Information Systems Frontiers

On Compliance Checking for Clausal Constraints in Annotated Process Models

Jörg Hoffmann · Ingo Weber · Guido Governatori

Received: February 23rd, 2009

Abstract Compliance management is important in several industry sectors where there is a high incidence of regulatory control. It must be ensured that business practices, as reflected in business processes, comply with the rules. Such compliance checks are challenging due to (1) the different life cycles of rules and processes, and (2) their disparate representations. (1) requires retrospective checking of process models. To address (2), we herein devise a framework where processes are annotated to capture the semantics of task execution, and compliance is checked against a set of constraints posing restrictions on the desirable process states. Each constraint is a clause, i.e., a disjunction of literals. If a process can reach a state that falsifies all literals of one of the constraints, then that constraint is violated in that state, and indicates non-compliance.

Naively, such compliance can be checked by enumerating all reachable states. Since long waiting times are undesirable, it is important to develop efficient (low-order polynomial time) algorithms that (a) perform exact compliance checking for restricted cases, or (b) perform approximate compliance checking for more general cases. Herein, we observe that methods of both kinds can be defined as a natural extension of our earlier work on semantic business process validation. We devise one method of type (a), and we devise two methods of type (b); both are based on similar restrictions to the processes, where the restrictions made by methods (b) are a subset of those made by method (a). The approximate methods each guarantee either of soundness (finding only non-compliances) or completeness (finding all non-compliances). We describe how one can trace the state evolution back to the process

J. Hoffmann
SAP Research,
Vincenz-Priessnitz-Str. 1
D-76131 Karlsruhe, Germany
Tel.: +49 6227 7-52540; Fax: +49 6227 78-51534
E-mail: joe.hoffmann@sap.com

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

G. Governatori
NICTA, Queensland Research Laboratory
Brisbane, AustraliaE-mail: guido.governatori@nicta.com.au

activities which caused the (potential) non-compliance, and hence provide the user with an error diagnosis.

Keywords Compliant process design · Compliance checking · Business process design · Formal process verification

1 Introduction

Compliance management is an area of increasing importance in several industry sectors where there is a high incidence of regulatory control e.g., financial services, gaming, and health care. Ensuring that business practices reflected in business process models are compliant to required regulations (existing and new) is a highly challenging task due to the following reasons. First, the life cycles of the two (regulatory obligations vs. business strategy) are not aligned in terms of time, governance, or stakeholders [30] and hence compliance requirements cannot simply be incorporated into the initial design of process models. Second, conceptually faithful specifications for compliance rules and process models respectively are fundamentally different from a representational point of view [35], thus making it difficult to provide comparison methods. Herein, we propose to provide retrospective checking of process models in acknowledgment of the disparate life cycles as mentioned above. That is (i) to check the compliance of a new or altered process against the compliance rules, and (ii) check the whole process repository against changed compliance rules, e.g., when new regulations come into being.

Compliance rules in our approach are represented as a *constraints base*. That constraints base is in conjunctive normal form: it is a conjunction (logical "and") of clauses, where each clause is a disjunction (logical "or") of literals. Literals are atomic logical statements, i.e., predicate symbols that may be positive or negated. The literals may contain variables. These are quantified universally, and range over the entities of interest at process execution time (e.g., in a process dealing with cheques, the constraints will be stated to hold for all cheques). Each clause is a constraint on the states that are desirable as per the compliance rules: if a state does not satisfy the clause, then that state is non-compliant. Due to the outer conjunction, all clauses must be satisfied. For example, say a cheque must be signed by any two of the people authorized to sign it. Say three people are authorized to sign cheques, $Henning$, Leo, and $Dietmar$. This corresponds to the rule $\forall x : cheque(x) \rightarrow sign(x, Henning, Leo) \vee sign(x, Henning, Dietmar) \vee sign(x, Leo, Dietmar)$, which is the same as $\forall x : \neg cheque(x) \vee sign(x, Henning, Leo) \vee sign(x, Henning, Dietmar) \vee sign(x, Leo, Dietmar)$.

Clearly, the complexity of compliance rules in general necessitates a more expressive language (see e.g., [19]) than this form of constraints bases. Our aim in this paper is not to provide a fully-fledged framework for compliance, but rather to develop computationally efficient compliance checking methods for this particular restricted form of compliance.

Fig. **??** gives an overview of our framework. Processes are modeled in terms of a typical workflow language, featuring task nodes (the activities carried out inside the process) as well as parallel splits/joins and xor splits/joins to model the control flow. Such a model specifies only which sequences of activities – which execution paths – may occur; it cannot model more subtle or indirect dependencies between the activities. To cater for the latter, we allow semantic annotations. Tasks are annotated with preconditions and effects, which are conjunctions of logical literals, formulated in the terms of an ontology that axiomatizes the underlying business domain.

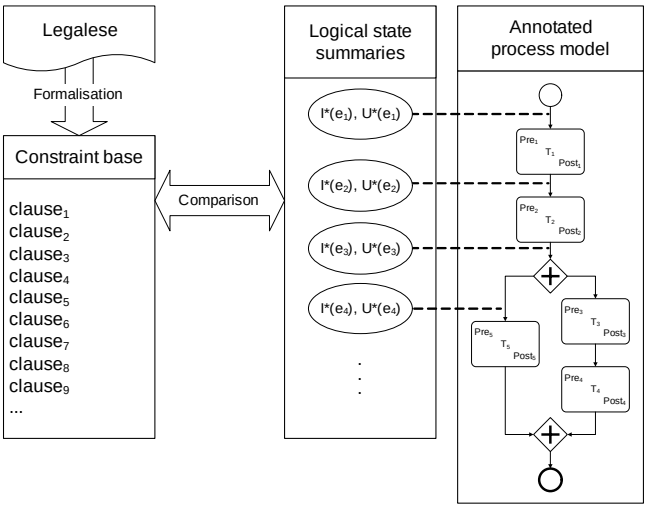

Fig. 1 An overview of our framework.

Given the semantic annotations, execution paths of the process traverse *states* that do not only define which edges of the process are active (carry a token), but also define a "logical state", i.e., how the logical propositions are interpreted. In the "Logical state summaries" part of Fig. **??**, $I^*(e_i)$ and $U^*(e_i)$ denote sets of literals which characterize particular properties of execution paths, relative to edges e_i. Namely, the literals in $I^*(e_i)$ are *guaranteed to be true* whenever e_i is active (so those sets correspond to the *intersection* of the logical states at e_i), and the literals in $U^*(e_i)$ *might be true* when e_i is active (so those sets correspond to the *union* of the logical states at e_i). I^* and U^* are computed as part of our compliance checking algorithms (more details below).

Note that the possibility to semantically annotate the process already provides opportunities for certain forms of compliance checking, even without introducing a constraints base: e.g., if, by a compliance rule expressing an obligation, activity A must always be performed prior to an activity B, then we can give B a (new) precondition p and include p into A's effect. The process is then compliant iff B's precondition is always guaranteed to be true.

We leave the detailed exploration of encoding methods as above for future work. Herein, we focus on clausal constraints – disjunctive compliance rules – which are more powerful. They enable the modeler to specify that one out of a number of conditions must always be satisfied – by contrast, preconditions formulate only conjunctive rules, specifying that *all* of a number of conditions must always be satisfied. An example of a disjunctive compliance rule has been given above already, where we have three people authorized to sign cheques, and any cheque must be signed by two of them, yielding the clause $\forall x : \neg cheque(x) \vee sign(x, Henning, Leo) \vee sign(x, Henning, Dietmar) \vee sign(x, Leo, Dietmar)$.

The compliance rules are checked against the logical states that can be traversed by the process. A naive way of checking compliance is hence to enumerate all those states. Clearly, given that the number of states is (in general) exponential in the size of the process, such an approach is not desirable. A human modeler will not tolerate long waiting times during process modeling, and checking the compliance of a whole process repository against an altered constraints base may become completely infeasible if every single process involves a

state enumeration. The question hence is: do restricted cases exist where we can check compliance efficiently? And can we devise approximation techniques for more general cases?

We herein give positive answers to both questions. We leverage on an algorithm, *I-propagation*, that we developed in previous work [33]. I-propagation was originally intended for validating a certain property of semantically annotated processes, namely whether or not all task nodes are "executable". A task node is executable if, whenever the task is activated by the control-flow, its precondition literals are guaranteed to be satisfied. Checking executability is essentially like checking compliance with trivial clauses of length 1 (unit clauses). Herein, we provide ways of extending the algorithm to deal with longer, non-trivial, clauses.

I-propagation runs in polynomial time and, for a particular restricted class of processes which we call *basic processes*, computes exactly the sets of literals that are necessarily true at particular points during process execution: namely, the I^* sets in Fig. **??** (the U^* sets can be derived easily from I^*). Basic processes have no loops, no effect conflicts (no parallel task nodes with contradicting effect annotations), the ontology axioms are all binary clauses (disjunctions of at most 2 literals), and all task nodes are executable.

Regarding binary axioms and executability, we proved that those restrictions are necessary for computational efficiency: determining the necessarily true literals is **NP**-hard or worse if we relax these restrictions [33].[1] For effect conflicts, it is an open question whether or not they could be handled efficiently. Note that parallel task nodes with conflicting effects (such as write operations onto the same database) are often not sensible, namely in applications where parallel nodes might execute at the same time point, or where the outcome of the process should not depend on the order of execution of parallel tasks. Nevertheless, there may be applications in which effect conflicts are intended, and compliance needs be checked in their presence. Figuring out how to do so is a topic for future work.

Regarding loops, as stated, the original definition of basic processes [33] disallows them. In the meantime, however, in other work we have managed to overcome this restriction, devising an extension of I-propagation that correctly handles basic processes *with* loops, in polynomial time. This extension is entirely orthogonal to the techniques we introduce herein for handling non-trivial clauses. In effect, our results generalize effortlessly. We do not include a formal treatment of loops, since that would amount to nothing but notational clutter. The extended I-propagation algorithm handles "structured loops" only, formalized in terms of (repeatable) sub-processes, so that the overall process takes the form of a tree of sub-processes. Since the issues of loops and non-trivial clauses are orthogonal, none of this additional formalism is of any relevance to the results contained herein. *Our results hold as stated also for processes with structured loops.* We will outline why this is so.

Our compliance checking methods are based on two observations:

- If a clause C is *non-contradicted* – there exists no task node effect invalidating any of C's literals – then we can compile compliance with C into compliance with a unit clause C', and hence re-use I-propagation for exact compliance checks. It is important to note here that such a situation is not uncommon; in the $cheque$ example above, e.g., one would not expect to have a task node with effect $\neg cheque(x)$ (saying that x is no longer a cheque), and neither would one expect to have tasks that "un-sign" a cheque.
- For the more general case of contradicted clauses, we can still exploit the information provided by I-propagation, namely in terms of two approximative tests. The first of those

[1] It may seem odd that executability is a prerequisite, since I-propagation was designed to check this same property. The latter can, in fact, be done, by a certain contra-position argument (outlined in Section 3 where we explain I-propagation).

essentially checks whether all literals of a clause are necessarily false. This method is sound but not complete (it guarantees to find only non-compliances, but not to find all non-compliances). The other method checks whether none of the literals of a clause is necessarily true. This method is complete but not sound.

All the methods inherit the restrictions of I-propagation, i.e., they handle basic processes. However, the approximate methods do *not* require executability – as we show herein, I-propagation still gives a certain guarantee of conservativeness without this prerequisite; that guarantee suffices to obtain soundness respectively completeness as desired.

In this paper we define the compliance checking methods and prove their relevant theoretical properties. We further describe how one can trace the state evolution back to the process activities which caused the non-compliance, and hence provide the user with a diagnosis facility. Detailed empirical evaluation of the proposed methods is beyond the scope of this paper. We remark that we have a prototypical implementation of I-propagation, which as expected exhibits fine runtime behavior. For example, the prototype handles a non-trivial process with 40 nodes and 46 edges within fractions of a second.

Section 2 introduces our formalism for semantically annotated processes, as well as our formalization of constraints bases. Section 3 explains the I-propagation algorithm we build on. Section 4 presents our methods for compliance checking, and Section 5 contains our diagnosis methods. Section 6 discusses related work, Section 7 concludes.

2 Annotated Business Processes and Constraint Bases

In this section we give our definitions regarding annotated process graphs and the constraints on their behavior, starting with the former.

2.1 Annotated Business Processes

We introduce a formalism for business processes whose tasks are annotated with logical preconditions and effects. This formalism is the basis of our work, since it allows us to model the behavior of process activities, and hence of the overall process, at a level that is fine-grained enough to sensibly check for the kind of compliance we target in this work. We first introduce our notions regarding control-flow, then we discuss the semantic annotations.

2.1.1 Control-Flow

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 the definition is, without any modification, adopted from the workflow literature, following closely the terminology and notation used in [32].

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*, *end 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 $IN(n_+) = e_+$.

Example 1. *Consider Fig. 2. The upper half of the figure depicts an example process graph in standard BPMN notation. In fact, this example is based on a BPMN diagram example from the BPMN 1.1 specification [28]. We will use this process graph as a running example throughout the paper.*

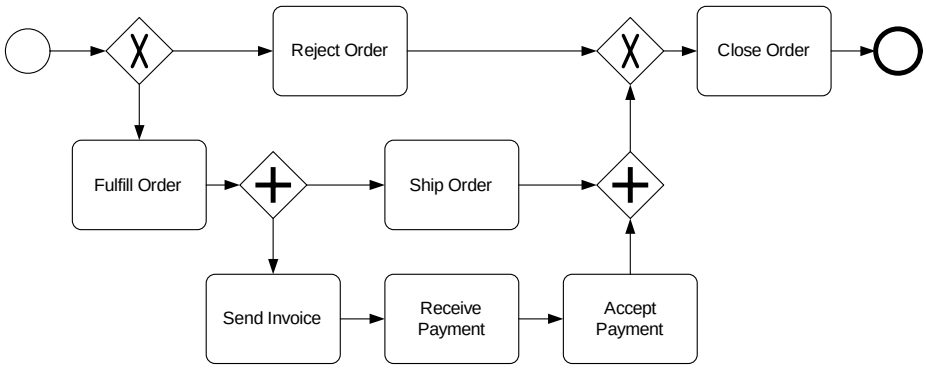

Fig. 2 Our illustrative running example, in BPMN notation.

The process contains edges, a start node (thin circle), an end node (thick circle), various tasks (e.g., "Receive Order", "Ship Order", etc.), and a number of routing nodes such as the xor split after "Receive Order". Only one of the branches after this xor split will be executed: either the one on which the order is rejected, or the other one which features several more task nodes. Note that "Ship Order" can be executed in parallel to the other task nodes, due to the parallel split and join nodes.

The intuitive meaning of the structures introduced by Definition 1 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 for processes in our formalism. Note that the formalism describes a common subset of process modeling notations like BPMN [28] and process execution languages like WSBPEL [27]. For example, the xor split in our formalism can be used to represent both the data-driven decision gateway and the event-driven decision gateway (also called deferred choice). Similarly, our distinction between a split and a join gateway is not a restriction, since a combined join-split gateway can be translated into two separate gateways. It is not the intention of our formalism to replace commonly used languages. Rather, the formalism only serves as an abstract notation to present our results.

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 for Definition 1, we closely follow [32].

Definition 2 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a process graph. A *state* t of $\mathcal{G}$ is a function $t : \mathcal{E} \mapsto \mathbf{N_0}$ from the set of edges into the natural numbers including 0; 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:
Tasks, parallel splits and joins (tokens from INs to OUTs). $n \in \mathcal{N}_T \cup \mathcal{N}_{PS} \cup \mathcal{N}_{PJ}$ and

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

Xor splits (token from IN to one OUT). $n \in \mathcal{N}_{XS}$ and there exists $e' \in OUT(n)$ such that

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

Xor joins (token from one IN to OUT). $n \in \mathcal{N}_{XJ}$ and there exists $e_0 \in IN(n)$ such that

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

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.

Note in Definition 2 that, in all transitions, $t(e)$ is implicitly constrained to be greater than 0 for the IN edges e from which tokens are taken: otherwise, $t'(e) = t(e)-1$ would have to be less than 0, which is not allowed because t' is a function into the natural numbers. In all other aspects, the definition 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 $t'(e_+) > 0$; there is no reachable state which has a token both on e_+ and on some other edge; and there are no dead transitions, i.e., for every transition there is an execution path that can fire it. These properties can be ensured using standard workflow validation techniques, e.g., [5,32].

Note that Definitions 1 and 2 do allow cycles in the graph, i.e., they do cater for loops. As stated, for the purpose of applying I-propagation we will later restrict our focus to basic processes, which disallow cycles. As also stated, we will outline extensions that cater for *structured* loops. That formalism does not allow arbitrary cycles in the process graph. Instead, the graph as such is acyclic, but it may contain loops in the form of repeatable subgraphs (of the same kind). The overall structure then is a tree of acyclic sub-graphs, where all but the root of the tree are repeatable. Definition 2 for such structured loops is straightforward, simply allowing control to pass into and out of sub-processes, and to pass from the end of a repeatable sub-process to its start.

For the notions considered in the rest of this section – semantic annotations and constraints bases – structured loops make no difference at all, i.e., these notions carry over exactly as stated.

2.1.2 Semantic Annotations

For the annotations, we use standard notions from logics, involving logical *predicates* and *constants* (the latter correspond to the entities of interest at process execution time).[2] We denote predicates with upper-case letters, usually G, H, I, and constants with lower-case letters, usually 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 *clause* is a universally quantified disjunction of logical atoms, i.e., of non-grounded literals. For example, $\forall x : \neg G(x) \vee \neg H(x)$ is a clause. The axiomatization that comes with an ontology is a *theory* θ: a conjunction of clauses.[3] Our polynomial-time algorithms will be designed for *binary* theories: a clause is binary if it contains at most two literals; a theory is binary if it is a conjunction of binary clauses. Note that binary clauses can be used to specify many common ontology properties such as subsumption $\forall x : G(x) \Rightarrow H(x)$ (where as usual $\phi \Rightarrow \psi$ 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 *ontology* Ω is a pair $(\mathcal{P}, \theta)$ where $\mathcal{P}$ is a set of predicates (Ω's formal terminology) and θ is a theory over $\mathcal{P}$ (constraining the behavior of the application domain encoded by Ω). For complexity considerations, throughout the paper we will assume *fixed arity*, i.e., a fixed upper bound on the arity of 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). Annotated process graphs are defined as follows.

Definition 3 An *annotated process graph* is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$. $(\mathcal{N}, \mathcal{E})$ is a process graph, $\Omega = (\mathcal{P}, \theta)$ is an ontology, and α, the *annotation*, is a function mapping $n \in \mathcal{N}_T \cup \{n_0, n_+\}$ to $(\text{pre}(n), \text{eff}(n))$ where $\text{pre}(n), \text{eff}(n) \subseteq \mathcal{P}[]$. We require that there does not exist an n so that $\theta \wedge \text{eff}(n)$ is unsatisfiable, or $\theta \wedge \text{pre}(n)$ is unsatisfiable.

We refer to $\text{pre}(n)$ as n's *precondition*, and to $\text{eff}(n)$ as n's *effect* (sometimes called *postcondition* in the literature). The annotation of tasks – atomic actions that on the IT level can e.g., 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, 11]) and WSMO (e.g., [12]). All the involved sets of literals ($\text{pre}(n)$, $\text{eff}(n)$) are interpreted as conjunctions. Similarly to Definition 1, the requirements stated in Definition 3 are just basic sanity checks.

Example 2. *Consider again our running example from Fig. 2. The semantic annotations are given in Table 1. For simplicity, the theory θ is empty, i.e., no axioms are given; we will discuss a modified example with non-empty θ below. Likewise, no preconditions are specified, and an accordingly modified example will be given further below. Note the negative effect of Accept Payment.*

The formal execution semantics is defined as follows.

Definition 4 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be an annotated process graph. Let C be the set of all constants appearing in any $\text{pre}(n), \text{eff}(n)$. A *state* s of $\mathcal{G}$ is a pair (t_s, i_s) where t is a token

[2] Hence our constants correspond to BPEL "data variables" [27]; note that the term "variables" in our context is reserved for variables as used in logics, quantifying over constants.

[3] As indicated, our compliance rules are also clauses; however, their formal interpretation is different. This will be explained in Section 2.2, when we formally introduce constraints bases.

Task	Effects
Start Node	order(o), received(o)
Reject Order	rejected(o)
Fulfill Order	fulfilled(o)
Ship Order	shipped(o)
Send Invoice	invoiceSent(o, i), paymentExpected(o)
Receive Payment	paymentReceived(i)
Accept Payment	paymentAccepted(i), *not* paymentExpected(o), paid(o)
Close Order	closed(o)

Table 1 Semantic annotations for the process in Fig. 2

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 $i_0 \models \theta[C] \wedge \text{eff}(n_0)$. 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. $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{PJ} \cup \mathcal{N}_{XS} \cup \mathcal{N}_{XJ}$, $i_s = i_{s'}$, and $t_s \rightarrow^n t_{s'}$ according to Definition 2.
2. $n \in \mathcal{N}_T$, $t_s \rightarrow^n t_{s'}$ according to Definition 2, $i_s \models \text{pre}(n)$ and $i_{s'}$ is a member of $min(i_s, \theta[C] \wedge \text{eff}(n))$. The latter is the set of all i that satisfy $\theta[C] \wedge \text{eff}(n)$ and that are minimal with respect to the partial order defined by

$$i_1 \leq i_2 \text{ :iff } \{p \in \mathcal{P}[C] \mid i_1(p) \neq i_s(p)\} \subseteq \{p \in \mathcal{P}[C] \mid i_2(p) \neq i_s(p)\}.$$

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 $(\mathcal{N}, \mathcal{E}, \Omega, \alpha)$, we will use the term *execution path of* $(\mathcal{N}, \mathcal{E})$ to refer to an execution over tokens that acts as if no annotations were present.

The part of Definition 4 dealing with $n \in \mathcal{N}_{PS} \cup \mathcal{N}_{PJ} \cup \mathcal{N}_{XS} \cup \mathcal{N}_{XJ}$ parallels Definition 2: the tokens pass as usual, and the interpretation remains unchanged.

Consider now the start states, of which there may be many, namely all those that comply with θ, as well as $\text{eff}(n_0)$ (if annotated). This models the fact that, at design time, we do not know the precise situation in which the process will be executed. All we know is that, certainly, this situation will comply with the domain behavior given in the ontology and with the properties guaranteed as per the annotation of the start node.

The semantics of task node executions is the most intricate bit. First, for the obvious reasons, $\text{pre}(n)$ is required to hold. The tricky bit lies in the definition of the possible outcome states i'. The semantics defines this to be *the set of all i' that comply with θ and eff(n), and that differ minimally from i.* This draws on the AI literature for a solution to the *frame* and *ramification* problems. The latter problem refers to the need to make additional inferences from $\text{eff}(n)$, as implied by θ. This is reflected in the requirement that i' complies with both $\text{eff}(n)$ and θ. The frame problem refers to the need to not change the previous state arbitrarily – e.g., if an activity changes an account A, then any account B different from A should not be affected. This is reflected in the requirement that i' differs minimally from i. More precisely, i' is allowed to change i only where necessary, such that there is no i'' that makes do with fewer changes. This semantics follows the *possible models approach* (PMA) [34]; while this approach is not universally accepted, it is widely used and in particular underlies most recent work on formal semantics for execution of Semantic Web services (e.g., [24, 8, 17]).

As stated, our compliance checking methods will be defined for binary theories only. Binary clauses specify certain consequences that *must* be implied by particular effects. In that way, binary clauses are a convenient modeling construct, and their semantics is "uncritical" in that there is no ambiguity about their implications; this is not so for clauses with more literals. The following example illustrates this.

Example 3. *Consider a variant of the process in Fig. 2 with a task node n that cancels the order o. Suppose that cancellation is annotated by eff$(n) = \{$orderCancelled(i)$\}$. As in Example 2, the ontology contains the predicate paymentExpected. Further, say θ contains a clause specifying that payment cannot be expected for any order that is already canceled: $\forall x : \neg$orderCancelled$(x) \vee \neg$paymentExpected(x). Say we execute n in a state s where we have paymentExpected(o). Which are the possible resulting states s', with $s \rightarrow^n s'$? By the definition of $min(i_s, \theta[C] \wedge$ eff$(n))$ in Definition 4, any such state must satisfy $(\neg$orderCancelled$(o) \vee \neg$paymentExpected$(o)) \wedge$ orderCancelled(o) which means of course that s' must satisfy $\neg$paymentExpected(o). So the value of paymentExpected(o) is changed as a side-effect of applying n.*

Now, among others, the ontology also contains the predicates shipped(.) and invoiceSent(.,.). Suppose that θ specifies that, for any order which has both shipped and for which an invoice has been sent, we expect the payment: $\forall x, y : \neg$shipped$(x) \vee \neg$invoiceSent$(x, y) \vee$ paymentExpected(x). Say that the state s from above has shipped(o) and invoiceSent(o, i). Now, upon executing n, as pointed out above we no longer expect payment for o and so the clause is no longer satisfied and we must "repair" it. Since the clause is not binary, this spawns a non-trivial behavior of the minimal change semantics. There are three options: falsify shipped(o), falsify invoiceSent(o, i), or falsify both. The first two options each yield a resulting state s'. The latter option, in contrast, does not *yield a resulting state s' because it is not a* minimal *change. One needs not assume that o is neither shipped nor invoiced. It suffices to assume one of those. The intuitive meaning of this semantics is that, since o was canceled (by n), something bad must have happened, i.e., the shipment failed, or there was a problem with the invoicing. While, of course, both may be the case, this seems an unlikely assumption and is hence not considered.*

2.2 Constraints Bases

It remains to define what constraints and non-compliances are:

Definition 5 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be an annotated process graph with constants C, where $\Omega = (\mathcal{P}, \theta)$. A *constraints base* β is a set of clauses over the predicates $\mathcal{P}$. Let $\phi = \forall X.\psi(X)$ be a clause in β. Then any grounding $\psi(C_0)$ of ψ with a tuple C_0 of constants from C is a *grounded constraint*. A reachable state s is a *non-compliance*, or *non-compliant state*, iff there exists a grounded constraint $\psi(C_0)$ such that $s \not\models \psi(C_0)$.

This definition is straightforward and should be self-explanatory. We will identify $\psi(C_0)$ with the set of literals it contains. Note that α and β share the vocabulary $\mathcal{P}$, and hence the semantic annotations may make statements of interest to compliance checking (this would not be the case for disjoint vocabularies). Of course, doing the annotation in this way – so that the annotations are adequate for compliance checking – may induce additional modeling effort in practice.

A subtle point is the distinction between β and θ. Both are formalized similarly. The difference lies in how they are interpreted. θ models the conditions that any state must satisfy,

due to the "physical" behavior of the underlying business domain – such as, "any purchase order of a particular product is, in particular, a purchase order". In contrast, β models the conditions that any state *should* satisfy, in order to comply with the rules of the business – such as, for example, that the auditor for any activity is different from the actor who performed or authorized the activity (separation of duties); there is no physical law enforcing these rules.[4] At the formal level, this difference is accounted for by using θ as part of the definition how states evolve, while using β to check whether the states are desirable or not.

Example 4. *Reconsider our running example from Fig. 2 and Table 1. Say our constraints base is* $\beta = \{\forall x : order(x) \wedge received(x) \implies rejected(x) \vee paid(x)\}$. *In words, we impose that any order which has been received must be either rejected, or paid.*

Consider the grounding of x *with* o*, i.e., the concrete order dealt with by the process. The antecedent of the implication,* $order(o) \wedge received(o)$*, is always true as soon as Receive Order has been executed. At that time, i.e., directly after executing Receive Order, the order will neither be rejected nor paid, so the constraint is violated at that point in the process. The constraint becomes satisfied after Reject Order, and it becomes satisfied after Accept Payment; it remains true after the xor join because, no matter which side of the join has been executed beforehand, one of the two options will be fulfilled.*

3 I-Propagation

We now describe the I-propagation algorithm, which we developed in previous work [33]. As stated in the introduction, I-propagation forms the starting point of our work herein.

The original purpose of I-propagation was to determine whether or not a process is *executable*. An individual task node n is executable if, whenever the task is activated, its preconditions are true: for all reachable states s with $t_s(IN(n)) > 0$, $s \models \text{pre}(n)$. The overall process is executable if every one of its task nodes is. I-propagation determines whether or not that is the case, by ways of computing, for each edge e, the set of literals that is always true when e carries a token.

I-propagation runs in low-order polynomial time, and works correctly for a restricted class of processes. To state this formally, we first need a little terminology. We refer to cycles in $(\mathcal{N}, \mathcal{E})$ as *loops*. Two edges e_1 and e_2 are *parallel* if there exists a reachable state s where $t_s(e_1) > 0$ and $t_s(e_2) > 0$; two task nodes are parallel if their incoming edges are. If n_1 and n_2 are parallel task nodes and $\theta \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ is unsatisfiable, then we say that n_1 and n_2 have an *effect conflict*. I-propagation handles what we call "basic" processes:

Definition 6 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$, $\Omega = (\mathcal{P}, \theta)$, be an annotated process graph. $\mathcal{G}$ is *basic* if it contains neither loops nor effect conflicts, and θ is binary.

Our compliance checking algorithms inherit these restrictions from I-propagation, i.e., non-basic processes are outside the more tractable cases that we identify. The various restrictions were discussed in the introduction already. For non-binary theories, we have proved in our previous work that this restriction cannot be relaxed without losing computational efficiency [33]. Whether or not this is the case for effect conflicts is an open question. Regarding loops, we have in the meantime devised an extension of I-propagation to structured loops. We stick to the original – much more concise – formalization because the extension

[4] A striking if imprecise illustration is that of gravity vs. traffic rules: any car must drive on the ground, by physical law; whether they use the left or right hand side of the road is a matter of rules.

to loops is orthogonal to the compliance issues considered herein. We will outline how the extended I-propagation works, and how our results on compliance checking carry over.

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 $\overline{L} := \{l \in \mathcal{P}[C] \mid \theta \wedge L \models l\}$, i.e., $\overline{L}$ is the closure of L under implications in the theory θ. Since θ is binary, $\overline{L}$ can be computed in polynomial time given fixed arity [6]. Note that, with binary θ, an effect conflict can be easily detected as the (negative) overlap of the closure over the effect sets, i.e., $\theta \wedge \text{eff}(n_1) \wedge \text{eff}(n_2)$ is not satisfiable iff $\overline{\text{eff}(n_1)} \cap \neg\overline{\text{eff}(n_2)} \neq \emptyset$.

I-propagation consists of two steps: (1) determine all pairs of parallel edges; (2) using that information, determine for each edge e the set of literals that is always true when e is active. In what follows, we explain only step (2), which is more directly connected to the results presented herein. Also, we focus on the details which are directly relevant to the remainder of this paper. The interested reader may look up all technical details in [33].

As the name suggests, I-propagation is based on propagation steps. The propagation starts at the outgoing edge of the start node, and proceeds by iteratively firing subsequent nodes in the graph. The propagation steps update sets of literals; one such set is assigned to each edge in the graph. When the propagation ends, these literal sets are exactly the desired ones, i.e., the literals that are always true whenever the respective edge is activated. One tricky bit is that we need to capture the "side effects" that any task node may have, on edges other than its own OUT edge. For this, we introduce the following notation: parallel-eff(e) is the collection of all parallel effect literals of an edge $e \in \mathcal{E}$. Precisely:

$$\text{parallel-eff}(e) := \bigcup_{e' \in \mathcal{E} \text{ parallel to } e, e' = OUT(n') \text{ for } n' \in \mathcal{N}_T} \text{eff}(n')$$

The formal definition of I-propagation follows. The definition is hard to read at first, but relies on straightforward key ideas; the reader may choose to skip directly to the intuitive explanation of the algorithm below.

Definition 7 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, with constants C. We define the function $I_0 : \mathcal{E} \mapsto 2^{\mathcal{P}[C]} \cup \{\bot\}$ as $I_0(e) = \overline{\text{eff}(n_0)}$ 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 $I(e) \neq \bot$ for all $e \in IN(n)$, and $I(e) = \bot$ for all $e \in OUT(n)$, and one of the following holds:

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

$$I'(e) = \begin{cases} I(IN(n)) \setminus \neg\overline{\text{parallel-eff}(e)} & e \in OUT(n) \\ I(e) & \text{otherwise} \end{cases}$$

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

$$I'(e) = \begin{cases} (\bigcup_{e' \in IN(n)} I(e')) \setminus \neg\overline{\text{parallel-eff}(e)} & e = OUT(n) \\ I(e) & \text{otherwise} \end{cases}$$

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

$$I'(e) = \begin{cases} (\bigcap_{e' \in IN(n)} I(e')) \setminus \neg\overline{\text{parallel-eff}(e)} & e = OUT(n) \\ I(e) & \text{otherwise} \end{cases}$$

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

$$I'(e) = \begin{cases} \overline{\text{eff}(n)} \cup (I(IN(n)) \setminus \neg\overline{\text{eff}(n)}) & e = OUT(n) \\ I(e) & \text{otherwise} \end{cases}$$

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

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

The definition of I_0 should be obvious: it just collects the literals that are guaranteed to hold at the start edge. The propagation algorithm, although formulated as a fixpoint operation, then performs a single pass over the process – due to the requirement, on every propagation step, that the IN edges are not set to $\perp$ and the OUT edges are set to $\perp$. For split nodes, the OUT edges simply copy their sets from the IN edge. We have to subtract the negated side effects of any parallel task nodes since those literals may be invalidated while the OUT edges are still activated. For parallel joins, the OUT edge assumes the union of $I(e)$ for all IN edges e; this is justified because all those literals must be true when n is executed. Again, we need to care about side effects. Dually, for xor joins we need to take the intersection instead since any one of the incoming edges may be active before n is executed. Finally, if n is a task node, then we need to take account of n's own effects. This is done in the obvious manner, removing the literals contradicted by $\text{eff}(n)$, and adding the literals contained in $\text{eff}(n)$. Note that side effects need not be taken into account here since effect conflicts are excluded by prerequisite.

Example 5. *We illustrate I-Propagation using our running example from Fig. 2 (workflow) and Table 1 (annotations). The outcome of I-Propagation is depicted in Fig. 3.*

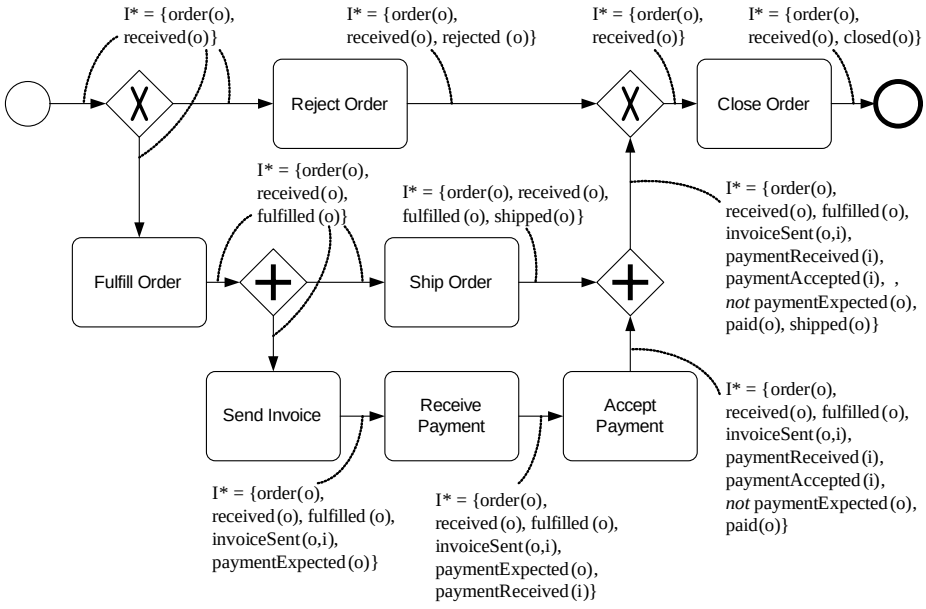

Fig. 3 Example process model from Fig. 2 also showing the I-Propagation results, I^*.

Observe how I-Propagation applies the effects of a task node by adding them to the task's outgoing edge. The simplest occurrence of this is the first task node, "Receive Order", where the ingoing edge has an empty I^*. A more intricate propagation is the one over "Accept Payment", since the task's negative effect not *paymentExpected*(o) *falsifies the previously true paymentExpected*(o)*. It is also evident how the I^* from the incoming edge of a split node is copied to its outgoing edges. In contrast, the parallel join takes the union of the I^* of its incoming edges (consider e.g. the literal shipped*(o)*). The xor join in turn takes the intersection of its incoming edges' I^*s – which is then the same as before the xor split.*

It should be noted that the details are actually not as straightforward as the above may suggest. The correctness proof takes 6 pages, determining for example particular properties of sets of parallel edges, and of binary theories. Let us briefly consider the latter. In the handling of task nodes, Definition 7 uses the notation $\overline{\text{eff}(n)}$. As stated above, this denotes the set of all literals which follow from θ and eff(n). Why is it correct to simply subtract $\neg\overline{\text{eff}(n)}$ and add $\overline{\text{eff}(n)}$? Recall that the semantics of task node executions is quite complex, c.f. Section 2.

The observation underlying the simple handling in Definition 7 is: *(*) With binary θ, if executing n makes literal l false in one possible transition, then $\neg l$ follows from $\theta \wedge$ eff(n).* Due to this observation, it suffices to subtract $\neg\overline{\text{eff}(n)}$: l does not become false in any successor state, unless its opposite is implied. This does *not* hold for more general θ. To see this, re-consider Example 3. We have an axiom $\forall x, y : \neg\text{shipped}(x) \vee \neg\text{invoiceSent}(x, y) \vee \text{paymentExpected}(x)$. We have a state s that satisfies all of shipped(o), invoiceSent(o, i), and paymentExpected(o). We execute a task that falsifies paymentExpected(o). As explained in Example 3, we get two possible transitions: one to a state which additionally falsifies shipped(o), and one to a state which additionally falsifies invoiceSent(o, i). Hence *the only thing that holds true in all possible outcome states is $\neg$paymentExpected*(o). Each of shipped(o) and invoiceSent(o, i) was made false in one transition, but neither follows from the effect of the task. This is in contrast to (*). Intuitively, restricting θ to binary clauses ensures that the side effects are always "deterministic". The main result regarding I-propagation is:

Lemma 1 *[Weber et al. [33]] Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be an executable basic annotated process graph. There exists exactly one I-propagation result I^*. For all $e \in \mathcal{E}$, we have that $l \in I^*(e)$ iff, for all reachable states s where $t_s(e) > 0$, $s \models l$. With fixed arity, the time required to compute I^* is polynomial in the size of $\mathcal{G}$.*

As indicated above, the proof of this lemma is non-trivial; apart from the sketched issue of binary clauses it contains other intricate parts which are not easily explained within a few sentences. Since these arguments are not of importance for the work at hand, we omit them and refer the reader to [33] for details. It is, however, important for the work at hand to note that the time complexity of I-propagation is *low-order* polynomial. The number of different literals $|\mathcal{P}[C]|$ is exponential only in predicate arity, i.e., the maximum number of arguments any predicate has, which is assumed to be fixed. Usually predicates have only 1 or 2, maximally 3 arguments. With binary θ, $\overline{L}$ for any set L of literals can be computed in $O(|\mathcal{P}[C]|^2)$, so $\overline{\text{eff}(n)}$ can be pre-computed for every relevant n in time $O(|\mathcal{N}_T| * |\mathcal{P}[C]|^2)$. Hence an upper bound on the time required for computing I^* is $O(|\mathcal{N}_T| * |\mathcal{P}[C]|^2 + |\mathcal{N}| * |\mathcal{P}[C]| * |\mathcal{E}|)$.

A remark is in order regarding the prerequisite that the process is executable, i.e., preconditions are always satisfied. We have proved in [33] that, without this prerequisite, testing

whether or not a literal is necessarily true at an edge is **NP**-hard. So, like the restriction on binary clauses, this prerequisite cannot be relaxed without losing computational efficiency.[5]

Due to the low-order polynomial time complexity, I-propagation can be expected to be fast – e.g., work in real time within a modeling environment – unless the processes are huge. In our experiments, a process with 17 control-flow nodes (start, end, split, join), 23 task nodes, and 46 edges has been processed in 0.15 seconds on a standard laptop computer with a single-core Pentium (M) CPU running at 1.6 GHz.

Let us say a few words on how I-propagation is extended to processes with structured loops. As one may expect, in the presence of loops a single pass over the process does not do – we need to take into account how changes made later on may feed back into earlier parts of the process, when a part of the process is being repeated. So our extended algorithm does not make use of the $\perp$ symbol to force the fixpoint operation into a single pass. Rather, all edges – except the start edge which is handled as before – are initialized to contain the entire set of literals (including in particular contradictory literals). Each step of the fixpoint process then intersects the old content of the outgoing edges with their new (propagated) content. The propagation steps as such remain the same, with straightforward extensions for start and end nodes of sub-processes (propagating into/out of/ back to the start of the loop). Since the contents of edges decrease monotonically, the number of propagation steps is bounded from above by the number of edges multiplied with the number of different literals. The guarantee given upon termination is exactly as stated in Lemma 1.

We remark that, while the extended algorithm sounds simple and intuitive, its formal write-up is rather complicated. The same goes for the proof of (the equivalent of) Lemma 1, which examines intricate connections between paths in the state space of the process, and paths of propagation steps.

4 Compliance Checking

We leverage on the outcome of I-propagation in order to design compliance checking techniques, determining whether a clausal constraint may be violated while some edge is active. As hinted, we devise an exact check for a particular restricted case where that is possible; we devise approximate checks for a more general case. We proceed in that order.

4.1 Exact Checks for Non-Contradicted Clauses

Clausal constraints can be checked exactly – and in polynomial time – if they are not "contradicted" by the process. Formally, say $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ is a basic annotated process graph, and β is a constraints base. Say $\psi(C_0)$ is a grounded constraint. We say that $\psi(C_0)$ is *contradicted* if there exists a literal $l \in \psi(C_0)$, as well as a task node $n \in \mathcal{N}_T$, so that the negation of l follows from the effect of n, i.e., $\neg l \in \overline{\text{eff}(n)}$. Our observation is that, if this is not the case, then checking compliance with $\psi(C_0)$ can be forumlated in terms of checking compliance with a unit clause:

[5] It may also be puzzling in this context that, as stated above, we designed I-propagation *in order to check* whether a process is executable. However, I^* can actually be used for that purpose: $\mathcal{G}$ is executable iff, for all $n \in \mathcal{N}_T$, pre$(n) \subseteq I^*(IN(n))$. First, if $\mathcal{G}$ is executable then by Lemma 1 we have that I^* captures exactly the literals which are necessarily true, and hence obviously pre$(n) \subseteq I^*(IN(n))$ for all n. Second, if n is not executable but all its predecessors are, then the arguments behind the proof of Lemma 1 can be applied up to n, and we get that I^* handles $IN(n)$ correctly, and hence pre$(n) \not\subseteq I^*(IN(n))$ must hold.

Theorem 1 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be an executable basic annotated process graph, with constraints base β; let $\psi(C_0)$ be a grounded constraint which is not contradicted. Let H be a new predicate symbol, and define $\mathcal{G}' = (\mathcal{N}, \mathcal{E}, \Omega, \alpha')$ by setting, for every $n \in \mathcal{N}_T \cup \{n_0, n_+\}$ where $\overline{\mathit{eff}(n)} \cap \psi(C_0) \neq \emptyset$, $\mathit{eff}'(n) := \mathit{eff}(n) \cup \{H\}$, where eff' denotes the effects assigned by α'. Let I^* be the I-propagation result for $\mathcal{G}'$, and let $e \in \mathcal{E}$ be arbitrary. Then $H \in I^*(e)$ iff, for every state s reachable in $\mathcal{G}$ where $t_s(e) > 0$, $s \models \psi(C_0)$.*

Proof In what follows, we denote reachable states of $\mathcal{G}$ with s, and reachable states of $\mathcal{G}'$ with s'. Clearly, $\mathcal{G}'$ is still executable and basic. Hence we can apply Lemma 1 and we know that $H \in I^*(e)$ iff, for every state s' where $t_{s'}(e) > 0$, $s' \models H$. It hence suffices to show that, for every edge $e \in \mathcal{E}$:

(*) all s where $t_s(e) > 0$ have $s \models \psi(C_0)$ iff all s' where $t_{s'}(e) > 0$ have $s \models H$.

That is, at every edge, $\psi(C_0)$ is "always true" iff H is "always true". This claim is proved by induction over the process structure. The induction base case is the outgoing edge of the start node, $e = n_0$. Here the claim follows by construction. For the inductive case, let $n \in \mathcal{N}$ be arbitrary. As induction hypothesis, we assume that (*) holds for each of n's incoming edges. As induction step, we prove that (*) holds for each of n's outgoing edges. If n is anything but a task node, then this is obvious, since n does not affect the truth of either $\psi(C_0)$ or H. More precisely, for split nodes $\psi(C_0)$ resp. H are always true on the outgoing edges iff they are always true on the ingoing edge; for parallel join nodes, $\psi(C_0)$ resp. H are always true on the outgoing edge iff they are always true on at least one ingoing edge; for xor join nodes, $\psi(C_0)$ resp. H are always true on the outgoing edge iff they are always true on all ingoing edges.

Say n is a task node. First, assume that all s where $t_s(IN(n)) > 0$ have $s \models \psi(C_0)$. By induction hypothesis, the same holds for all s' and H. Since $\psi(C_0)$ is not contradicted, and by construction, we get the same properties for $OUT(n)$, showing (*) as required.

Second, assume that $\overline{\mathrm{eff}(n)} \cap \psi(C_0) \neq \emptyset$. Then, since literals from $\psi(C_0)$ are never invalidated, all s where $t_s(OUT(n)) > 0$ have $s \models \psi(C_0)$. By construction, n makes H true, which is also never invalidated, and hence all s' where $t_{s'}(e) > 0$ have $s \models H$, showing (*) as required.

We are left with the case where $\overline{\mathrm{eff}(n)} \cap \psi(C_0) = \emptyset$ and there exists s where $t_s(IN(n)) > 0$ and $s \not\models \psi(C_0)$. By induction hypothesis, there exists s' where $t_{s'}(IN(n)) > 0$ and $s' \not\models H$. In s, we can execute n (note here the prerequisite of executability) and reach a state s_1 that has $t_{s_1}(OUT(n)) > 0$ and $s_1 \not\models \psi(C_0)$. Similarly, in s' we can execute n (note here the prerequisite of executability) and reach a state s'_1 that has $t_{s'_1}(OUT(n)) > 0$ and $s'_1 \not\models H$. Hence the outgoing edge has neither $\psi(C_0)$ nor H necessarily true, and (*) holds again. This concludes the argument.

Theorem 1 can be exploited for compliance checking, in the obvious manner. That is, we define our first compliance checking method as follows:

(A) Given a grounded constraint $\psi(C_0)$, construct the process $\mathcal{G}'$ as per the claim of Theorem 1, and run I-propagation on $\mathcal{G}'$. From the resulting I^*, for every edge e one can read directly whether or not $\psi(C_0)$ may be violated while in a state where e carries a token.

Theorem 1 and method (A) carry over directly to processes with structured loops, with exactly the same way of constructing $\mathcal{G}'$. The proof of Theorem 1 in this setting uses the same core arguments, except that now we need to add an induction over process structure,

first proving (*) for process graphs with no sub-graphs (i.e., with no loops), then considering processes where all sub-graphs satisfy (*) by induction hypothesis.

The following example illustrates method (A).

Example 6. *Reconsider our running example, and the grounded constraint* $\neg order(o) \vee \neg received(o) \vee rejected(o) \vee paid(o)$. *We wish to check at which points in the process – at which edges – this constraint is satisfied. First, note that the constraint is not contradicted: consider Table 1 or Fig. 3 to verify that none of its literals is negated by the effect of any node, other than the start node. Hence, we can apply method (A). We introduce a new predicate* H *which we insert into the effect of every node that achieves one of the literals in the constraint. These task nodes are Reject Order (which achieves* $rejected(o)$*) and Accept Payment (which achieves* $paid(o)$*). By I-propagation, we get* H *at the outgoing edges of these two task nodes. Consequently, we get* H *on both ingoing edges of the xor join node. Taking the intersection there, we get* H *on the outgoing edge of the xor split – reflecting the fact that the constraint has been satisfied in either case – and we finally get it on the stop edge of the process. For all other edges* e, H *is not contained in* $I^*(e)$. *This correctly reflects the points in the process where the constraint may be violated (where there exists an execution violating the constraint while the respective edge carries a token) and where this is never the case.*

It is important to note that method (A) really "works" only if the grounded constraint is not contradicted. The following is an example where that prerequisite is not given.

Example 7. *Consider a modified example where, between Accept Payment and the parallel join, we insert another task node, Refund Payment, with effects* $\neg paymentAccepted(i)$ *and* $\neg paid(o)$. *This is depicted in Fig. 4.*

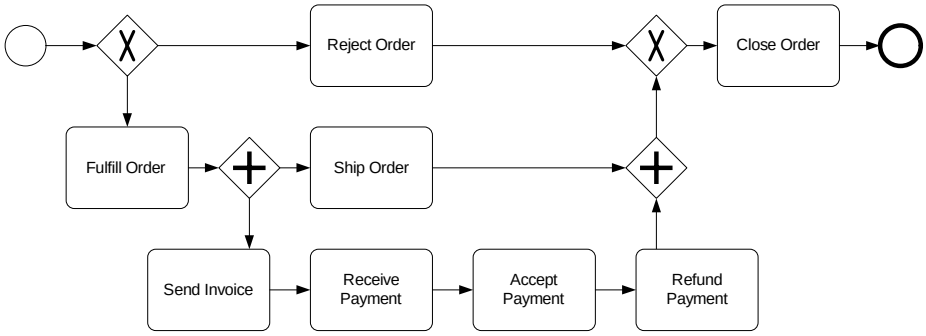

Fig. 4 Modified running example including a new "Refund Payment" task.

Say that we have the constraint $\forall x, y : \neg invoiceSent(x, y) \vee paymentExpected(x) \vee paymentAccepted(y)$. *That is, whenever an invoice has been sent the corresponding payment needs to be either expected or accepted. We next consider the grounded constraint* $\neg invoiceSent(o, i) \vee paymentExpected(o) \vee paymentAccepted(i)$. *This constraint is contradicted, because Accept Payment negates* $paymentExpected(o)$. *Say we nevertheless try to apply method (A). Up to Receive Payment, we get the correct result simply because none of the literals has been contradicted so far. After Accept Payment, method (A) still gets the correct result, namely that the constraint is true on the outgoing edge,* $H \in I^*(e)$ *where* e *is the outgoing edge of Accept Payment. However, this correct result is just a coincidence – method (A) "gets lucky". To see this, consider that method (A) completely*

ignores how Accept Payment contradicts the constraint, namely by the effect that falsifies $paymentExpected(o)$*. Since, before Accept Payment, the constraint was true only due to that fact, the constraint would now actually be violated – unrecognized by method (A) – were it not for the additional effect of Accept Payment that establishes* $paymentAccepted(i)$*.*[6] *In the next task node, Refund Payment, there is no such lucky coincidence. The task node contradicts* $paymentAccepted(i)$*, and hence the constraint is violated at its outgoing edge. Ignoring the contradiction, method (A) does not notice this. I-propagation assigns* H *to the outgoing edge of Refund Payment, and we wrongly conclude that the constraint will always be complied with at that point.*

4.2 Approximate Checks for Contradicted Clauses

It is as yet an open question whether contradicted clauses can be checked exactly in polynomial time. Herein, we instead provide two approximation methods. The methods are dual; one guarantees to find only non-compliances (but not necessarily all), the other guaranteeing to find all non-compliances (but may report spurious ones). Both methods are based on the information provided by I-propagation. However, we generalize the method: in difference to before, we do not require the process to be executable. As it turns out, even in this situation I-propagation gives the guarantee that we need for our approximation techniques. Namely, we can prove the following variant of Lemma 1:

Lemma 2 *Let* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ *be a basic annotated process graph, and let* I^* *be the I-propagation result. Let* $e \in \mathcal{E}$ *be arbitrary, and let* $l \in I^*(e)$*. Then, for all reachable states* s *where* $t_s(e) > 0$*, we have* $s \models l$*.*

Proof Let $\mathcal{G}' = (\mathcal{N}, \mathcal{E}, \Omega, \alpha')$ be like $\mathcal{G}$ except that $\text{pre}'(n)$ has been set to $\emptyset$ for all $n \in \mathcal{N}_T$. Since $\mathcal{G}'$ does not alter the structure of $\mathcal{G}$, there is a 1-to-1 correspondence between the states reachable in $\mathcal{G}$ and the states reachable in $\mathcal{G}'$. We denote corresponding states with s and s', in the obvious manner. Further, for $e \in \mathcal{E}$, denote by $\bigcap e$ the set of literals true in all states s where $t_s(e) > 0$, and denote by $\bigcap' e$ the set of literals true in all states s' where $t_{s'}(e) > 0$.

Obviously, $\mathcal{G}'$ is executable. Hence we can apply Lemma 1, and get that I^* is correct for $\mathcal{G}'$: for all $e \in \mathcal{E}$, we have that $\bigcap' e = I^*(e)$. Hence it suffices to show that:

$$\text{(*) for every } e \in \mathcal{E}, \bigcap e \supseteq \bigcap' e.$$

We prove (*) by means of proving the following:

$$\text{(**) for every } e \in \mathcal{E}, \{s \mid t_s(e) > 0\} \subseteq \{s' \mid t_{s'} > 0\}$$

That is, the states reachable in $\mathcal{G}$ (at e) are a subset of those reachable in $\mathcal{G}'$. Obviously, this implies (*). It is easy to prove (**) by induction over the process structure. The base case, outgoing edge of the start node, is obvious (the sets of states are identical). The inductive case is likewise obvious in all cases except task nodes. As for the latter, if (**) holds on the incoming edge then it also holds on the outgoing edge due to the role that preconditions play in the semantics as per Definition 4: if the precondition is not satisfied, then a transition is disallowed; otherwise, the precondition has no influence. This concludes the argument.

[6] Note that this "lucky coincidence" suggests a simple generalization of non-contradicted constraints: a task node may negate one of the constraint's literals as long as it makes another one true.

In words, Lemma 2 says that, if we ignore preconditions – if we act as if the process is executable – then we can only make it harder for a literal to be always true. Hence the outcome of I^* is conservative, in that sense. Exactly the same claim, with exactly the same proof arguments, applies to processes with structured loops.

One of our approximation methods is based directly on I^*. The other method is based on the dual notion of U^*. This is defined as follows. Say $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ is a basic annotated process graph with constants C. If I^* is the I-propagation result, then for $e \in \mathcal{E}$ we denote $U^*(e) := \{l \mid l \in \mathcal{P}[C], \neg l \notin I^*(e)\}$. In words, $U^*(e)$ is the set of literals that are not contradicted by $I^*(e)$. By Lemma 2, we immediately get:

Lemma 3 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ be arbitrary, and let l be a literal so that there exists a reachable state s where $t_s(e) > 0$ and $s \models l$. Then we have $l \in U^*(e)$.*

Proof Assume to the contrary of the claim that $l \notin U^*(e)$. Then, by construction, we have $\neg l \in I^*(e)$. By Lemma 2, this means that, for all reachable states s where $t_s(e) > 0$, $s \models \neg l$. This is a contradiction to the prerequisite, and concludes the argument.

In that sense, U^* is conservative – includes all literals that may possibly be true – since I^* is conservative in the dual way (obviously, the same holds true for processes with structured loops). This directly leads to the main result underlying our approximation techniques:

Theorem 2 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph; let I^* be the I-propagation result. Then, for all $e \in \mathcal{E}$:*

1. *If there exists a grounded constraint $\psi(C_0)$ such that for all $l \in \psi(C_0) : \neg l \in I^*(e)$, then every reachable state s with $t_s(e) > 0$ is a non-compliance.*
2. *If there exists a non-compliant state s with $t_s(e) > 0$, then there exists a grounded constraint $\psi(C_0)$ such that for all $l \in \psi(C_0) : \neg l \in U^*(e)$.*

Proof Obviously, any state s is a non-compliance iff it violates one of the grounded constraints. Hence, the claim is a simple consequence of Lemmas 2 and 3. First, if for all $l \in \psi(C_0) : \neg l \in I^*(e)$, then by Lemma 2 every reachable state s with $t_s(e) > 0$ violates all of $\psi(C_0)$'s literals. Second, if s violates $\psi(C_0)$, then, for every $l \in \psi(C_0)$, we have $s \models \neg l$ and hence, by Lemma 3, $\neg l \in U^*(e)$. This concludes the argument.

Theorem 2 immediately suggests our two approximate methods: for every edge e, check whether there exists a grounded constraint $\psi(C_0)$ such that

(B) for all $l \in \psi(C_0) : \neg l \in I^*(e)$, or
(C) for all $l \in \psi(C_0) : \neg l \in U^*(e)$.

If (B) applies, then we know for sure that a non-compliant state exists, presuming that a state activating e is reachable. If (C) applies, then we know that a non-compliant state *may* exist; by contra-position, if (C) does not apply for any e and $\psi(C_0)$ then we know that the process complies with the constraints base. Clearly, if all predicates have a fixed arity and if the number of ground constraints is polynomial (i.e., if the number of variables in any constraint is fixed), then all the tests can be performed in polynomial time.

Since, as stated, Lemmas 2 and 3 carry over directly to processes with structured loops, the same is true of Theorem 2 as well as methods (B) and (C).

The advantage of tests (B) and (C), over method (A) as defined above, is that they do not require the constraint to be non-contradicted, and neither do they require the task nodes

to be executable. We illustrate this, and the difference between tests (B) and (C), with some examples. We start with the example of a contradicted clause.

Example 8. *Reconsider the modified example from Fig. 4, with the grounded constraint* $\neg invoiceSent(o,i) \vee paymentExpected(o) \vee paymentAccepted(i)$. *As explained above, with method (A) we come to the wrong conclusion that this constraint is always satisfied at the outgoing edge of Refund Payment. However, method (B) detects the violation. If* e *is the outgoing edge of Refund Payment, then we get* $\{invoiceSent(o,i), \neg paymentExpected(o), \neg paymentAccepted(i)\} \subseteq I^*(e)$. *Hence test (B) applies and we have proved that, whenever Refund Payment has been executed, the constraint is violated. (This could be repaired by stating explicitly that refund Payment retracts the invoice, i.e., by giving it the effect* $\neg invoiceSent(o,i)$.*) Of course, test (C) applies as well.*

We next modify this example some further to illustrate how test (B) may fail to detect a non-compliance, which may never happen for test (C).

Example 9. *Say we make Refund Payment an optional node, i.e., in difference to before we insert it as one of the branches of an xor construct. This is depicted in Fig. 5.*

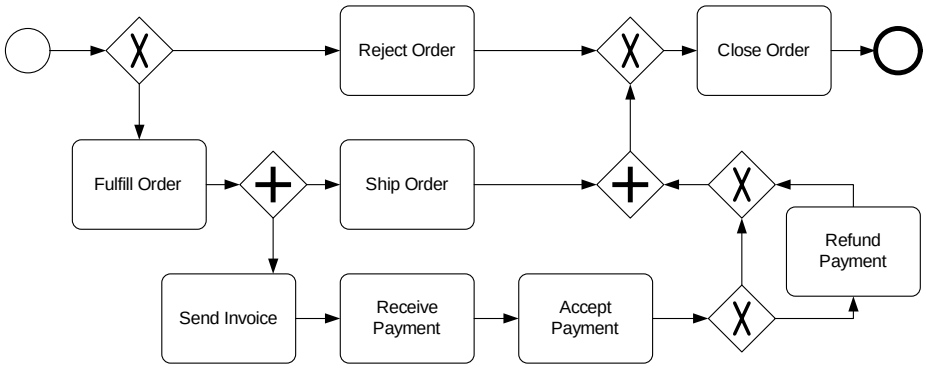

Fig. 5 Modified running example including an optional "Refund Payment" task.

Again, consider the grounded constraint $\neg invoiceSent(o,i) \vee paymentExpected(o) \vee paymentAccepted(i)$. *Test (B) will, as before, correctly detect that this constraint is violated at the outgoing edge of Refund Payment. However, that information is lost at the outgoing edge* e *of the xor join: at this point in the process, Refund Payment has not necessarily been executed. This is reflected in the fact that (among other things)* $\neg paymentAccepted(i) \notin I^*(e)$. *Hence test (B) does not apply for* e. *This is incorrect since, of course, it* may *happen that* e *carries a token while the constraint is violated, namely in the cases where Refund Payment was indeed executed. Test (C) correctly detects this possibility. None of* $invoiceSent(o,i)$, $\neg paymentExpected(o)$, *or* $\neg paymentAccepted(i)$ *are contradicted by* $I^*(e)$, *hence they are all contained in* $U^*(e)$, *hence test (C) applies.*

We conclude this section with a final example illustrating the role of preconditions, and how test (C) may wrongly report correct behavior as non-compliant.

Example 10. *Say we give Close Order the precondition* $paid(o)$. *Obviously, the task is then not executable anymore because its precondition is violated in case the order has been rejected. I-propagation ignores this fact, and consequently we have* $I^*(e) = \{order(o),$

$received(o), closed(o)\}$ *as before (where* e *is the outgoing edge of Close Order). However, really, every reachable state activating* e *also satisfies* $paid(o)$*, simply because the precondition admits only such states. So we see that – as guaranteed by Lemma 2 –* $I^*(e)$ *is a subset of the true literals; a proper subset, in this case.*

The literal missing from $I^*(e)$ *results in a misbehavior of method (C). Say we simply want to check whether, at the end of the process,* $paid(o)$ *(the grounded constraint containing only this single literal) is satisfied. Test (B) does not apply because it cannot be deduced that* $paid(o)$ *is necessarily false. However, test (C) applies because I-propagation mistakenly came to the conclusion that* $paid(o)$ may *be false. (Remember here that, as stated before, testing truth of even single literals is* **NP***-hard in the presence of non-executable task nodes [33].)*

Summing up, we devised three methods for compliance checking: one exact method (A) for clauses which are not contradicted, one sound but incomplete approximate method (B), and one unsound but complete approximate method (C). Note that, due to their respective properties, it makes sense to schedule these methods in a certain way. If the constraint is non-contradicted, then one should run only (A). Else, one should first try (B) which guarantees to only flag edges that are actually erroneous. Once (B) does not report any non-compliances anymore, one should try method (C); if that completes without reporting errors, then it is certain that the process is fully compliant. We reiterate that exactly the same methods apply, with exactly the same guarantees, to processes with structured loops.

5 Diagnosis

In order to efficiently support the user in compliance checking, it is of high value to be able to point out the sources of an error. Since we check the compliance rules against summaries of the logical states that may occur, we do so by tracing how the logical states leading to non-compliance may come into being. At base, there are three questions we are interested in answering: (1) What are the reasons for a literal l to be *necessarily* true at an edge e? (2) What are the reasons for a literal l to be *possibly* true at an edge e? (3) What are *possible* reasons for a literal l to be *possibly* true at an edge e? Based on answers to these questions, we can provide diagnosis techniques for the various compliance checking methods introduced in the previous section.[7] In what follows, we first include a sub-section detailing how questions (1), (2), (3) can be answered. Then another sub-section explains how this information can be employed for diagnosing non-compliances.

5.1 Tracing Literals

Consider first question (1): what are the reasons for a literal l to be necessarily true at an edge e? The answer is, all nodes that cause l and/or that belong to a path between such a cause and e. The set of these nodes, $R^{\top}(e, l)$, can be computed as follows.

Definition 8 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ and let l be a literal. If $l \notin I^*(e)$, then $R^{\top}(e, l) := \emptyset$. Else:

[7] Note that we only talk about "reasons for a literal being true", not about "reasons for a literal being false". We get the latter for free due to the duality between positive and negative literals. For example, if we want to ask "what are the reasons for a literal l to be necessarily false at an edge e?" then this is the same as question (1) for $\neg l$.

1. $n \in R^{\top}(e,l)$ where n is the node with $e \in OUT(n)$;
2. if $n \in (\mathcal{N}_{XS} \cup \mathcal{N}_{PS}) \cap R^{\top}(e,l)$, then $n' \in R^{\top}(e,l)$ where n' is the node with $OUT(n') \cap IN(n) \neq \emptyset$;
3. if $n \in \mathcal{N}_T \cap R^{\top}(e,l)$, and $l \notin \overline{\mathit{eff}(n)}$, then $n' \in R^{\top}(e,l)$ where n' is the node with $OUT(n') \cap IN(n) \neq \emptyset$;
4. if $n \in (\mathcal{N}_{XJ} \cup \mathcal{N}_{PJ}) \cap R^{\top}(e,l)$, then $n' \in R^{\top}(e,l)$ where n' is any node so that there exists $e' \in OUT(n') \cap IN(n)$ where $l \in I^*(e')$.

Definition 8 is best understood in terms of defining $R^{\top}(e,l)$ by a certain form of backward chaining from e. Starting at e, the chaining includes into $R^{\top}(e,l)$ all nodes on whose outgoing edges l is contained in I^*; it stops when it reaches a task node that causes l to be true. It is easy to see that this set of nodes indeed captures the reasons for l being necessarily true at e, in the following sense:

Proposition 1 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ and let l be a literal such that $l \in I^*(e)$. Define $\mathcal{G}' = (\mathcal{N}, \mathcal{E}, \Omega, \alpha')$ which is like $\mathcal{G}$ except that $\mathit{eff}(n) := \emptyset$ for all $n \notin R^{\top}(e,l)$. Let $I^{*\prime}$ be the I-propagation result for $\mathcal{G}'$. Then $l \in I^{*\prime}(e)$.*

In words, $R^{\top}(e,l)$ includes enough nodes to make l true at e, even when ignoring the effects of all other nodes. This is simply because Definition 8 backchains from e until it has collected all potentially relevant task nodes. One may wonder whether $R^{\top}(e,l)$ is minimal in that property, i.e., whether removing any node from it will necessarily disvalidate Proposition 1. This is not the case: for parallel joins n, l may be contained in $I^*(OUT(n))$ even if it is contained in $I^*(e')$ for only a subset of the edges $e' \in IN(n)$. Definition 8 collects all these e'. To be minimal, it would have to select just a single such e'. However, that would not be appropriate for diagnosis reasons since we are interested in *all* reasons why l is necessarily true at e.

Obviously, $R^{\top}(e,l)$ can be computed in low-order polynomial time. We finally remark that $R^{\top}(e,l)$ never includes a task node n where $\neg l \in \overline{\mathit{eff}(n)}$. In such a case, clearly we cannot have $l \in I^*(OUT(n))$, whereas it is easy to see that this holds for any $n \in R^{\top}(e,l)$: this is an invariant over the backchaining steps performed in Definition 8.

For processes with structured loops, we can simply extend Definition 8 by: handling the start nodes of repeatable sub-processes like xor joins (control may come here either from outside the loop, or from its end); and handling the end nodes of repeatable sub-processes like xor splits (control may go either outside the loop, or back to its start). Proposition 1 and the rest of our discussion above then carry over exactly as stated.

Example 11. *Reconsider our running example from Fig. 2 and Table 1, the constraint $\neg order(o) \vee \neg received(o) \vee rejected(o) \vee paid(o)$, and the literal H introduced by test (A). We have $H \in I^*(e_+)$, i.e., the constraint is guaranteed to hold at the end of the process. Constructing $R^{\top}(e_+, H)$, we include: Close Order; the xor join; Reject Order; the parallel join; Accept Payment. This sub-graph correctly reflects the reason why the constraint is necessarily true at e_+.*

Consider now question (2) from above: what are the reasons for a literal l to be possibly true at an edge e? Here we consider the case where, in difference to question (1), $l \notin I^*(e)$; we only have $l \in U^*(e)$. What we want to know is, which nodes contribute to making l true at e? The answer is similar to before. We define:

Definition 9 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ and let l be a literal. If $l \notin U^*(e)$, then $R^{\triangleright}(e,l) := \emptyset$. Else:

1. $n \in R^{\triangleright}(e,l)$ where n is the node with $e \in OUT(n)$, or $IN(n)$ is parallel to e and $l \in \overline{\text{eff}(n)}$;
2. if $n \in (\mathcal{N}_{XS} \cup \mathcal{N}_{PS}) \cap R^{\triangleright}(e,l)$, then $n' \in R^{\triangleright}(e,l)$ where n' is the node with $OUT(n') \cap IN(n) \neq \emptyset$;
3. if $n \in \mathcal{N}_T \cap R^{\triangleright}(e,l)$, and $l \notin \overline{\text{eff}(n)}$, then $n' \in R^{\triangleright}(e,l)$ where n' is the node with $OUT(n') \cap IN(n) \neq \emptyset$;
4. if $n \in (\mathcal{N}_{XJ} \cup \mathcal{N}_{PJ}) \cap R^{\triangleright}(e,l)$, then $n' \in R^{\triangleright}(e,l)$ where n' is any node so that there exists $e' \in OUT(n') \cap IN(n)$ where $l \in U^*(e')$.

This is like Definition 8, with two differences. First, the backchaining starts not only from e but also from any parallel task nodes achieving l. (Note however that the latter nodes will not generate any further chaining since the rule for task nodes stops when l is an effect.) Second, for join nodes, we include predecessors where $l \in U^*(e')$ rather than $l \in I^*(e')$ – this accounts for the fact that l isn't necessarily in I^* in the first place, i.e., at e itself. Similarly as for $R^{\top}(e,l)$, $R^{\triangleright}(e,l)$ suffices to make l possibly true at e, i.e., we have:

Proposition 2 *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ and let l be a literal such that $l \in U^*(e)$. Define $\mathcal{G}' = (\mathcal{N}, \mathcal{E}, \Omega, \alpha')$ which is like $\mathcal{G}$ except that eff$(n) := \emptyset$ for all $n \notin R^{\triangleright}(e,l)$. Let $I^{*\prime}$ be the I-propagation result for $\mathcal{G}'$. Then $l \in U^{*\prime}(e)$.*

This holds due to same arguments as given above for Proposition 1. Likewise, minimality is not given, $R^{\triangleright}(e,l)$ can be computed in low-order polynomial time, and a node n with $\neg l \in \overline{\text{eff}(n)}$ can never be part of $R^{\triangleright}(e,l)$. Note further that, for any e and l, $R^{\triangleright}(e,l) \supseteq R^{\top}(e,l)$. This is because I^* is always a subset of U^*. Again, for processes with structured loops, the same properties are achieved by handling the start nodes of repeatable subprocesses like xor joins, and the end nodes of repeatable sub-processes like xor splits.

Example 12. *Reconsider our running example from Fig. 2 and Table 1, in a modification that has the literal $\neg paid(o)$ in the annotation of the start node. We have $paid(o) \in U^*(e_+)$, i.e., the literal may be true at the end of the process. Constructing $R^{\triangleright}(e_+, paid(o))$, we include: Close Order; the xor join; the parallel join; Accept Payment. Clearly, these are exactly the nodes responsible for the possibility to have $paid(o)$ true in the end.*[8]

Consider finally question (3) from above: what are the possible reasons for a literal l to be possibly true at an edge e? What we target with this question – what we mean with "possible reasons" – is the set of nodes that *could* in principle contribute to making l true at e, but that do not do so, due to the process structure. We define this set of nodes as:

Definition 10 Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ be a basic annotated process graph, and let I^* be the I-propagation result. Let $e \in \mathcal{E}$ and let l be a literal. Then $R^{\triangleleft}(e,l) := \{n \in \mathcal{N}_T \mid l \in \overline{\text{eff}(n)}, n \notin R^{\triangleright}(e,l)\}$.

A node n may be in $R^{\triangleleft}(e,l)$ because either: a node n' in between n and e falsifies l; or n is ordered after e; or e and n belong to alternative (xor'ed) parts of the process.

Example 13. *Reconsider our running example. Say e is the outgoing edge of Accept Payment. Then $R^{\triangleleft}(e, closed(o))$ contains only the task node Close Order.*

[8] If the start node does not have the effect $\neg paid(o)$, then this is formally interpreted to mean that $paid(o)$ might be true at the beginning already. Consequently, $R^{\triangleright}(e_+, paid(o))$ collects all nodes on paths from n_0 to n_+ that do not contain Accept Payment – i.e., all nodes except Send Invoice and Receive Payment.

It is of course debatable whether these or other definitions are most suitable for diagnosis purposes. That is true especially of our answer to question (3), which may seem rather arbitrary. Then again, it appears difficult to come up with a more informed technique, since we cannot look into the head of the human modeler and guess what she really meant to do. For a better understanding of these issues, one needs to run large-scale empirical experiments with alternative options. This is left for future research.

5.2 Tracing Non-Compliance

Equipped with the literal tracing methods from above, we can now relatively easily assemble some methods for tracing non-compliances. We distinguish the different possible outcomes of our compliance checking methods. Say $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Omega, \alpha)$ is a basic annotated process graph and e is an edge.

1. **Test (A) applies, non-contradicted constraint $\psi(C_0)$ found to be violated.** Say we constructed $\mathcal{G}'$ according to Theorem 1, ran I-propagation on $\mathcal{G}'$, found that $H \notin I^{*\prime}(e)$, and hence proved that there exists a reachable state s where e is active and $\psi(C_0)$ is violated.
   In this situation, obviously it must be the case that $\psi(C_0) \cap I^*(e) = \emptyset$. We distinguish between two kinds of literals in $\psi(C_0)$: those that are certainly false, $\psi(C_0) \setminus U^*(e)$; and those that may be true, $\psi(C_0) \cap U^*(e)$.
   For each $l \in \psi(C_0) \setminus U^*(e)$, most importantly we want to know why it is false at e. That is, we highlight the set of nodes $R^{\top}(e, \neg l)$. If desired, the user can be given the option to also highlight $R^{\triangleleft}(e, l)$, i.e., the nodes that could have contributed to making l true, but that don't for some flaw in the process structure.
   For the literals $l \in \psi(C_0) \cap U^*(e)$, which may indeed be true, the first thing we are interested in is which nodes contribute to making l true at e, i.e., $R^{\triangleright}(e, l)$. If desired, the user can also highlight $R^{\triangleright}(e, \neg l)$ – the nodes contributing to make l false at e – as well as $R^{\triangleleft}(e, l)$ – the nodes that could have contributed to making l true.
2. **Test (B) applies, contradicted constraint $\psi(C_0)$ found to be violated.** Say we ran I-propagation on $\mathcal{G}$, found that for all $l \in \psi(C_0) : \neg l \in I^*(e)$, and hence proved that every reachable state s where e is active violates $\psi(C_0)$.
   Most importantly, for every $l \in \psi(C_0)$ we want to highlight the reason for being false, i.e., $R^{\top}(e, \neg l)$. If desired, the user can be given the option to also highlight $R^{\triangleleft}(e, l)$.
3. **Test (C) applies, contradicted constraint $\psi(C_0)$ found to be possibly violated.** Say we ran I-propagation on $\mathcal{G}$ and found that for all $l \in \psi(C_0) : \neg l \in U^*(e)$; so we could not disprove the existence of a reachable state where e is active and $\psi(C_0)$ is violated.
   Again, we distinguish between $\psi(C_0) \setminus U^*(e)$ and $\psi(C_0) \cap U^*(e)$. For the former literals l, which are known to be false, we highlight $R^{\top}(e, \neg l)$ and, if desired, $R^{\triangleleft}(e, l)$. The latter literals are neither known to be true nor known to be false. The first thing of interest is, hence, $R^{\triangleright}(e, l)$, the nodes that contribute to making l true. If desired, the user can also highlight $R^{\triangleright}(e, \neg l)$ and/or $R^{\triangleleft}(e, l)$.

Equipped with the above techniques, one can not only detect (potentially) non-compliant parts of the process automatically, but also conveniently have a look at the reasons for that. Clearly, the methods apply also to processes with structured loops, when using the appropriately extended versions of Definitions 8 and 9. An empirical analysis of the techniques is beyond the scope of this paper.

6 Related Work

There are two main lines of related work. First, there exist some works on specification of compliance for business process models, some of which also provide checking methods. Second, a relation to work in Petri nets arises because, to some extent, our formalism can be compiled into such nets. We discuss the two lines of related work in that order.

6.1 Compliance Specification and Checking

While the issue of compliance of business process models with normative specifications started receiving attention in the past few years, the study of how to formally represent normative specifications has a long history and a full detailed comparison with the vast literature is out of the scope of the paper. In the context of this paper it is worth remembering that the use of logical clauses for normative specifications goes back to Kowalski and Sergot [31], who proposed to encode regulations and normative systems as logic programs. More recently [14] proposed to use Event Calculus and logic programming as executable specifications for contracts, though the main focus is on monitoring the performance of a contract.

[16] considers an approach similar to ours, where the tasks of a business process model, written in BPMN, are annotated with the effects, and a technique to propagate and accumulate the effects from a task to a successive contiguous one is proposed. The technique is designed to take into account possible conflicts between the effects of tasks and to determine the degree of compliance of a BPMN specification. Effects are accumulated in Semantic Process Networks (SPN), which are nested structures with set of literals. The nested structures corresponds to OR splits in a business process. Contrary to what we do, this approach does not determine at design time whether a business process is compliant. Further, the approach cannot handle loops, and may exhibit exponential runtime behavior (the size of the SPN may grow exponentially in the size of the process).

[10] investigates compliance in the context of agents and multi-agent systems based on a classification of paths of tasks. It defines patterns according to which the behaviour of an agent conforms to a protocol.

The approach of [25,26] 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 [4]. Contrary to our approach, compliance is limited to constraints on relationships between tasks in a process. In fact, mutual exclusivity and dependency constraints can be simulated using a subset of our framework (using only preconditions/effects, with an empty ontology). Namely, for each task we introduce a ground literal corresponding to the task, forming the task's effect. To model exclusion between a and b, we add $\neg a$ to the precondition of b, and we add $\neg b$ to the precondition of a. To model that b depends on a, we simply add a to the precondition of b. Hence the model considered by [25,26] can be viewed as a special case of our framework. On the algorithms side, [25,26] consider exploration of execution traces, and propose techniques to speed up compliance checking for adapted processes, based on excluding paths that are not affected by the changes made in the adaptation. Clearly, this is very different from our work, which identifies polynomial-time propagation algorithms for restricted classes of processes.

In a number of approaches, no semantic annotations are added to the business process model, and hence compliance is limited to the structure and relationships of tasks in the process. [7] uses BPMN-Q, a visual language based on BPM to query a business process model by matching a process graph to a query graph, to express compliance rules as queries. After this step, the procedure retrieves BPMN sub-graphs, that are then manipulated and reduced, to be then transformed into formulas in temporal logic (PLPL linear past temporal logic). The temporal logic formulas are processed using model checking to verify compliance. Similarly, [23] proposes the use of LTL (linear temporal logic) and model checking to verify the compliance of BPEL processes. [29] proposes Concurrent Transaction Logic to model the states of a workflow and presents some algorithms to determine whether the workflow is compliant with a contract. The algorithms take advantage of features of the logic to apply graph transformations, identifying inconsistent patters among the process nodes. The process is compliant with a contract if the constraints imposed by the contract do not generate inconsistencies.

A limitation of most of the approaches to compliance, including the one presented herein, is that they do not natively deal with the normative aspects of compliance, i.e., whether a logical statement refers to an obligation, a permission, or a prohibition. Also, preferences between obligations are often difficult to express. An exception is in previous work [20] from one of the authors of this paper, proposing to use FCL (Formal Contract Language). FCL is a simple rule based logic enriched with deontic operators to specify the obligations a process has to fulfill. [20] argues that compliance is the relationship between the potential execution states of a process and the normative specifications. We have taken first steps towards extending our work to incorporate FCL [18]. This essentially involves extended versions of the propagation mechanisms presented herein. The extended algorithms keep track of the history of particular facts/constraints, in order to determine temporal aspects such as whether a constraint always holds after a particular condition became true. Contrary to the paper at hand, the algorithms are in an early stage and have not yet been analyzed formally.

6.2 Petri Nets

Petri net theory has come up with a wealth of complexity results for various classes of Petri nets, including in particular tractability results for a number of restricted classes. One can apply some of these results to annotated process graphs, via compiling such graphs into Petri nets. However, the results obtainable in this way are substantially weaker than what we provide herein.

How can annotated process graphs be compiled into Petri nets? First, consider the case where there are no ontology axioms. For this case, a straightforward compilation exists. Encode each task as a transition, and encode edges as places. Joins and splits can then be encoded in a straightforward way, using the rules in [2]. Likewise, loops, i.e., transitions into and out of sub-graphs, are encoded in the straightforward fashion. Next, enumerate all facts that can be built from the predicates and constants. Create an additional place for each fact, as well as one for its negation. Add an arc for each precondition/effect literal to the respective place; similarly, encode xor/loop conditions.

If ontology axioms are present, such a compilation is not possible, at least not in a straightforward/natural way. Petri nets do not cater for a “minimal change semantics” of transitions between states. Note that this is quite fundamental. As indicated earlier, for non-binary theories we have proved in our previous work [33] that reasoning about state transi-

tions with such a semantics is computationally hard. To cater for this semantics, one would hence have to somehow encode the "minimal change" into a specialized class of Petri nets with worst-case exponential behavior, and then include an instance of that class into every transition of the overall compiled net. If, on the other hand, the axioms are all binary, then (as we also show in [33]) a compilation preserving the relevant properties of I-propagation (c.f. Lemma 1) can be done by replacing task node effects with their deductive closure ($\overline{\text{eff}}$ in our notation herein).

Apart from the process structure, we need to express our compliance checking task in terms of Petri net queries. Assume a grounded consraint $\psi(C_0)$ and an edge e. Compliance with $\psi(C_0)$ at e in the annotated process model is equivalent to the question whether, whenever e carries a token, at least one of the places in $\psi(C_0)$ (i.e., places encoding the respective literals) carry a token as well. Hence we need to be able to test whether a given place p implies the disjunction of a set of other places $p_1, \ldots, p_k$. This is a rather unusual query for Petri nets. For the restricted case where $k = 1$, it is related to what has been termed "implicit places" (see e.g. [9,3,15]). An implicit (or "redundant") place is a place p_1 that always carries a token if any other place p does, where p and p_1 occur together in the input of any transition. Note that this notion refers to *all* transitions and places p, while what we are interested in is the connection (if any) between $p_1, \ldots, p_k$ and *one particular* place p. It is an open question whether techniques for detecting implicit places can be adapted to perform this kind of test, in particular for $k > 1$. We remark that the only known polynomial-time technique to detect implicit places is the detection of "structural implicit places" [9,3], which are a special case of implicit places, hence providing for a sound but incomplete checking method.

What we can derive are two tractable classes for the simpler question where we only want to check whether a constraint may be violated globally – regardless of any edge at which that may happen. The tractability results are based on restricting the process in a way so that the compiled Petri net becomes free-choice [13], respectively conflict-free [22].[9] To encode a violation of the entire constraint, we introduce a new place p_0 with a transition t that takes and replaces tokens from all of $\neg l$ for $l \in \psi(C_0)$, and puts a token on p_0. Clearly, p_0 is reachable iff there exists a reachable marking that has tokens on all of the literals $\neg l$.[10] We have:

(1) Say that a literal l is *consumed* by node n if either $l \in \text{pre}(n)$ or $\neg l \in \text{eff}(n)$. If every literal l is consumed by at most one task node, and no literal $\neg l$ for $l \in \psi(C_0)$ is consumed by any task node, then the compiled Petri net is free-choice.
(2) If the process has no loops and no xor splits, and for every literal l and task node n we have that $\neg l \notin \text{eff}(n)$, then the compiled Petri net is conflict-free.

Both for free-choice and conflict-free Petri nets, it can be decided in polynomial time whether there exists a reachable marking activating a given place. Hence, (1) and (2) identify tractable classes for global constraint checking. Note that both of these classes restrict the constraint to be non-contradicted, and are hence sub-classes – rather restricted sub-classes, at that – of what is handled as per our Theorem 1.

[9] In free-choice nets, for every two transitions either the input places are disjoint, or identical. In conflict-free nets, every place either is on the input of only one transtion, or is on the output of all such transitions.

[10] One can extend this method to encode compliance checking at an edge e, simply by assigning e as another input place of t. However, this construction is necessarily neither free-choice nor conflict-free, due to the competition between t and the node that consumes the control-flow tokens on e.

7 Conclusion

We have presented a formalism for annotated process models, including a notion of clausal compliance constraints. We have devised low-order polynomial time methods for checking compliance in this framework. The checks are partly exact, partly approximate guaranteeing only either of soundness or completeness.

Of course, this is only a first step in exploring this form of compliance checking. First, there are several open questions within our current formalism. In particular: (1) Is it computationally hard to check contradicted clauses in executable basic processes? (2) Is it possible to efficiently check compliance in the presence of effect conflicts? (3) How can we design compliance checking methods for hard cases? As for (1), it is entirely unclear to us, at this point, what the answer is. We would guess that the problem is hard, but our attempts to prove this have been unsuccesful, so far. As for (2), we have drafted an I-propagation algorithm that should work in the presence of effect conflicts, but we have not yet verified whether the algorithm is actually correct. If it is, the compliance checks should generalize effortlessly, similarly as for structured loops. Regarding (3), we have made a few first experiments encoding executability checking for the Model Checking tool SPIN [21]. The results are not encouraging so far, but there certainly is room for improvement, using search enhancements and/or alternative encoding methods.

Apart from this kind of issues, the formalism lacks expressiveness in several respects. On the process modeling side, things that cannot be adequately modeled are, e.g., data content, or temporal aspects of the behavior of activities. For data content, all we can currently do is to annotate predicates representing qualitative properties of the data, e.g., whether or not a set of numbers is sorted, or whether a number is greater than 0. Regarding temporal behavior, our model is limited to what is encoded in the control-flow; for example, quantitative messures of how long an activity takes can not be expressed.

There clearly is also a lack of expressiveness in the model of compliance rules. Clausal constraints are a rather blunt way to state regulations, which (just for example) do not cater for preferences (“if you can’t do A, then at least do B”). Richer notions of compliance rules exist, e.g., the FCL [20] formalism mentioned already in Section 6. To cater for such compliance notions, beside an extension to deal with preferences between obligations, our formalism must be extended with, e.g., ways of expressing resource allocations and temporal aspects.

At the time of writing, beyond our aforementioned initial work on FCL [18], it is not clear to us how any of the mentioned extensions should best be done. For certain, such extensions are not trivial. Resource allocations may to some extent be expressible in terms of semantic annotations. To deal with data content, a careful extension to allow (some) arithmetic could be quite useful. Regarding temporality, a lot of added value might lie in the simple extension that annotates each activity with a constant execution time; a fruitful direction for such a setting may be to extend I-propagation with time windows expressing *when* the literals will be necessarily true. As best we can tell, more complex notions of temporality will add a whole new level of complexity to both the formalism and the algorithms required for dealing with it.

Acknowledgments

This work has in part been funded through NICTA and through the SUPER project. SUPER (FP6- 026850, http://www.ip-super.org) is funded through the European Union’s 6th

Framework Programme, within Information Society Technologies (IST) priority. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. A. Ankolekar et al: DAML-S: Web service description for the semantic web. In: ISWC (2002)
2. Aalst, W.: Formalization and Verification of Event-driven Process Chains. Information and Software Technology **41**(10), 639–650 (1999)
3. Aalst, W.: Interorganizational Workflows: An Approach based on Message Sequence Charts and Petri Nets. Systems Analysis - Modelling - Simulation **34**(3), 335–367 (1999)
4. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: An approach based on temporal logic. In: Meersman, R., Tari, Z., Hacid, M.S., Mylopoulos, J., Pernici, B., Babaoglu, Ö., Jacobsen, H.A., Loyall, J.P., Kifer, M., Spaccapietra, S. (eds.) OTM Conferences (1), *Lecture Notes in Computer Science*, vol. 3760, pp. 130–147. Springer (2005)
5. van der Aalst, W.M.P., van Hee, K.: Workflow Management: Models, Methods, and Systems (Cooperative Information Systems). The MIT Press (2002). URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{%&}path=ASIN/0262011891`
6. Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters **8**, 121–123 (1979)
7. Awad, A., Decker, G., Weske, M.: Efficient compliance checking using bpmn-q and temporal logic. In: Dumas, M., Reichert, M., Shan, M.C. (eds.) Business Process Management, 6th International Conference, BPM 2008, *Lecture Notes in Computer Science*, vol. 5240, pp. 326–341. Springer (2008)
8. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: First results. In: AAAI (2005)
9. Berthelot, G.: Transformations and Decompositions of Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties, *LNCS*, vol. 254, pp. 360–376. Springer-Verlag (1987)
10. Chopra, A.K., Sing, M.P.: Producing compliant interactions: Conformance, coverage and interoperability. In: Baldoni, M., Endriss, U. (eds.) Declarative Agent Languages and Technologies IV, *LNAI*, vol. 4327, pp. 1–15. Springer-Verlag, Berlin Heidelberg (2007)
11. Coalition, T.O.S.: OWL-S: Semantic Markup for Web Services (2003)
12. D. Fensel et al: Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer-Verlag (2006)
13. Desel, J., Esparza, J.: Free choice Petri nets. Cambridge University Press, New York, NY, USA (1995)
14. Farrell, A., Sergot, M., Sallé, M., Bartolini, C.: Using the event calculus for tracking the normative state of contracts. International Journal of Cooperative Information Systems **14**(2-3), 99–129 (2005)
15. Garcia-Valles, F., Colom, J.: Implicit places in net systems. In: Petri Nets and Performance Models, 1999. Proceedings. The 8th International Workshop on, pp. 104–113 (1999)
16. Ghose, A., Koliadis, G.: Auditing business process compliance. In: Service Oriented Computing, ISOC 2007, LNCS, pp. 169–180. Springer (2007)
17. Giacomo, G.D., Lenzerini, M., Poggi, A., Rosati, R.: On the update of description logic ontologies at the instance level. In: AAAI (2006)
18. Governatori, G., Hoffmann, J., Sadiq, S., Weber, I.: Detecting regulatory compliance for business process models through semantic annotations. In: BPD-08: 4th International Workshop on Business Process Design (2008)
19. Governatori, G., Milosevic, Z.: A formal analysis of a business contract language. International Journal of Cooperative Information Systems **15**(4), 659–685 (2006)
20. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance checking between business processes and business contracts. In: Hung, P.C.K. (ed.) 10th International Enterprise Distributed Object Computing Conference (EDOC 2006), pp. 221–232. IEEE Computing Society (2006). DOI 10.1109/EDOC.2006.22
21. Holzmann, G.: The Spin Model Checker - Primer and Reference Manual. Addison-Wesley (2003)
22. Howell, R., Rosier, L.: Problems concerning fairness and temporal logic for conflict-free petri nets. Theoretical Computer Science **64**(3), 305–329 (1989)
23. Liu, Y., Müller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Systems Journal **46**(2), 335–362 (2007)
24. Lutz, C., Sattler, U.: A proposal for describing services with DLs. In: DL (2002)

25. Ly, L.T., Rinderle, S., Dadam, P.: Semantic correctness in adaptive process management systems. In: BPM06: Proc. 4th Int'l Conf. on Business Process Management, pp. 193–208. Vienna, Austria (2006)
26. Ly, L.T., Rinderle, S., Dadam, P.: Integration and verification of semantic constraints in adaptive process management systems. Data Knowl. Eng. **64**(1), 3–23 (2008)
27. OASIS: Web Services Business Process Execution Language Version 2.0 (2007). URL `http://www.ibm.com/developerworks/webservices/library/ws-bpel/`
28. OMG: Business Process Modeling Notation – BPMN 1.1. OMG Specification, `http://www.bpmn.org/` (2008)
29. Roman, D., Kifer, M.: Reasoning about the behaviour of semantic web services with concurrent transaction logic. In: VLDB, pp. 627–638 (2007)
30. Sadiq, S., Governatori, G., Namiri, K.: Modelling control objectives for business process compliance. In: Proc. 5th International Conference on Business Process Management. Brisbane, Australia (2007)
31. Sergot, M.J., Sadri, F., Kowalski, R.A., Kriwaczek, F., Hammond, P., Cory, H.: The british nationality act as a logic program. Commun. ACM **29**(5), 370–386 (1986)
32. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models though SESE Decomposition. In: Krämer, B., Lin, K., Narasimhan, P. (eds.) 5th International Conference on Service-Oriented Computing (ICSOC), *Lecture Notes in Computer Science*, vol. 4749, pp. 43–55. Springer-Verlag Berlin Heidelberg (2007)
33. Weber, I., Hoffmann, J., Mendling, J.: Semantic business process validation. In: Proceedings of the 3rd International Workshop on Semantic Business Process Management (SBPM'08) (2008)
34. Winslett, M.: Reasoning about actions using a possible models approach. In: AAAI (1988)
35. zur Muehlen, M., Indulska, M., Kemp, G.: Business process and business rule modeling languages for compliance management: A representational analysis. In: Proc. 26th International Conference on Conceptual Modelling - ER2007 - Tutorials, Posters, Panels and Industrial Contributions. Auckland, New Zealand (2007)