y$: ¬calculationDrafted(x,y) ∨¬calculationUpdated(x,y)
Order approval	$\forall x, y$: shipmentApproved(x,y) $\implies$ orderApproved(x)

Table 1. Semantic annotations in the example. Top: preconditions and postconditions of the services. Bottom: axioms of the domain theory.

In a discussion with the production engineer the key account manager is embarrassed to learn that his set of services will not work for the production process of GMP, although the workflow of this process is sound. There are:

Non-executable tasks: 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.

Precondition conflicts: The new *Arrange Logistics* task require 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 *finalized*.

Effect conflicts: The new *Arrange Logistics* and *Production* services of GMP update the prize calculation, as indicated by their effect *calculationUpdated*. On the other hand, the parallel task *Complete Price Calculation* establishes the conflicting post-condition *calculationFinalized*. Hence, whether or not the calculation is finalized at the end of the process depends on which one of these nodes is executed last – a clearly undesirable behavior.

3 Polynomial-Time Validation Methods

We now specify an efficient validation algorithm for a particular class of processes, namely:

Definition 7. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$, $\mathcal{O} = (\mathcal{P}, \mathcal{T})$, be an annotated process graph. $\mathcal{G}$ is* basic *if it contains neither loops nor case distinctions, and $\mathcal{T}$ is binary.*

Note that our example from Section 2.3 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 $\mathcal{P}$. 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 $\overline{L} := \{l \in \mathcal{P}[C] \mid \mathcal{T} \wedge L \models l\}$, i.e., $\overline{L}$ is the closure of L under implications in the theory $\mathcal{T}$. Since $\mathcal{T}$ is binary, $\overline{L}$ can be computed in polynomial time given fixed arity [4]. Note that, with binary $\mathcal{T}$, an effect conflict can be easily detected as the (negative) overlap of the closure over the effect sets, i.e., $\mathcal{T} \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ is not satisfiable iff $\overline{\text{eff}(n_1)} \cap \neg\overline{\text{eff}(n_2)} \neq \emptyset$, and similarly for precondition conflicts.

Our validation algorithm performs three steps: (1) Determine a numbering $\#$ of the edges $\mathcal{E}$ 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 $\#$, execute an *M-propagation* algorithm that determines all pairs of parallel task nodes. (3) Determine, for each edge e, the set of literals that are always true when e is active. In what follows, we explain in detail only step (3). Steps (1) and (2) can be looked up in [26]. The outcome of step (2) is a matrix M^* whose rows and columns correspond to the set of edges $\mathcal{E}$. ${M^*}_i^j$ is a Boolean referring to the edges e, e' with $\#(e) = i, \#(e') = j$; this value determines parallelity in the following sense:

Lemma 2. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be an annotated process graph. There exists exactly one M-propagation result M^*, and for all $n_1, n_2 \in \mathcal{N}_T$ 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 $\mathcal{G}$.*

M^* is used to determine all parallel task nodes, and, with that, whether any precondition or effect conflicts arise. If so, those are indicated to the human modeller. Once the conflicts have been resolved, step (3) of our validation algorithm can be started. This step determines, for each edge e, the set of literals that is always true when e is active. The computation is based on propagation steps, updating sets of literals that are assigned to the edges. Upon completion, these literal sets are exactly the desired ones.

Definition 8. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be a basic annotated process graph without effect conflicts, and with constants C. We define the function $I_0 : \mathcal{E} \mapsto 2^{\mathcal{P}[C]} \cup \{\bot\}$ as $I_0(e) = \overline{\mathit{eff}(n_0)}$ if $e = OUT(n_0)$, $I_0(e) = \bot$ otherwise. Let $I, I' : \mathcal{E} \mapsto 2^{\mathcal{P}[C]} \cup \{\bot\}$, $n \in \mathcal{N}$. We say that I' is the* propagation of I at n *iff one of the following holds:*

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

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

If $\mathcal{A}(n)$ is not defined then $\mathit{eff}(n) := \emptyset$ 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 8 is a little hard to read but 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.[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 [27] definition of minimal change for solving the frame problem, c.f. Section 2.1. Our key observation is:

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

[4] 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 conflcits if desired.

Due to this observation, it suffices to subtract $\neg\overline{\text{eff}(n)}$ 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 $\mathcal{T}$; see [26] for an example where $\mathcal{T}$ is Horn.

Note that the I-propagation algorithm ignores preconditions. This may seem odd since preconditions play an essential part in executability. The trick is as follows. Let, for any edge e, $\bigcap e$ be the set of literals that will always be true when e is active. Then, *assuming that all task nodes are executable*, one can prove that $\bigcap e = I^*(e)$. Now, if all nodes are executable, then, since $\bigcap(e) = I^*(e)$, we have $\text{pre}(n) \subseteq I^*(IN(n))$ for all $n \in \mathcal{N}_T \cup \{n_+\}$. Conversely, if $n \in \mathcal{N}_T$ is the first node where $\text{pre}(n) \not\subseteq I^*(IN(n))$, then all of n's predecessors are executable and we know, with the same arguments as before, that $I^*(IN(n)) = \bigcap IN(n)$. Hence n is not executable. We get:

Theorem 1. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ be a basic annotated process graph without effect conflicts. There exists exactly one I-propagation result I^*. $\mathcal{G}$ is executable iff, for all $n \in \mathcal{N}_T \cup \{n_+\}$, pre$(n) \subseteq I^*(IN(n))$. With fixed arity, the time required to compute I^* is polynomial in the size of $\mathcal{G}$.*

4 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.

Lemma 3. *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 NP-hard. Deciding whether $\mathcal{G}$ is executable is coNP-hard.*

Lemma 4. *Assume an annotated process graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$ that is basic except that $\mathcal{T}$ is not binary. Deciding whether $\mathcal{G}$ is reachable is Σ_2^p-hard in general, and NP-hard if $\mathcal{T}$ is Horn. Deciding whether $\mathcal{G}$ is executable is Π_2^p-hard in general, and coNP-hard if $\mathcal{T}$ is Horn.*

Importantly, Lemma 4 holds even for propositional $\mathcal{T}$. Our main result regarding the computational borderline follows directly from Lemmas 3 and 4:

Theorem 2. *Assume an annotated process graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{O}, \mathcal{A})$. 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 $\mathcal{T}$ may involve Horn clauses.*

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 3 and 4 do not exploit this feature.

Corollary 1. *Theorem 2 holds even if predicate arity is fixed to 0, and eff(n_0) is complete, i.e., if for every $p \in \mathcal{P}[C]$: either $p \in$ eff(n_0) or $\neg p \in$ eff(n_0).*

5 Fixing Bugs

It would of course be desirable to be able to not only inform the human modeller that a process is flawed, but to also suggest actual fixes for the detected bugs. This is a highly difficult task – in general, it requires an understanding of the intentions behind the process; such an understanding may not be apparent from the semantic annotations. This notwithstanding, it is interesting to observe that some bug fixes are possible based on the information determined by our algorithms from Section 3.

Assume that task node n is not executable. Assume further that we know the set of literals $I(n)$ that are true in any state where n is activated – i.e., in case the process is basic, $I(n)$ is the set $I^*(IN(n))$ as per Definition 8. Then we can fix this bug by inserting a new task node n' before n, and setting pre$(n') := I(n)$ and eff$(n') :=$ pre(n). SWS discovery and composition can then be used to seek an implementation of n'. In other words, we can try to automatically fill the holes in the process implementation. Note that this is only possible if our validation method answers not only "yes/no"; as is the case for our method from Section 3, the method needs to know internally which literals will be true at which points. If the validation method uses a decision procedure, like e.g. a SAT solver, for a case where answering "yes/no" is hard, then one may have to make extra arrangements to also obtain the information $I(n)$.

If n_1 and n_2 have a precondition or effect conflict, then it may be possible to resolve that conflict via enforcing an ordering constraint between n_1 and n_2. In many cases, this can be done by inserting an additional parallel split and join between n_1 and n_2. Note that, for effect conflicts, there is a choice between the two possible orders. Note also that such a bug fix, and also the bug fix for executability outlined above, may introduce new bugs. Some of these issues can be overcome, e.g., by careful selection of SWS for the implementation, and/or by careful selection of ordering constraints.

6 Related Work

Verification of business process models has been a field of intense research since several years; see, e.g. [24] as an entry point into this literature. Indeed, as stated, for its workflow part our formalism adopts the Petri Net based approach from [24], in the notation used by [25].[5] What distinguishes our approach from all this work are our use of preconditions and effects, and of ontologies – of logical theories $\mathcal{T}$ – that constrain the behaviour of the underlying business domain. In particular, while propositional preconditions and effects can be modelled in terms of Petri Nets, this is not possible for our theories $\mathcal{T}$ and their impact on the semantics of executing tasks (as per Definition 4).

Some other works have appeared that extend Business Process validation beyond workflows; none of these works explores the same direction as ours, but it is worthwhile to point out the differences. The most closely related work is [13], which annotates task nodes with logical effects, and uses a propagation algorithm somewhat reminiscent of our I-propagation. There are, however, a number of important differences between the two approaches. [13] allow CNF effects, which are considerably more expressive

[5] We choose [25] since it works on the workflow model itself, rather than on a corresponding Petri Net, and hence simplifies our notations.

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), in stark difference to our polynomial time methods. Further, [13] consider neither preconditions nor logical theories that constrain the domain behavior. Finally, while we provide a formal execution semantics and prove our methods correct relative to that semantics, the work of [13] proceeds at a rather informal intuitive level.

Another related work is [15], which like our work introduces a notion of "semantic" correctness. Namely, [15] allow to annotate pairs of tasks as "exclusive" (respectively "dependent"), meaning they cannot (they must) co-occur in any process. This is a limited form of semantic annotation – not incorporating any logical formalism – which can be modelled in our approach using appropriate preconditions and effects, with empty theory $\mathcal{T}$. In that sense, [15] handles an interesting special case of our framework.

In terms of its terminology, [6] is strikingly similar to our approach, extending process models with predicates, constants, and variables. However, the meaning of these constructs is very different from ours, using them to specify constraints on role assignments, i.e., on the assignment of resources to tasks – while in our model, logics is used to detail the actual execution of the process within a given business domain.

[16] describe methods to check compliance of a process with a set of "rules". This is related to our approach in that a theory $\mathcal{T}$ could (to some extent) be defined to model such rules; but not vice versa since we build on general logics, while [16] covers some practical special cases. Perhaps more importantly, the focus of [16] is different from ours, concentrating on role assignments as well as "actions" to be triggered during process execution. [22] also deal with rules and business processes, but in a setting where the rules model (parts of) the workflow itself, rather than the surrounding ontology.

Another somewhat 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 [20,21]. Data flow analysis builds on compiler theory [2] (where data flows are examined for sequential programs mostly); it does neither consider ontologies (theories $\mathcal{T}$) nor logical conflicts, and hence explores a direction complementary to ours.

7 Conclusion

In Semantic Business Process Management, validation techniques are needed that consider the semantic annotations to check whether a process is consistent. Such validation techniques are semantic in that they need to work with ontologies. We have devised a first formalism for this kind of validation. We have identified efficient algorithms for a class of processes, and proved that this class is maximal with respect to the expressiveness of the annotation. Our next step will be to explore our notions from a more practical perspective. In particular, we believe certain classes of loops can be addressed with variants of our current algorithms.

References

1. A. Ankolekar et al. DAML-S: Web service description for the semantic web. In *ISWC*, 2002.

2. A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.
3. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, 2003.
4. B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
5. F. Baader, C. Lutz, M. Milicic, U. Sattler, and F. Wolter. Integrating description logics and action formalisms: First results. In *AAAI*, 2005.
6. E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information Systems Security*, 2(1):65–104, 1999.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
8. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, 2003.
9. D. Fensel et al. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, 2006.
10. T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, 57(2-3):227–270, 1992.
11. G. De Giacomo, M. Lenzerini, A. Poggi, and R. Rosati. On the update of description logic ontologies at the instance level. In *AAAI*, 2006.
12. A. Herzig and O. Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, 115(1):107–138, 1999.
13. George Koliadis and Aditya Ghose. Verifying semantic business process models in inter-operation. In *Intl. Conf. Services Computing (SCC 2007)*, 2007.
14. C. Lutz and U. Sattler. A proposal for describing services with DLs. In *DL*, 2002.
15. L. Thao Ly, S. Rinderle, and P. Dadam. Semantic correctness in adaptive process management systems. In *BPM*, 2006.
16. Kioumars Namiri and Nenad Stojanovic. A model-driven approach for internal controls compliance in business processes. In *SBPM*, 2007.
17. OASIS. *Web Services Business Process Execution Language Version 2.0*, April 2007.
18. OMG. Business Process Modeling Notation – BPMN 1.0. http://www.bpmn.org/, 2006.
19. A. Pnueli. The Temporal Logic of Programs. In *IEEE Annual Symposium on the Foundations of Computer Science*, 1977.
20. M. Reichert, S. Rinderle, and P. Dadam. ADEPT workflow management system: Flexible support for enterprise-wide business processes. In *BPM*, 2003.
21. M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive process management with ADEPT2. In *ICDE*, 2005.
22. S. Sadiq, M. Orlowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Journal of Information Systems*, 30(5):349–378, 2005.
23. W. van der Aalst, H. de Beer, and B. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *OTM Conferences*, 2005.
24. W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.
25. J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models though sese decomposition. In *ICSOC*, 2007.
26. I. Weber, J. Hoffmann, and J. Mendling. Semantic business process validation. Technical report, University of Innsbruck, 2008. Available at http://members.deri.at/~joergh/papers/tr-sbpm08.pdf.
27. M. Winslett. Reasoning about actions using a possible models approach. In *AAAI*, 1988.