Behavioral Reasoning on Semantic Business Processes in a Rule-Based Framework*

Fabrizio Smith and Maurizio Proietti

National Research Council, IASI "Antonio Ruberti" - Viale Manzoni 30, 00185 Roma, Italy
{fabrizio.smith, maurizio.proietti}@iasi.cnr.it

Abstract. We propose a representation method for semantically enriched business processes by combining in a uniform logical framework both the procedural and the domain dependent knowledge. First, we define a rule-based procedural semantics for a relevant fragment of BPMN, a very popular graphical notation for specifying business processes. Our semantics defines a state transition system by following an approach similar to the Fluent Calculus, and allows us to specify state change in terms of preconditions and effects of the enactment of activities. Then, we show how the procedural process knowledge can be seamlessly integrated with the domain knowledge specified by using the OWL-RL rule-based ontology language. As a result, our framework provides a wide range of reasoning services by using standard logic programming inference engines.

Keywords: Business Processes, Ontologies, Rule-Based Reasoning, Verification

1 Introduction

The adoption of structured and systematic approaches for the management of the Business Processes (BPs) operating within an organization is constantly gaining popularity in various industrial sectors, especially in medium to large enterprises, and in the public administration. The core of such approaches is the development of BP models that represent the knowledge about processes in machine accessible form. However, standard BP modeling languages are not fully adequate to capture process knowledge in all its aspects. While their focus is on the procedural representation of a BP as a workflow graph that specifies the planned order of operations, the domain knowledge regarding the entities involved in such a process, i.e., the business environment in which processes are carried out, is often left implicit. This kind of knowledge is typically expressed through natural language comments and labels attached to the models, which constitute very limited, informal and ambiguous pieces of information.

The above issues are widely recognized as an obstacle for the further automation of BP Management (BPM) tools and methodologies [8]. Process modeling, in particular, is still mainly a manual activity, where a very limited support is given in terms of reuse and retrieval functionalities, or automated analysis facilities, i.e., for verifying whether the

* This work has been partly funded by the European Commission through the ICT Project BIVEE: Business Innovation and Virtual Enterprise Environment (FoF-ICT-2011.7.3-285746).

requirements specified over the models are enforced. The latter aspect is addressed in the BPM community mainly from a control flow perspective, with the aim of verifying whether the behavior of the modeled system presents logical errors (see, for instance, the notion of soundness [24]).

However, in order to verify that a BP actually behaves as expected, additional domain knowledge is required. In this respect, the application of well-established techniques stemming from the area of Knowledge Representation in the domains of BP modeling [8, 11, 26] and Web Services [2, 6] has been shown as a promising approach. In particular, the use of computational ontologies is the most established approach for representing in a machine processable way the knowledge about the domain where business processes operate, providing formal definitions for the basic entities involved in a process, such as activities, actors, data items, and the relations between them. However, there are still several open issues regarding the combination of BP modeling languages (with their execution semantics) and ontologies, and the accomplishment of behavioral reasoning tasks involving both these components.

The main objective of this paper is to design a framework for representing and reasoning about business process knowledge from both the procedural and ontological point of views. To achieve this goal, we do *not* propose yet another business process modeling language, but we provide a rule-based framework for reasoning about process-related knowledge expressed by using de-facto standards for BP modeling, like BPMN [17], and ontology definition, like OWL [9]. To this end we define a rule-based procedural semantics for a relevant fragment of BPMN, and we extend it in order to take into account OWL annotations that describe preconditions and effects of activities and events occurring within a BP. Our procedural BP semantics seamlessly integrates with OWL-RL [9], a fragment of the OWL ontology language which has a suitable rule-based presentation, and is achieving increasing success because it constitutes an excellent compromise between expressivity and efficiency.

In essence, the contributions of this paper can be summarized as follows. In Section 2 we introduce a set of rules, expressed in the logic programming formalism [13], for modeling the procedural semantics of a BP regarded as a workflow. The proposed rule set can cope with a relevant fragment of the BPMN 2.0 specification, allowing us to deal with a large class of process models. We then propose in Section 3 an approach for the semantic annotation of BP models, where preconditions and effects of BP elements are described by using an OWL-RL ontology. In Section 4 we provide a general verification mechanism by encoding the temporal logic CTL [4] as a set of rules which allow us to analyze properties of BPs depending on both the control flow and semantic annotations. Finally, in Section 5 we show how we can perform some very sophisticated reasoning tasks, such as verification, querying and trace compliance checking, that combine both the procedural and domain knowledge relative to a BP.

2 Behavioral Semantics of BP Schemas

In this section we introduce a formal representation of business processes by means of the notion of *Business Process Schema* (BPS). A BPS, its meta-model, and its procedural (or *behavioral*) semantics will all be specified by sets of rules, for which we adopt the standard notation and semantics of logic programming (see, for instance, [13]). In

particular, a rule is of the form $A \leftarrow L_1 \wedge \ldots \wedge L_n$, where A is an *atom* (i.e., a formula of the form $p(t_1,\ldots,t_m)$) and $L_1,\ldots,L_n$ are *literals* (i.e., atoms or negated atoms). If $n = 0$ we call the rule a *fact*. A rule (atom, literal) is *ground* if no variables occur in it. A *logic program* is a finite set of rules. Throughout the paper we will consider the class of (*locally*) *stratified* logic programs, i.e., programs that can be layered into strata such that negated atoms in higher strata are defined by rules in lower strata. Every program P in this class has a unique *perfect model*, denoted $Perf(P)$, constructed as shown in [18].

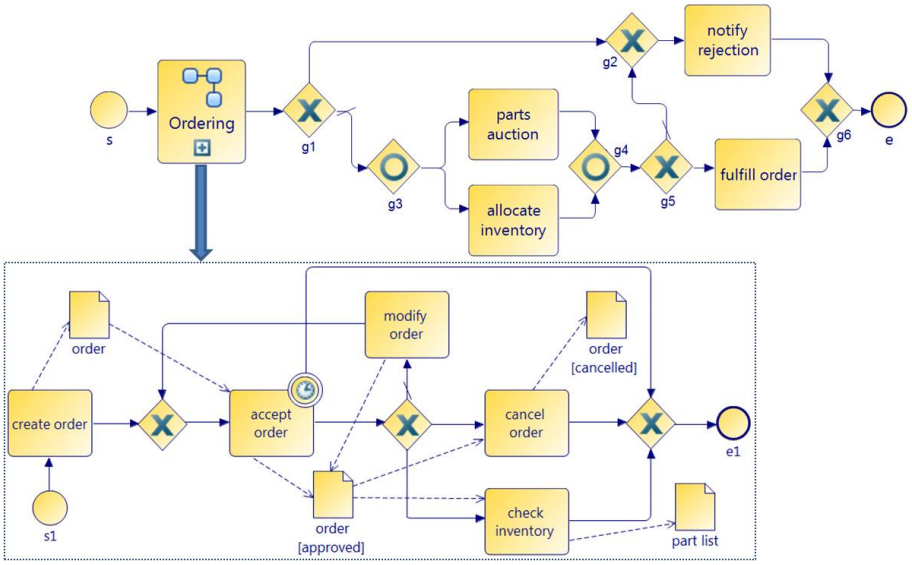

Fig. 1: Handle Order BP

2.1 Business Process Schemas

We show how a BPS is specified by means of an example. The full definition can be found in [21]. Let us consider the BP depicted in Figure 1, where the handling of a purchase order is represented using the BPMN notation. The process starts with the *ordering* activity, which is a *compound* activity where, upon receiving a customer request, a purchase order is created (*create order*), approved (*accept order*) or canceled (*cancel order*). An approved order can also be subjected to a number of modifications (*modify order*). If the order is canceled, the rejection is notified to the customer and the order is archived (*notify rejection*). Otherwise, after the requisition of the requested items (*parts auction* and *allocate inventory*), the delivery of products takes place together with the payment of the order (*fulfill order*).

A BPS (e.g., *Handle Order*) consists of a set of *flow elements* and *relations* between them, and it is associated with a unique *start event* and a unique *end event*, which are flow elements that represent the entry point and the exit point, respectively, of the process. An *activity* is a flow element that represents a unit of work performed within the process. A *task* represents an atomic activity (e.g., *accept order*), i.e., no further decomposable, while a *compound activity* is associated with a process that provides

the definition of its internal structure (e.g., *ordering*). An *intermediate event* represents "something that occurs during the process execution" (e.g., the *time-out exception* attached to the *accept order* activity). The sequencing of flow elements is specified by the *sequence flow* relation (corresponding to solid arrows), and the *branching/merging* of the control flow is specified by using three types of *gateways*: *exclusive* (XOR, e.g., *g1*), *inclusive* (OR, e.g., *g3*), and *parallel* (AND, not exemplified in Figure 1). The *item flow* relation (corresponding to dotted arrows) specifies that a flow element uses as *input* (e.g., *accept order* and *order*) or produces as *output* (e.g., *create order* and *order*) a particular *item*, i.e., a physical or information object.

A BPS can also represent other entities usually employed to model processes, such as *participants* and *messages*, not presented here for lack of space. Indeed, by following our approach we can represent the constructs common to the most used BP modeling languages and, in particular, the ones based on the BPMN specification [17].

Formally, a BPS is specified by a set of ground facts of the form $p(c_1,\ldots,c_n)$, where $c_1,\ldots,c_n$ are constants denoting flow elements (e.g., activities, events, and gateways) and p is a predicate symbol. An excerpt of the translation of the Handle Order process (referred to as *ho*) as a BPS is shown in Table 1.

Table 1: BPS representing the Handle Order process

bp(ho,s,e)	*seq(notify_rejection,g6,ho)*	*seq(parts_auction,g4,ho)*
seq(ordering,g1,ho)	*exc_branch(g1)*	*seq(g4,g5,ho)*
seq(g1,g3,ho)	*inc_branch(g3)*	*seq(g5,fulfill_order,ho)*
seq(g3,allocate_inventory,ho)	*comp_act(ordering, s_1, e_1)*	*seq(fulfill_order,g6,ho)*
seq(allocate_inventory,g4,ho)	*seq(s, ordering, ho)*	*seq(g6,e,ho)*
seq(g5,g2,ho)	*seq(g1,g2, ho)*	*exc_merge(g2)*
seq(g2,notify_rejection,ho)	*seq(g3,parts_auction,ho)*	...

Our formalization also includes a set of rules that represent the *meta-model*, defining a number of structural properties which regard a BPS as a directed graph, where edges correspond to sequence and item flow relations. Two categories of structural properties should be verified by a *well-formed* (i.e., syntactically correct) BPS: *i) local* properties related to its elementary components (e.g., every activity must have at most one ingoing and at most one outgoing sequence flow), and *ii) global* properties related to the overall structure of the BPS (e.g., every flow element must lie on a path from the *start* to the *end* event). Furthermore, other meta-model properties are related to the notions of path and reachability between flow elements, such as the following ones, which will be used in the sequel: $seq^+(E_1,E_2,P)$, representing the transitive closure of the *sequence flow* relation, and $n\_reachable(E_1,E_2,E_3,P)$, which holds if there is a path in P between E_1 and E_2 not including E_3.

2.2 Behavioral Semantics

Now we present a formal definition of the behavioral semantics, or *enactment*, of a BPS, by following an approach inspired to the *Fluent Calculus*, a well-known calculus for action and change (see [23] for an introduction). In the Fluent Calculus, the state of the world is represented as a collection of *fluents*, i.e., terms representing atomic properties that hold at a given instant of time.

An action, also represented as a term, may cause a change of state, i.e., an update of the collection of fluents associated with it. Finally, a *plan* is a sequence of actions that leads from the initial to the final state.

For states we use set notation (here we depart from [23], where an associative-commutative operator is used for representing collections of fluents). A fluent is an expression of the form $f(a_1,\ldots,a_n)$, where f is a fluent symbol and $a_1,\ldots,a_n$ are constants or variables. In order to model the behavior of a BPS, we represent states as *finite sets* of ground fluents. We take a closed-world interpretation of states, that is, we assume that a fluent F holds in a state S iff $F \in S$. This set-based representation of states relies on the assumption that the BPS is *safe*, i.e., during its enactment there are no concurrent executions of the same flow element [24]. This assumption enforces that the set of states reachable by a given BPS is finite.

A *fluent expression* is built inductively from fluents, the binary function symbol *and*, and the unary function symbol *not*. The satisfaction relation assigns a truth value to a fluent expression with respect to a state. This relation is encoded by a predicate $holds(F,S)$, which holds if the fluent expression F is true in the state S. We also introduce a constant symbol *true*, such that $holds(true,S)$ holds for every state S. Accordingly to the closed-world interpretation given to states, the satisfaction relation is defined by the following rules:

$holds(F,S) \leftarrow F = true$
$holds(F,S) \leftarrow F \in S$
$holds(not(F),S) \leftarrow \neg holds(F,S)$
$holds(and(F_1,F_2),S) \leftarrow holds(F_1,S) \wedge holds(F_2,S)$

We will consider the following two kinds of fluents: $cf(E_1,E_2,P)$, which means that the flow element E_1 has been executed and the flow element E_2 is waiting for execution, during the enactment of the process P (*cf* stands for *control flow*); $en(A,P)$, which means that the activity A is being executed during the enactment of the process P (*en* stands for *enacting*). To clarify our terminology note that, when a flow element E_2 is waiting for execution, E_2 might not be enabled to execute, because other conditions need to be fulfilled, such as those depending on the synchronization with other flow elements (see, in particular, the semantics of merging behaviors below).

We assume that the execution of an activity has a beginning and a completion (although we do not associate a *duration* with activity execution), while the other flow elements execute instantaneously. Thus, we will consider two kinds of actions: $begin(A)$ which starts the execution of an activity A, and $complete(E)$, which represents the completion of the execution of a flow element E (possibly, an activity). The change of state determined by the execution of an action will be formalized by a relation $result(S_1,A,S_2)$, which holds if the action A can be executed in the state S_1 leading to the state S_2. For defining the relation $result(S_1,A,S_2)$ the following auxiliary predicates will be used: (i) $update(S_1,T,U,S_2)$, which holds if $S_2 = (S_1 - T) \cup U$, where S_1,T,U, and S_2 are sets of fluents, and (ii) $setof(F,C,S)$, which holds if S is the set of ground instances of fluent F such that condition C holds.

The relation $r(S_1,S_2)$ holds if a state S_2 is *immediately reachable* from a state S_1, that is, some action A can be executed in state S_1 leading to state S_2:

$r(S_1,S_2) \leftarrow result(S_1,A,S_2)$

We say that a state S_2 is *reachable* from a state S_1 if there is a finite sequence of actions (of length ≥ 0) from S_1 to S_2, that is, $reachable\_state(S_1, S_2)$ holds, where the relation *reachable_state* is is the reflexive-transitive closure of r.

In the rest of this section we present a fluent-based formalization of the behavioral semantics of a BPS by focusing on a core of the BPMN language. The proposed formal semantics, reported in Table **??**, mainly refers to the BPMN semantics, as described (informally) in the most recent specification of the language [17]. Most of the constructs considered here (e.g., parallel or exclusive branching/merging) have the same interpretation in most workflow languages. However, when different interpretations are given, e.g., in the case of inclusive merge, we stick to the BPMN one.

Activity and Event Execution. The enactment of a process P begins with the execution of the associated start event E in a state where the fluent $cf(start, E, P)$ holds, being *start* a reserved constant. After the execution of the start event, its unique successor waits for execution (rule E1). The execution of an end event leads to the final state of a process execution, in which the fluent $cf(E, end, P)$ holds, where E is the end event associated with the process P and *end* is a reserved constant (rule E2).

The execution of an activity is enabled to begin after the completion of its unique predecessor flow element. The effects of the execution of an activity vary depending on its type (i.e., atomic task or compound activity). The beginning of an atomic task A is modeled by adding the $en(A, P)$ fluent to the state (rule A1). At the completion of A, the $en(A, P)$ fluent is removed and the control flow moves on to the unique successor of A (rule A2). The execution of a compound activity, whose internal structure is defined as a process itself, begins by enabling the execution of the associated *start event* (rule A3), and completes after the execution of the associated *end event* (rule A4).

According to the informal semantics of BPMN, *intermediate events* are intended as instantaneous patterns of behavior that are registered at a given time point. Thus, we formally model the execution of an intermediate event as a single state transition, as defined in rule E3. Intermediate events in BPMN can also be attached to activity boundaries to model exceptional flows. Upon occurrence of an *exception*, the execution of the activity is interrupted, and the control flow moves along the sequence flow that leaves the event (rule E4).

Branching Behaviors. When a branch gateway is executed, a subset of its successors is selected for execution. We consider here exclusive, inclusive, and parallel branch gateways. An exclusive branch leads to the execution of exactly one successor, while an inclusive branch leads to the concurrent execution of a non-empty subset of its successors. The set of successors of exclusive or inclusive decision points is selected by using *guards*, i.e., fluent expressions whose truth value is tested with respect to the current state. The value of guards may depend on fluents different from $cf(E_1, E_2, P)$ and $en(A, P)$. Indeed, extra fluents can be introduced for modeling the effects of the execution of flow elements (e.g., operations on items) as shown in Section 3.2. A guard is associated with a gateway by the predicate $c\_seq(G, B, Y, P)$ modeling a *conditional sequence flow*, where G is a fluent expression denoting a guard, B is an exclusive or inclusive branch gateway and Y is a successor flow element of B in the process P. We also have the rule $seq(B, Y, P) \leftarrow c\_seq(G, B, Y, P)$. The semantics of inclusive branches is defined in rule B1. The semantics of exclusive branches can be defined in a similar

way and is omitted. Finally, a parallel branch leads to the concurrent execution of all its successors (rule B2).

Table 2: Fragment of the behavioral semantics of the BPAL language

(E1) $result(S_1, complete(E), S_2) \leftarrow start\_event(E) \wedge holds(cf(start,E,P), S_1) \wedge seq(E,X,P) \wedge update(S_1, \{cf(start,E,P)\}, \{cf(E,X,P)\}, S_2)$

(E2) $result(S_1, complete(E), S_2) \leftarrow end\_event(E) \wedge holds(cf(X,E,P), S_1) \wedge update(S_1, \{cf(X,E,P)\}, \{cf(E,end,P)\}, S_2)$

(E3) $result(S_1, complete(E), S_2) \leftarrow int\_event(E) \wedge holds(cf(X,E,P), S_1) \wedge seq(E,Y,P) \wedge update(S_1, \{cf(X,E,P)\}, \{cf(E,Y,P)\}, S_2)$

(E4) $result(S_1, complete(E), S_2) \leftarrow exception(E,A,P) \wedge int\_event(E) \wedge holds(en(A,P), S_1) \wedge seq(E,Y,P) \wedge update(S_1, \{en(A,P)\}, \{cf(E,Y,P)\}, S_2)$

(A1) $result(S_1, begin(A), S_2) \leftarrow task(A) \wedge holds(cf(X,A,P), S_1) \wedge update(S_1, \{cf(X,A,P)\}, \{en(A,P)\}, S_2)$

(A2) $result(S_1, complete(A), S_2) \leftarrow task(A) \wedge holds(en(A,P), S_1) \wedge seq(A,Y,P) \wedge update(S_1, \{en(A,P)\}, \{cf(A,Y,P)\}, S_2)$

(A3) $result(S_1, begin(A), S_2) \leftarrow comp\_act(A,S,E) \wedge holds(and(cf(X,A,P), not(en(A,P))), S_1) \wedge update(S_1, \{cf(X,A,P)\}, \{cf(start,S,A), en(A,P)\}, S_2)$

(A4) $result(S_1, complete(A), S_2) \leftarrow comp\_act(A,S,E) \wedge holds(and(cf(E,end,A), en(A,P)), S_1) \wedge seq(A,Y,P) \wedge update(S_1, \{en(A,P), cf(E,end,A)\}, \{cf(A,Y,P)\}, S_2)$

(B1) $result(S_1, complete(B), S_2) \leftarrow inc\_branch(B) \wedge holds(cf(X,B,P), S_1) \wedge setof(cf(B,Y,P), (c\_seq(G,B,Y,P) \wedge holds(G, S_1)), Succ) \wedge update(I, \{cf(X,B,P)\}, Succ, S_2)$

(B2) $result(S_1, complete(B), S_2) \leftarrow par\_branch(B) \wedge holds(cf(X,B,P), S_1) \wedge setof(cf(B,Y,P), seq(B,Y,P), Succ) \wedge update(S_1, \{cf(X,B,P)\}, Succ, S_2)$

(O1) $result(S_1, complete(M), S_2) \leftarrow inc\_merge(M) \wedge enabled\_im(M, S_1, P) \wedge seq(M,Y,P) \wedge setof(cf(X,M,P), holds(cf(X,M,P), S_1), PredM) \wedge update(S_1, PredM, \{cf(M,Y,P)\}, S_2)$

(O2) $enabled\_im(M, S_1, P) \leftarrow holds(cf(X,M,P), S_1) \wedge \neg exists\_upstream(M, S_1, P)$

(O3) $exists\_upstream(M, S_1, P) \leftarrow seq(X,M,P) \wedge holds(not(cf(X,M,P)), S_1) \wedge holds(cf(Y,U,P), S_1) \wedge upstream(U,X,M,S_1,P)$

(O4) $upstream(U,X,M,S_1,P) \leftarrow n\_reachable(U,X,M,P) \wedge \neg exists\_path(U,M,S_1,P)$

(O5) $exists\_path(U,M,S_1,P) \leftarrow holds(cf(K,M,P), S_1) \wedge n\_reachable(U,K,M,P)$

(P1) $result(S_1, complete(M), S_2) \leftarrow par\_merge(M) \wedge \neg exists\_non\_executed\_pred(M,P,S_1) \wedge seq(M,Y,P) \wedge setof(cf(X,M,P), seq(X,M,P), PredM) \wedge update(S_1, PredM, \{cf(M,Y,P)\}, S_2)$

(P2) $exists\_non\_executed\_pred(M,P,S_1) \leftarrow seq(X,M,P) \wedge holds(not(cf(X,M,P)), S_1)$

Merging Behaviors. An exclusive merge can be executed whenever at least one of its predecessors has been executed. Here we omit the straightforward formal definition.

For the inclusive merge several operational semantics have been proposed, due to the complexity of its non-local semantics, see e.g., [10]. An inclusive merge is supposed to be able to synchronize a varying number of threads, i.e., it is executed only when $n(\geq 1)$ predecessors have been executed and no other will be eventually executed. Here we refer to the semantics described in [25] adopted by BPMN, stating that (rule O1) an inclusive merge M can be executed if the following two conditions hold (rules O2, O3):

(1) at least one of its predecessors has been executed,

(2) for each non-executed predecessor X, there is no flow element U which is waiting for execution and is *upstream* X. The notion of being upstream captures the fact that U may lead to the execution of X, and is defined as follows. A flow element U is upstream X if (rules O4, O5): *a)* there is a path from U to X not including M, and *b)* there is no path from U to an executed predecessor of M not including M.

Finally, a parallel merge can be executed if all its predecessors have been executed as defined in rule P1, where $exists\_non\_executed\_pred(M,P,S_1)$ holds if there exists no predecessor of M which has not been executed in state S_1 (rule P2).

3 Semantic Annotation

In the previous section we have shown how the behavioral semantics of the workflow specified by a BPS can be modeled in our rule-based framework. However, not all the relevant knowledge regarding process enactment is captured by a workflow model, which defines the planned order of operations but does not provide an explicit representation of the domain knowledge regarding the entities involved in such a process, i.e., the business environment in which processes are carried out.

Similarly to proposals like *Semantic BPM* [8] and *Semantic Web Services* [6], we will make use of *semantic annotations* to enrich the procedural knowledge specified by a BPS with domain knowledge expressed in terms of a given business reference ontology. Annotations provide two kinds of ontology-based information: *(i)* formal definitions of the basic entities involved in a process (e.g., activities, actors, items) to specify their meaning in an unambiguous way (*terminological* annotations), and *(ii)* specifications of preconditions and effects of the enactment of flow elements (*functional* annotations). In this work we focus on functional annotations and on their interaction with the control flow to define the behavior of a BPS, thus extending the framework presented in [21] where terminological annotations only were considered.

3.1 Rule-Based Ontologies

A business reference ontology is intended to capture the semantics of a business scenario in terms of the relevant vocabulary plus a set of axioms (TBox) which define the intended meaning of the vocabulary terms. In order to represent the semantic annotations and the behavioral semantics of a BPS in a uniform way, we will represent ontologies by sets of rules. To this end, we consider a fragment of OWL falling within the OWL 2 RL [9] profile, which is an upward-compatible extension of RDF and RDFS whose semantics is defined via a set of Horn rules, called OWL 2 RL/RDF rules. OWL 2 RL ontologies are modeled by means of the ternary predicate $t(s,p,o)$ representing an OWL statement with subject s, predicate p and object o. For instance, the assertion $t(a, rdfs{:}subClassOf, b)$ represents the inclusion axiom $a \sqsubseteq b$. Reasoning on triples is supported by OWL 2 RL/RDF rules of the form $t(s,p,o) \leftarrow t(s_1,p_1,o_1) \wedge \cdots \wedge t(s_n,p_n,o_n)$. For instance, the rule $t(A, rdfs{:}subClassOf, B) \leftarrow t(A, rdfs{:}subClassOf, C) \wedge t(C, rdfs{:}subClassOf, B)$ defines the transitive closure of the subsumption relation.

An OWL 2 RL ontology is represented as a set O of rules, consisting of a set of facts of the form $t(s,p,o)$, called *triples*, encoding the OWL TBox and the set of Horn

rules encoding the OWL 2 RL/RDF rules. This kind of representation allows us to take advantage of the efficient resolution strategies developed for logic programs, in order to perform the reasoning tasks typically supported by Description Logics reasoning systems, such as concept subsumption and ontology consistency.

Table 3: Business Reference Ontology excerpt

$ClosedPO \sqsubseteq Order$	$ApprovedPO \sqsubseteq Order$
$CancelledPO \sqsubseteq ClosedPO$	$FulfilledPO \sqsubseteq ClosedPO$
$UnavailablePL \sqsubseteq PartList$	$AvailablePL \sqsubseteq PartList$
$payment \sqsubseteq related$	$\exists\, payment^{-} \sqsubseteq Invoice$
$CancelledPO \sqcap ApprovedPO \sqsubseteq \bot$	$UnavailablePL \sqcap AvailablePL \sqsubseteq \bot$
$ApprovedPO \sqcap \exists related.Invoice \sqsubseteq FulfilledPO$	$Order \sqcap \exists related.UnavailablePL \sqsubseteq CancelledPO$

3.2 Functional Annotation

By using the ontology vocabulary and axioms, we define semantic annotations for modeling the behavior of individual process elements in terms of *preconditions* under which a flow element can be executed and *effects* on the state of the world after its execution. Preconditions and effects, collectively called *functional annotations*, can be used, for instance, to model input/output relations of activities with data items, which are the standard way of representing information storage in BPMN diagrams. Fluents can represent the *status* of a data item affected by the execution of an activity at a given time during the execution of the process. A precondition specifies the status a data item must posses when an activity is enabled to start, and an effect specifies the status of a data item after having completed an activity. In order to provide concrete examples to illustrate the main ideas, in the rest of the paper we refer to the excerpt of reference ontology reported in Table 3, describing the items involved in the BPS depicted in Figure 1.

Functional annotations are formulated by means of the following two relations:

- $pre(A, C, P)$, which specifies the fluent expression C, called *enabling condition*, which must hold to execute an element A in the process P;
- $eff(A, E^-, E^+, P)$, which specifies the set E^- of fluents, called *negative effects*, which do not hold after the execution of A and the set of fluents E^+, called *positive effects*, which hold after the execution of A in the process P. We assume that E^- and E^+ are disjoint sets.

In the presence of functional annotations, the enactment of a BPS is modeled as follows. Given a state S_1, a flow element A can be enacted if A is waiting for execution according to the control flow semantics, and its enabling condition C is satisfied, i.e., $holds(C, S_1)$ is true. Moreover, given an annotation $eff(A, E^-, E^+, P)$, when A is completed in a given state S_1, then a new state S_2 is obtained by taking out from S_1 the set E^- of fluents and then adding the set E^+ of fluents. We will assume that effects satisfy a *consistency condition* which guarantees that: (i) no contradiction can be derived from the fluents of S_2 by using the state independent axioms of the reference ontology, and (ii) no fluent belonging to E^- holds in S_2. This consistency condition will be formally defined later in this section, and can be regarded as a way of tackling the *Ramification Problem* due to indirect effects of actions (see e.g., [23, 19]). The state update is formalized by extending the *result* relation so as to take into account the *pre* and *eff* relations. We only consider the case of task execution. The other cases are similar and will be omitted.

Table 4: Functional annotations for the Handle Order process

Flow Element	**Enabling Condition**	**Effects**
create order		$t_f(o, \textit{rdf:type}, \textit{bro:Order})$
accept order	$t_f(o, \textit{rdf:type}, \textit{bro:Order})$	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$
cancel order	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$	$\neg t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$, $t_f(o, \textit{rdf:type}, \textit{bro:CancelledPO})$
check inventory	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$	$t_f(o, \textit{bro:related}, pl), t_f(pl, \textit{rdf:type}, \textit{bro:PartList})$
check inventory	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$	
parts auction	$t_f(pl, \textit{rdf:type}, \textit{bro:PartList})$	$t_f(pl, \textit{rdf:type}, \textit{bro:AvailablePL})$
parts auction	$t_f(pl, \textit{rdf:type}, \textit{bro:PartList})$	$\neg t\,f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$, $t_f(pl, \textit{rdf:type}, \textit{bro:UnavailablePL})$
fulfill order	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$	$t_f(o, \textit{bro:payment}, i)$

Gateway	**Target**	**Guard**
g1	g3	$t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO})$
g1	g2	$\textit{not}(t_f(o, \textit{rdf:type}, \textit{bro:ApprovedPO}))$
g3	parts auction	$\textit{and}(t_f(o, \textit{related}, pl), t_f(pl, \textit{rdf:type}, \textit{bro:PartList}))$
g5	g2	$t_f(o, \textit{rdf:type}, \textit{bro:CancelledPO})$
g5	fulfill order	$\textit{not}(t_f(o, \textit{rdf:type}, \textit{bro:CancelledPO}))$

$\textit{result}(S_1, \textit{begin}(A), S_2) \leftarrow \textit{task}(A) \wedge \textit{holds}(\textit{cf}(X, A, P), S_1) \wedge \textit{pre}(A, C, P) \wedge \textit{holds}(C, S_1) \wedge$
$\textit{update}(S_1, \{\textit{cf}(X, A, P)\}, \{\textit{en}(A, P)\}, S_2)$

$\textit{result}(S_1, \textit{complete}(A), S_2) \leftarrow \textit{task}(A) \wedge \textit{holds}(\textit{en}(A, P), S_1) \wedge \textit{eff}(A, E^-, E^+, P) \wedge$
$\textit{seq}(A, Y, P) \wedge \textit{update}(S_1, \{\textit{en}(A, P)\} \cup E^-, \{\textit{cf}(A, Y, P)\} \cup E^+, S_2)$

The enabling conditions and the negative and positive effects occurring in functional annotations are fluent expressions built from fluents of the form $t_f(s, p, o)$, corresponding to the OWL statement $t(s, p, o)$, where we adopt the usual *rdf*, *rdfs*, and *owl* prefixes for names in the OWL vocabulary, and the *bro* prefix for names relative to our specific examples. We assume that the fluents appearing in functional annotations are either of the form $t_f(a, \textit{rdf:type}, c)$, corresponding to the unary atom $c(a)$, or of the form $t_f(a, p, b)$, corresponding to the binary atom $p(a, b)$, where a and c are *individuals*, while c and p are concepts and properties, respectively, defined in the reference ontology O. Thus, fluents correspond to assertions about individuals, i.e., the ABox of the ontology, and hence the ABox may change during process enactment due to the effects specified by the functional annotations, while O, providing the ontology definitions and axioms, i.e., the TBox of the ontology, does not change.

Let us now present an example of specification of functional annotations. In particular, our example shows nondeterministic effects, that is, a case where a flow element A is associated with more than one pair (E^-, E^+) of negative and positive effects.

Example 1 *Consider again the* Handle Order *process in Figure 1. After the execution of* create order*, a purchase order is issued. This order can be approved or canceled upon execution of the activities* accept order *and* cancel order*, respectively. Depending on the inventory capacity checked during the* check inventory *task, the requisition of parts performed by an external supplier is performed (*parts auction*). Once that all the order parts are available, the order can be fulfilled and an invoice is associated with the order. This behavior is specified by the functional annotations reported in Table 4.*

In order to evaluate a statement of the form $holds(t_f(s,p,o),X)$, where $t_f(s,p,o)$ is a fluent and X is a state, the definition of the *holds* predicate given previously must be extended to take into account the axioms belonging to the reference ontology O. Indeed, we want that a fluent of the form $t_f(s,p,o)$ be true in state X not only if it belongs to X, but also if it can be inferred from the fluents in X and the axioms of the ontology. For instance, let us consider the fluent $F = t_f(o, rdf{:}type, bro{:}CancelledPO)$. We can easily infer that F holds in a state which contains $\{t_f(o, rdf{:}type, bro{:}CancelledPO)\}$ (e.g., reachable after the execution of *cancel order*) by using the rule $holds(F,X) \leftarrow F \in X$. However, by taking into account the ontology excerpt given in Table 3, we also want to be able to infer that F holds in a state which contains $\{t_f(o, rdf{:}type, bro{:}Order)$, $t_f(o, bro{:}related, pl)$, $t_f(pl, rdf{:}type, bro{:}UnavailablePL)\}$ (e.g., reachable after the execution of *parts auction*).

In our framework the inference of new fluents from fluents belonging to states is performed by including extra rules derived by translating the OWL 2 RL/RDF entailment rules as follows: every triple of the form $t(s,p,o)$, where s refers to an individual, is replaced by the atom $holds(t_f(s,p,o),X)$. Below we show the rules for concept subsumption (1), role subsumption (2), domain restriction (3), transitive property (4), and concept disjointness (5) .

1. $holds(t_f(S, rdf{:}type, C), X) \leftarrow holds(t_f(S, rdf{:}type, B), X) \wedge t(B, rdfs{:}subClassOf, C)$
2. $holds(t_f(S,P,O),X) \leftarrow holds(t_f(S,P1,O),X) \wedge t(P1, rdfs{:}subPropertyOf, P)$
3. $holds(t_f(S, rdf{:}type, C), X) \leftarrow holds(t_f(S,P,O),X) \wedge t(P, rdfs{:}domain, C)$
4. $holds(t_f(S,P,O),X) \leftarrow holds(t_f(S,P,O_1),X) \wedge holds(t_f(O_1,P,O),X) \wedge$
   $t(P, rdf{:}type, owl{:}TransitiveProperty)$
5. $holds(false,X) \leftarrow holds(t_f(I_1, rdf{:}type, A),X) \wedge holds(t_f(I_2, rdf{:}type, B),X) \wedge$
   $t(A, owl{:}disjointWith, B)$

We denote by $\mathcal{F}$ the set of rules that encode the functional annotations, that is, the facts defining the relations $pre(A,C,P)$ and $\mathit{eff}(A,E^-,E^+,P)$, along with the rules for evaluating $holds(t_f(s,p,o),X)$ atoms (such as rules 1–5 above). The rules in $O \cup \mathcal{F}$ may also be needed to evaluate atoms of the form $holds(G,X)$ in the case where G is a guard expression associated with inclusive or exclusive branch points via the relation $c\_seq(G,B,Y,P)$. Indeed, G may depend on fluents introduced by functional annotations.

We are now able to define the consistency condition for effects in a rigorous way. We say that *eff* is *consistent* with process P if, for every flow element A and states S_1, S_2, the following implication is true:
If the state S_1 is reachable from the initial state of P, the relation $result(S_1, complete(A), S_2)$ holds, and the relation $\mathit{eff}(A,E^-,E^+,P)$ holds,
Then $O \cup \mathcal{F} \cup \{\neg holds(false, S_2)\}$ is consistent *And* for all $F \in E^-$, $O \cup \mathcal{F} \cup \{\neg holds(F, S_2)\}$ is consistent.

We will show in Section 5 how the consistency of effects can be checked by using the rule-based temporal logic we will present in the next section.

4 Temporal Reasoning

In order to provide a general verification mechanism for behavioral properties, in this section we propose a model checking methodology based on a formalization of the tem-

poral logic CTL (*Computation Tree Logic*, see [4] for a comprehensive overview) as a set of rules. Model checking is a widely accepted technique for the formal verification of BP schemas, as their execution semantics is usually defined in terms of states and state transitions, and hence the use of temporal logics for the specification and verification of properties is a very natural choice [7, 12]. The abstract syntax of a CTL formula F is defined as follows:

$$F ::= e \mid \mathit{true} \mid \mathit{false} \mid \neg F \mid F_1 \wedge F_2 \mid \mathbf{EX}(F) \mid \mathbf{EU}(F_1, F_2) \mid \mathbf{EG}(F)$$

where e is a fluent expression. Other operators can be defined in terms of the ones given above, e.g., $\mathbf{EF}(F) \equiv \mathbf{EU}(\mathit{true}, F)$ and $\mathbf{AG}(F) \equiv \neg\mathbf{EF}(\neg F)$ [4].

The semantics of CTL formulas is defined by taking into account the immediate reachability relation r between states (i.e., finite sets of ground fluents) introduced in Section 2.2, which here is also called the *transition relation*.

In the definition of the semantics of CTL given in [4], the transition relation r is assumed to be *total*, that is, every state S_1 has at least one *successor state* S_2 for which $r(S_1, S_2)$ holds. This assumption is justified by the fact that the reactive systems considered in [4] can be thought as ever running processes. However, this assumption is not realistic in the case of business processes, for which there is always at least one state with no successors, namely one where the *end* event of a BPS has been completed. For this reason the semantics of the temporal operators given in [4], which refers to *infinite* sequences of states, is suitably changed here by taking into consideration *maximal paths*, i.e., sequences of states that are either infinite or end with a state that has no successors, called a *sink*.

Now we give a rule-based formalization of the semantics of CTL by extending the definition of the predicate *holds*. (A similar formalization based on constraint logic programming is proposed in [16], where however the semantics refers to infinite paths.)

$\mathbf{EX}(F)$ holds in state S_0 if F holds in a successor state of S_0:

$\mathit{holds}(\mathit{ex}(F), S_0) \leftarrow r(S_0, S_1) \wedge \mathit{holds}(F, S_1)$

$\mathbf{EU}(F_1, F_2)$ holds in state S_0 if there exists a maximal path π: $S_0\, S_1 \ldots$ such that for some S_n occurring in π we have that F_2 holds in S_n and, for $j = 0, \ldots, n-1$, F_1 holds in S_j:

$\mathit{holds}(\mathit{eu}(F_1, F_2), S_0) \leftarrow \mathit{holds}(F_2, S_0)$
$\mathit{holds}(\mathit{eu}(F_1, F_2), S_0) \leftarrow \mathit{holds}(F_1, S_0) \wedge r(S_0, S_1) \wedge \mathit{holds}(\mathit{eu}(F_1, F_2), S_1)$

$\mathbf{EG}(F)$ holds in a state S_0 if there exists a maximal path π starting from S_0 such that F holds in each state of π. Since the set of states is finite, $\mathbf{EG}(F)$ holds in S_0 if there exists a finite path $S_0 \ldots S_k$ such that, for $i = 0, \ldots, k$, F holds in S_i, and either (1) $S_j = S_k$, for some $0 \leq j < k$, or (2) S_k is a sink state. Thus, the semantics of the operator **EG** is encoded by the following rules:

$\mathit{holds}(\mathit{eg}(F), S_0) \leftarrow \mathit{fpath}(F, S_0, S_0)$
$\mathit{holds}(\mathit{eg}(F), S_0) \leftarrow \mathit{holds}(F, S_0) \wedge r(S_0, S_1) \wedge \mathit{holds}(\mathit{eg}(F), S_1)$
$\mathit{holds}(\mathit{eg}(F), S_0) \leftarrow \mathit{sink}(S_0) \wedge \mathit{holds}(F, S_0)$

where: (i) the predicate $\mathit{fpath}(F, X, X)$ holds if there exists a path from X to X itself, consisting of at least one r arc, such that F holds in every state on the path and (ii) the predicate $\mathit{sink}(X)$ holds if X is a sink state.

Finally, we define a special fluent expression $\mathit{final}(P)$ the *final* state of a process P, whose semantics is given by the following rule:

$holds(final(P), Z) \leftarrow bp(P, S, E) \wedge holds(cf(E, end, P), Z)$

Note that our definition of the semantics of **EG** avoids the introduction of greatest fixed points of operators on sets of states which are required by the approach described in [4]. Indeed, the rules defining $holds(eg(F), S_0)$ are interpreted according to the usual least fixpoint semantics (i.e., the least Herbrand model [13]). Note also that in some special cases the assumption that paths are maximal, but not necessarily infinite, matters. For instance, if S_0 is a sink state, then $holds(ag(F), S_0)$ is true iff $holds(F, S_0)$ is true, since the only maximal path starting from S_0 is the one constituted by S_0 only.

5 Reasoning Services

Our rule-based framework supports several reasoning services which can combine complex knowledge about business processes from different perspectives, such as the workflow structure, the ontological description, and the behavioral semantics. In this section we will illustrate three such services: verification, querying, and trace compliance.

Let us consider the following sets of rules: (1) $\mathcal{B}$, representing a set of BP schemas and the BP meta-model defined in Section 2.1, (2) $\mathcal{T}$, defining the behavioral semantics presented in Section 2.2, (3) O, collecting the OWL triples and rules which represent the business reference ontology defined in Section 3.1, (4) $\mathcal{F}$, encoding the functional annotations defined in Section 3.2, and (5) $\mathcal{CTL}$, defining the semantics of CTL presented in Section 4. Let $\mathcal{KB}$ be the set of rules $\mathcal{B} \cup \mathcal{T} \cup O \cup \mathcal{F} \cup \mathcal{CTL}$. $\mathcal{KB}$ is called a *Business Process Knowledge Base* (BPKB). We have that $\mathcal{KB}$ is a locally stratified logic program and its semantics is unambiguously defined by its unique perfect model, denoted by $Perf(\mathcal{KB})$ [18].

Verification. In the following we present some examples of properties that can be specified and verified in our framework. A property is specified by a predicate *prop* defined by a rule C in terms of the predicates defined in $\mathcal{KB}$. The verification task is performed by checking whether or not $prop \in Perf(\mathcal{KB} \cup \{C\})$.

(1) A very relevant behavioral property of a BP p is that from any reachable state, it is possible to complete the process, i.e., reach the final state. This property, also known as *option to complete* [24], can be specified by the following rule, stating that the property *opt_com* holds if the CTL property $\mathbf{AG}(\mathbf{EF}(final(p)))$ holds in the initial state s_0 of p:

$opt\_com \leftarrow holds(ag(ef(final(p))), s_0)$

where $s_0 = \{cf(start, st\_ev, p)\}$ and *st_ev* is the start event associated with p.

(2) Temporal queries allow us to verify the consistency conditions for effects introduced in Section 3.2. In particular, given a BPS p, inconsistencies due to the violation of some integrity constraint defined in the ontology by rules of the form $false \leftarrow G$ (e.g., concept disjointness) can be verified by defining the *inconsistency* property as follows:

$inconsistency \leftarrow holds(ef(false), s_0)$

(3) Temporal queries can also be used for the verification of *compliance rules*, i.e., directives expressing internal policies and regulations aimed at specifying the way an enterprise operates. In our Handle Order example, one such compliance rule may be that every *order* is eventually *closed*. In order to verify whether this property holds or not, we can define a *noncompliance* property which holds if it is possible to reach the final

state of the process where, for some O, it can be inferred that O is an *order* which is not *closed*. In our example *noncompliance* is satisfied, and thus the compliance rule is not enforced. In particular, if the exception attached to the *accept order* task is triggered, the enactment continues with the *notify rejection* task (due to the guards associated to $g1$), and the order is never *canceled* nor *fulfilled*.

$$noncompliance \leftarrow holds(ef(and(t_f(O, rdf{:}type, bro{:}Order), \\ and(not(t_f(O, rdf{:}type, bro{:}ClosedPO)), final(p))), s_0)$$

The verification of a property *prop* is performed by evaluating the query $\leftarrow prop$ in $\mathcal{KB} \cup \{C\}$ using *SLG-resolution*, that is, resolution for general logic programs augmented with the *tabling* mechanism [3]. The following definition is needed for presenting the termination, soundness, and completeness of query evaluation.

Definition 1. Let f be a term representing a CTL formula. A subterm e of f is *grounding* if one of the following conditions hold: (i) f is an atomic fluent and e is f, (ii) f is $and(f_1, f_2)$ and e is a grounding subterm of either f_1 or f_2, (iii) f is $ex(f_1)$ and e is a grounding subterm of f_1, (iv) f is $eu(f_1, f_2)$ and e is a grounding subterm of f_2, (v) f is $eg(f_1)$ and e is a grounding subterm of f_1.

Theorem 1. Let C be a rule of the form $prop \leftarrow L_1 \wedge \ldots \wedge L_n$, where, for $i = 1, \ldots, n$, the predicate of L_i is defined in $\mathcal{KB}$. Suppose that: (i) if L_i is of the form $holds(f, S)$, all free variables of f occur in atomic fluents, and (ii) each variable X of C has its leftmost occurrence in a positive literal L_i such that either (ii.1) L_i has not predicate *holds* or (ii.2) $L_i = holds(f, S)$ and the occurrence of X is in a grounding subterm of f.

Then: (1) the evaluation of the query $\leftarrow prop$ in $\mathcal{KB} \cup \{C\}$ terminates by using SLG-resolution with left-to-right computation rule, and (2) the query succeeds iff $prop \in Perf(\mathcal{KB} \cup \{C\})$.

Hypothesis (i) guarantees that no variable ranges over an infinite domain, such as the set of all CTL formulas. Hypothesis (ii) guarantees that the query *does not flounder*, that is, non-ground negative literals are never selected during SLG resolution. The termination property (1) can be proved by showing that $\mathcal{KB} \cup \{C\}$ satisfies the *bounded-term-size property* [3]. The soundness and completeness property (2) follows from the soundness and completeness of SLG resolution with respect to the perfect model semantics [3].

Querying. The inference mechanism based on SLG-resolution can be used for computing boolean answers to ground queries, but also for computing, via unification, substitutions for variables occurring in non-ground queries. By exploiting this query answering mechanism we can easily provide, besides the verification service described in the previous section, also reasoning services for the retrieval of process fragments.

The following queries show how process fragments can be retrieved according to different criteria: q_1 computes every activity A (and the process P where it occurs) which operates on an *order* as an effect (e.g., *create order* and *cancel order*); q_2 computes every exclusive branch G occurring along a path delimited by two activities A and B which operate on *orders* (e.g., *create order*) and *invoices* (e.g., *fulfill order*), respectively; finally, q_3 is a refinement of q_2, where it is also required that the enactment of B is always preceded by the enactment of A.

$$q_1(A, P) \leftarrow eff(A, E^-, E^+, P) \wedge holds(t_f(O, rdf{:}type, bro{:}Order), E^+)$$

$q_2(A,G,B,P) \leftarrow \mathit{eff}(A,E_A^-,E_A^+,P) \wedge \mathit{seq}^+(A,G,P) \wedge$
$\mathit{holds}(t_f(O,\mathit{rdf{:}type},\mathit{bro{:}Order}),E_A^+) \wedge \mathit{exc\_branch}(G) \wedge \mathit{seq}^+(G,B,P) \wedge$
$\mathit{eff}(B,E_B^-,E_B^+,P) \wedge \mathit{holds}(t_f(I,\mathit{rdf{:}type},\mathit{bro{:}Invoice}),E_B^+)$
$q_3(A,G,B,P) \leftarrow q_2(A,G,B,P) \wedge \mathit{holds}(\mathit{not}(\mathit{eu}(\mathit{not}(\mathit{en}(A,P)),\mathit{en}(B,P))),s_0)$

Trace Compliance. The execution of a process is modeled as an *execution trace* (corresponding to a plan in the Fluent Calculus), i.e., a sequence of actions of the form $[\mathit{act}(a_1),\ldots,\mathit{act}(a_n)]$ where *act* is either *begin* or *complete*. The predicate $\mathit{trace}(S_1,T,S_2)$ defined below holds if T is a sequence of actions that lead from state S_1 to state S_2:

$\mathit{trace}(S_1,[\,],S_2) \leftarrow S_1 = S_2$
$\mathit{trace}(S_1,[A|T],S_2) \leftarrow \mathit{result}(S_1,A,U) \wedge \mathit{trace}(U,T,S_2)$

A *correct trace* T of a BPS P is a trace leading from the initial to the final state of P:

$\mathit{ctrace}(T,P) \leftarrow \mathit{trace}(s_0,T,Z) \wedge \mathit{holds}(\mathit{final}(P),Z)$

The correctness of a trace t with respect to a given BPS p can be verified by evaluating a query of the type $\leftarrow \mathit{ctrace}(t,p)$, where t is a ground list and p is a process name. The rules defining the predicate *ctrace* can also be used to *generate* the correct traces of a process p that satisfy some given property. This task is performed by evaluating a query of the type $\leftarrow \mathit{ctrace}(T,p) \wedge \mathit{cond}(T)$, where T is a free variable and $\mathit{cond}(T)$ is a property that T must enforce. For instance, we may want to generate traces where the execution of a flow element a is followed by the execution of a flow element b:

$\mathit{cond}(T) \leftarrow \mathit{concat}(T_1,T_2,T) \wedge \mathit{complete}(a) \in T_1 \wedge \mathit{complete}(b) \in T_2$

The termination of querying and trace correctness checking can be proved under assumptions similar to the ones of Theorem 1. However, stronger assumptions are needed for the termination of trace generation if we want to compute the set of *all* correct traces satisfying a given condition, as this set may be infinite in the presence of cycles.

6 Related Work

Among several mathematical formalisms proposed for defining a formal semantics of BP models, Petri nets [24] are the most used paradigm to capture the execution semantics of the control flow aspects of graph-based procedural languages. (The BPMN case is discussed in [5].) Petri net models enable a large number of analysis techniques, but they do not provide a suitable basis to represent and reason about additional domain knowledge. In our framework we are able to capture the token game semantics underlying workflow models, and we can also declaratively represent constructs, such as exception handling behavior or synchronization of active branches only (inclusive merge), which, due to their non-local semantics, are cumbersome to capture in standard Petri nets. In addition, the logical grounding of our framework makes it easy to deal with the modeling of domain knowledge and the integration of reasoning services.

Program analysis and verification techniques have been largely applied to the analysis of process behavior, e.g., [7, 12]. These works are based on the analysis of finite state models through model checking techniques [4] where temporal logics queries specify properties of process executions. However, these approaches are restricted to properties regarding the control flow only (e.g., properties of the ordering, presence, or absence of tasks in process executions), and severe limitations arise when taking into consideration ontology-related properties representing specific domain knowledge.

There is a growing body of contributions beyond pure control flow verification. In [26] the authors introduce the notion of Semantic Business Process Validation, which aims at verifying properties related to the absence of logical errors which extend the notion of workflow soundness [24]. Validation is based on an execution semantics where token passing control flow is combined with the AI notion of state change induced by domain related logical preconditions/effects. The main result is constituted by a validation algorithms which runs in polynomial time under some restrictions on the workflow structure and on the expressivity of the logic underlying the domain axiomatization, i.e., binary Horn clauses. This approach is focused on providing efficient techniques for the verification of specific properties, while the verification of arbitrary behavioral properties, such as the CTL formulae allowed in our framework, is not addressed. Moreover, our language for annotations is more expressive than binary Horn clauses.

Several works propose the extension to business process management of techniques developed in the context of the semantic web [8]. Meta-model process ontologies, e.g., [11], are derived from BP modeling languages and notations with the aim of specifying in a declarative, formal, and explicit way concepts and constraints of a particular language. Semantic Web Services approaches, such as OWL-S [2] and WSMO [6], make an essential use of ontologies in order to facilitate the automation of discovering, combining and invoking electronic services over the Web. To this end they describe services from two perspectives: from a *functional* perspective a service is described in terms of its functionality, preconditions and effects, input and output; from a process *perspective*, the service behavior is modeled as an orchestration of other services. However, in the above approaches the behavioral aspects are abstracted away, since the semantics of the provided constructs is not axiomatized within their respective languages, hampering the availability of reasoning services related to the execution of BPs.

To overcome such limitations, several solutions for the representation of service compositions propose to translate the relevant aspects of the aforementioned service ontologies into a more expressive language, such as first-order logic, and to add a set of axioms to this theory that constrains the models of the theory to all and only the intended interpretations. Among them, [22] adopts the high-level agent programming language Golog [19], [1, 15] rely on Situation Calculus variants. However, such approaches are mainly tailored to automatic service composition (i.e., finding a sequence of service invocations such that a given goal is satisfied). Thus, the support provided for process definition, in terms of workflow constructs, is very limited and they lack a clear mapping from standard modeling notations. In contrast, our framework allows a much richer procedural description of processes, directly corresponding to BPMN diagrams. Moreover, a reference ontology can be used to "enrich" process descriptions by means of OWL-RL annotations, a widespread language for ontology representation.

Other approaches based on Logic Programming which are worth to mention are [14, 20]. These approaches mainly focus on the verification and on the enactment of BPs, while we are not aware of specific extensions that deal with the semantic annotation of procedural process models with respect to domain ontologies.

Finally, with respect to our previous works [21], we have proposed several extensions: (1) we have increased the expressivity from a workflow perspective, by modeling arbitrary cycles, unstructured diagrams and exceptions; (2) we have introduced

functional annotations and we have provided a semantics for their integration with the control flow; (3) we have introduced a general verification mechanism based on CTL.

7 Conclusions

The rule-based approach for representing and reasoning about business processes presented in this paper offers several advantages. First of all, it enables the combination of the procedural and ontological perspectives in a very smooth and natural way, thus providing a uniform framework for reasoning on properties that depend on the sequence of operations that occur during process enactment and also on the domain where the process operates. Another advantage is the generality of the approach, which is open to further extensions, since other knowledge representation applications can easily be integrated, by providing a suitable translation to logic programming rules. Our approach does not introduce a new business process modeling language, but provides a framework where one can map and integrate knowledge represented by means of existing formalisms. This is very important from a pragmatic point of view, as one can express process-related knowledge by using standard modeling languages such as BPMN for business processes and OWL for ontologies, while adding extra reasoning services. Finally, since our rule-based representation can be directly mapped to a class of logic programs, we can use standard logic programming systems to perform reasoning tasks such as verification and querying.

We have implemented in the XSB logic programming system[1] the various sets of rules representing a Business Process Knowledge Base, and on top of the latter, the verification, querying, and trace compliance services. The resolution mechanism based on tabling [3] provided by XSB guarantees a sound and complete evaluation of a large class of queries. We have also integrated the aforementioned services in the tool described in [21], which implements an interface between the BPMN and OWL representations of business processes and reference ontology specifications on one hand, and our rule-based representation on the other hand, so that, as already mentioned, we can use the reasoning facilities offered by our framework as add ons to standard tools. First experiments are encouraging and show that very sophisticated reasoning tasks can be performed on business process of small-to-medium size in an acceptable amount of time and memory resources. Currently, we are investigating various program optimization techniques for improving the performance of our tool and enabling our approach to scale to large BP repositories.

References

[1] Battle, S., et al. (2005). Semantic Web Services Ontology. http://www.w3.org/Submission/SWSF-SWSO.

[2] Burstein, M., et al. (2004). OWL-S: Semantic Markup for Web Services. W3C Member Submission, http://www.w3.org/Submission/OWL-S/.

[3] Chen, W. and Warren, D. S. (1996). Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43:20–74.

[4] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

[1] The XSB Logic Programming System. Version 3.2: http://xsb.sourceforge.net

[5] Dijkman, R. M., Dumas, M., and Ouyang, C. (2008). Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.*, 50:1281–1294.

[6] Fensel, D., et al. (2006). *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer.

[7] Fu, X., Bultan, T., and Su, J. (2004). Analysis of Interacting BPEL Web Services. In *Int. Conf. on World Wide Web*, pages 621–630. ACM Press.

[8] Hepp, M., et al. (2005). Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. In *Int. Conf. on e-Business Engineering*. IEEE Computer Society.

[9] Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., and Rudolph, S. (2009). OWL 2 Web Ontology Language. W3C Recommendation, `http://www.w3.org/TR/owl2-primer/`.

[10] Kindler, E. (2006). On the Semantics of EPCs: Resolving the Vicious Circle. *Data Knowl. Eng.*, 56(1):23–40.

[11] Lin, Y. (2008). *Semantic Annotation for Process Models: Facilitating Process Knowledge Management via Semantic Interoperability*. PhD thesis, Norwegian University of Science and Technology.

[12] Liu, Y., Müller, S., and Xu, K. (2007). A Static Compliance-Checking Framework for Business Process Models. *IBM Syst. J.*, 46:335–361.

[13] Lloyd, J. W. (1987). *Foundations of logic programming*. Springer-Verlag New York, Inc.

[14] Montali, M., Pesic, M., Aalst, W. M. P. v. d., Chesani, F., Mello, P., and Storari, S. (2010). Declarative Specification and Verification of Service Choreographies. *ACM Trans. Web*, 4(1):3:1–3:62.

[15] Narayanan, S. and McIlraith, S. (2003). Analysis and Simulation of Web services. *Comp. Networks*, 42:675–693.

[16] Nilsson, U. and Lübcke, J. (2000). Constraint Logic Programming for Local and Symbolic Model-checking. In *Computational Logic*, LNAI 1861. Springer.

[17] OMG (2011). Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0.

[18] Przymusinski, T. C. (1988). On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc.

[19] Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

[20] Roman, D. and Kifer, M. (2008). Semantic Web Service Choreography: Contracting and Enactment. In *Int. Semantic Web Conference*, LNCS 5318, pages 550–566. Springer.

[21] Smith, F., Missikoff, M., and Proietti, M. (2012). Ontology-Based Querying of Composite Services. In *Business System Management and Engineering*, LNCS 7350, pp. 159–780. Springer.

[22] Sohrabi, S., Prokoshyna, N., and McIlraith, S. A. (2009). Web Service Composition via the Customization of Golog Programs with User Preferences. In *Conceptual Modeling: Foundations and Applications*, pages 319–334. Springer.

[23] Thielscher, M. (1998). Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, 2:179–192.

[24] van der Aalst, W. M. P. (1998). The Application of Petri Nets to Workflow Management. *J. Circuits, Systems, and Computers*, 8(1):21–66.

[25] Völzer, H. (2010). A New Semantics for the Inclusive Converging Gateway in Safe Processes. In *Int. Conf. on Business Process Management*, LNCS 6336, pages 294–309, Berlin, Heidelberg. Springer.

[26] Weber, I., Hoffmann, J., and Mendling, J. (2010). Beyond Soundness: On the Verification of Semantic Business Process Models. *Distrib. Parallel Dat.*, 27:271–343.