ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
"Antonio Ruberti"
CONSIGLIO NAZIONALE DELLE RICERCHE

M. Missikoff, M. Proietti, F. Smith

# LINKING ONTOLOGIES TO BUSINESS PROCESS SCHEMAS

R. 10-20, 2010

**Michele Missikoff** – Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30, I-00185 Roma, Italy. Email: missikoff@iasi.cnr.it

**Maurizio Proietti** – Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30, I-00185 Roma, Italy. Email: proietti@iasi.cnr.it

**Fabrizio Smith** – Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30, I-00185 Roma, Italy. Email: fabrizio.smith@iasi.cnr.it

This work was partially done under the IASI support

ISSN: 1128-3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.cnr.it
URL: http://www.iasi.cnr.it

**Abstract**

This report intends to lay the formal foundations of a framework for the semantic augmentation of BP schemas, in order to provide a declarative representation of workflow models, enriched with domain knowledge encoded by means of computational ontologies. The proposed framework is centered around BPAL (Business Process Abstract Language), a logic-based language for modeling and reason with both the structural specification and the dynamic behavior of a business process from a workflow perspective. BPAL relies on a formalism, logic programming, that is particularly well-suited for its use within a wider knowledge representation framework, in particular in conjunction with rule based ontology languages to capture the semantics of a business scenario. The resulting Business Process Knowledge Base provides a uniform and formal representation framework, suited for automatic reasoning and equipped with a powerful inference mechanism supported by the solutions developed in the area of Logic Programming. At the same time it has been conceived to be used in conjunction with the existing BP management tools as an 'add-on' facility, by supporting BPMN, in particular its XPDL linear form, as a modeling notation, and OWL, for the definition of the reference ontologies.

**Keywords**: business process modeling, semantic enrichment, ontology, BPMN, logic programming, BPAL.

# 1 Introduction

Business Process (BP) management [1] is constantly gaining popularity in various industrial sectors, especially in medium to large enterprises, and in the public administration. BP modeling is a complex human activity, requiring a special competence and, typically, the use of a BP design tool. Several tools[1] are today available on the market, open source or free of charge. Many of these tools are able to provide, besides a graphical editor, additional services, such as some forms of verification, simulation of the designed processes, execution or (semi) automatic generation of executable code (e.g., in the form of BPEL[2] code). The availability of the mentioned tools has further pushed the diffusion of several languages (e.g., BPMN [2]) used both in the academic and in the industrial realities. But, despite the growing academic interest and the penetration in the business domain, heterogeneous and ad-hoc solutions that often lack a formal semantics have been so far proposed to deal with the different perspectives that are of interest for an effective BP management: workflow modeling, enactment, business rules verification, integration of organizational and data models, data flow, query and retrieval of BP fragments.

In order to increase the level of automation in the specification, analysis, implementation and monitoring of BPs, the adoption of results from the area of the Knowledge Representation is gaining much attention. The emerging stream of research in the so-called Semantic Business Process Management [3] advocates the enhancement of BP management systems[3] by means of well-established techniques from the area of the Semantic Web. In particular, computational ontologies supports unambiguous definitions of the entities occurring in the domain, and eases the interoperability between software applications and the reuse/exchange of knowledge between human actors. However, there are still several open issues regarding the combination of workflow languages (with their specific execution semantics) and ontologies, and the accomplishment of reasoning tasks involving both these components. With this respect, most of the proposals for the semantic enrichment of BPs either ignore the execution semantics of processes, adopting meta-model ontologies and focusing on structural properties through Description Logic reasoning; or, rarely, they are able to represent constraints on runtime sequences of behavior and suffer from a representational divide with respect to the original process specification.

This paper mainly intends to lay the formal foundations of a framework for the semantic augmentation of BP schemas, in order to provide a declarative representation of workflow models, enriched with domain knowledge encoded by means of computational ontologies. The proposed framework is centered around BPAL (Business Process Abstract Language) [4], a logic-based language for modeling and reason with both the structural specification and the dynamic behavior of a business process from a workflow perspective. BPAL relies on a formalism, logic programming, that is particularly well-suited for its use within a wider knowledge representation framework, in particular in conjunction with rule based ontology languages [5] to capture the semantics of a business scenario. The resulting Business Process Knowledge Base provides a uniform and formal representation framework, suited for automatic reasoning and equipped with a powerful inference mechanism supported by the solutions developed in the area of Logic Programming [6]. At the same time it has been conceived to be used in conjunction with the existing BP management tools as an 'add-on' facility, by supporting BPMN, in particular its XPDL [7] linear form, as a modeling notation, and OWL [8], for the definition of the reference ontologies.

---

[1] See for instance: Intalio BPMS Designer, Tibco Business Studio, ProcessMaker, YAWL, JBPM, Enhydra Shark

[2] Business Process Execution Language. See: http://www.oasis-open.org/specs/#wsbpelv2.0

[3] See, e.g., the SUPER (http://www.ip-super.org/), COIN (http://www.coin-ip.eu/) and PLUG-IT (http://plug-it.org/) European projects.

### 1.1 Business Process Knowledge Representation

The final goal of the proposed approach is to define a Business Process Knowledge Base (*BPKB*) as a collection of logical theories that formalize the knowledge about the various business processes constituting a BP repository. In particular, three different perspective will be taken into consideration to represent and reason about process knowledge: 1) the *structural specification*, i.e., a formal representation of the workflow graph associated with each process, 2) the *behavioral semantics*, i.e., a formal description of the execution of a BP, and 3) the *domain knowledge*, i.e., an ontology to be used for attaching a semantics with each individual participating in a BP (e.g., *actors*, *objects*, and *activities*).

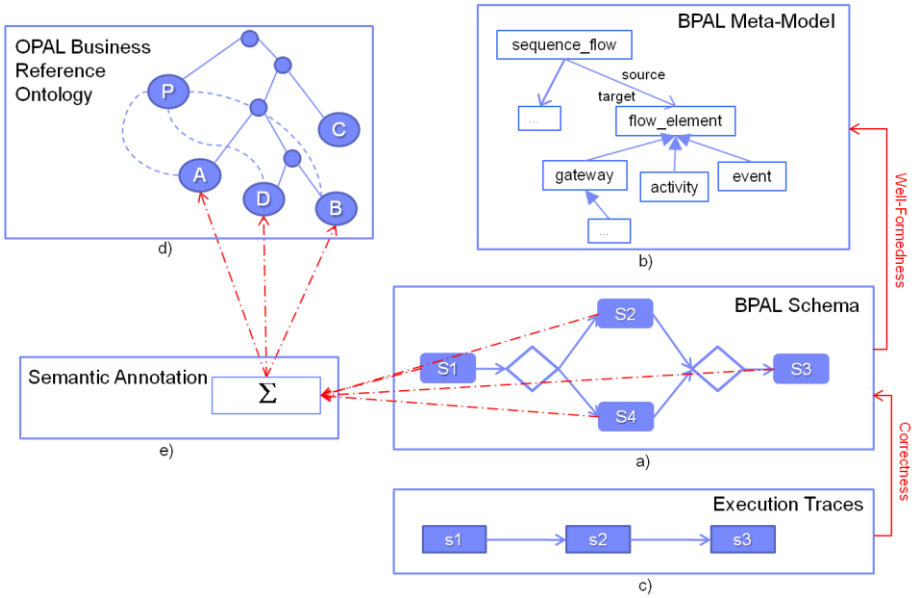


**Fig. 1.** Knowledge Representation Framework

The *BPKB* is based on the knowledge representation framework sketchily depicted in Figure 1. In this framework, well-formed BP schemas are encoded in BPAL (Figure1.a) according to the BPAL meta-model (Figure 1.b) and their behavioral semantics is defined in terms of the correct *execution traces* (Figure 1.c). Then business domain knowledge is captured by an Business Reference Ontology (Figure 1.d) and, finally, the Semantic Annotation (Figure 1.e) links elements of a BP schema to ontology concepts in order to provide a semantic description of the former in term of the latter. For the definition of the BRO we consider as the reference framework OPAL, proposed in [9] as an ontology representation methodology based on UML and OWL aimed at building business-oriented domain ontologies.

In the rest of the report the BPKB will be presented in details, introducing BPAL in Section 2, the meta-model theory in Section 3 and the trace theory in Section 4. Section 5 presents the enrichment of BP schemas through domain ontologies. The resulting knowledge representation framework and its inference capabilities are described in Section 6. Section 7 describes the software implementation. Related works are discussed in Section 8. Finally, conclusions in Section 9 end the report.

### 1.2 Background and Notation

In order to clarify the terminology and the notation used throughout the report, in this section we recall some basic notions of Logic Programming [6] and Description Logics [10] as foundations for the OWL [8] standard.

**Logic Programs.** A logic program is written using a language that consist of predicate, function, constant and variable symbols. We use capital letters for variables, lowercase strings for predicate symbols, function symbols and constants. A *term* is inductively defined as follows: each variable and each constant is a term, and if $f$ is a function symbol and $t_1 \ldots t_n$ are terms, $f(t_1,\ldots,t_n)$ is a term. An *atom* is a formula $p(t_1,\ldots,t_n)$, where $p$ is a predicate symbol and $t_1,\ldots,t_n$ are terms. A *clause* is a formula of the form $A_0 \leftarrow A_1 \wedge \ldots \wedge A_m \wedge not\ A_{m+1} \wedge \ldots \wedge not\ A_n$, where each $A_i$ is an atom. "*not*" stands for negation-as-failure, a logically non-monotonic form of negation whose semantics differs, in general, significantly from the semantics of classical negation ($\neg$). Intuitively, *not A* means "*A* is not possible to prove that *A* is true" (i.e., is unknown or false), whereas $\neg A$ means "A is false". A term (or atom, or clause) is said to be *ground* if no variable occurs in it. A clause $r$ of the form $A_0 \leftarrow$ (whose body is empty) is called a *fact.* Intuitively, each clause can be viewed as universally quantified at the outermost level. More precisely, each clause can be viewed as standing for the set of all its ground instances. A *logic program* is then a theory consisting of a set of general clauses. In particular, in this report we deal only with the class of *stratified* programs, where the negation connective "*not*" is interpreted according to the Perfect Model Semantics [11].

**Description Logics and OWL.** Description logics (DLs) are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. DLs are typically adopted for the definition of ontologies since on the one hand, the important notions of the domain are described by concept *descriptions*, i.e. expressions that are built from atomic concepts (usually thought as sets of individuals) and atomic roles (relations between concepts) using the concept and role constructors provided by the particular DL. On the other hand, DLs correspond to decidable fragments of classical FOL, and thus are equipped with a formal, *logic*-based semantics which makes such languages suitable for automatic reasoning.

DLs constitute the formal grounding of the Ontology Web Language (OWL), which is one of the most promising standards for ontology and meta-data sharing over the (semantic) web. The profiles introduced in the OWL2 specification define several sub-languages with different computational properties that can be considered as syntactic variants of well-known DL families. In particular, we will refer throughout the paper to the OWL-RL profile, since ontologies defined according to this profile can be mapped to logic programs.

OWL ontologies can thus be seen in terms of a correspondence to FOL, where concepts correspond to unary predicates, roles correspond to binary predicates and inclusion axioms correspond to implication. To be more precise, individuals are equivalent to FOL constants, concepts and concepts expressions are equivalent to FOL formulae with one free variable, and roles and role expressions are equivalent to FOL formulae with two free variables. Table 1 summarizes the above equivalences and put in relation the DL notation and the corresponding FOL formulae with some of the OWL statements, where $C$ and $D$ are concepts (OWL classes), $P$ and $Q$ are roles (OWL properties), $a$ and $b$ constants, $x$ and $y$ variables.

**Table 1.** Owl statements and FOL equivalence

| ***OWL Axiom*** | ***DL Expression*** | ***FOL Formula*** |
|---|---|---|
| *a type C* | $a : C$ | $C(a)$ |
| *a P b* | $(a,b) : P$ | $P(a,b)$ |
| *subClassOf* | $C \sqsubseteq D$ | $\forall x.C(x) \rightarrow D(x)$ |
| *disjointWith* | $C \sqsubseteq \neg D$ | $\forall x.C(x) \rightarrow \neg D(x)$ |
| *domain* | $\top \sqsubseteq \forall P^{-}.C$ | $\forall x,y.P(x,y) \rightarrow C(x)$ |
| *range* | $\top \sqsubseteq \forall P.C$ | $\forall x,y.P(x,y) \rightarrow C(y)$ |
| *transitiveProperty* | $P^{+} \sqsubseteq P$ | $\forall x,y,z.(P(x,y) \wedge P(y,z)) \rightarrow P(x,z)$ |
| *functionalProperty* | $\top \sqsubseteq \leq 1\ P$ | $\forall x,y,z.(P(x,y) \wedge P(x,z)) \rightarrow y=z$ |
| *inverseOf* | $P \equiv Q^{-}$ | $\forall x,y.P(x,y) \leftrightarrow Q(y,x)$ |
| ***OWL Constructor*** | | |
| *intersectionOf* | $C \sqcap D$ | $C(x) \wedge D(x)$ |
| *unionOf* | $C \sqcup D$ | $C(x) \vee D(x)$ |
| *allValuesFrom* | $\forall P.C$ | $\forall x.(P(x,y) \rightarrow C(y))$ |
| *someValuesFrom* | $\exists P.C$ | $\exists x.(P(x,y) \wedge C(y))$ |
| *complementOf* | $\neg D$ | $\neg D(x)$ |

## 2 Introducing BPAL

The Business Process Abstract modeling Language (BPAL) is a logic-based language conceived to provide a declarative modeling method capable of fully capturing procedural knowledge in a business process. BPAL constructs are common to the most used and widely accepted BP modeling languages (e.g., BPMN, UML activity diagrams, EPC) and, in particular, its core is based on BPMN 2.0 specification [2].

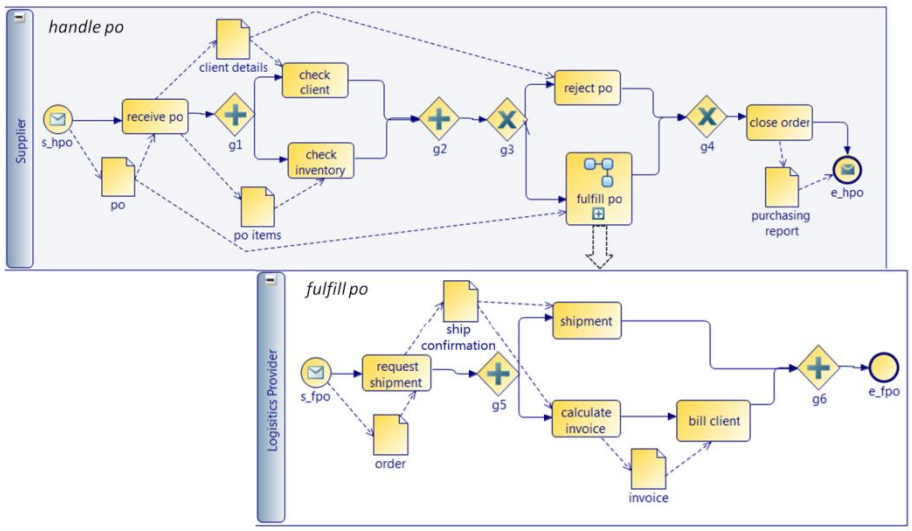


**Fig. 2.** Handle Order BP example

For illustration, consider the BP depicted in Figure 2, where an orchestration specifying the handling of a purchase order in an *eProcurement* scenario is represented using the BPMN notation. Upon receiving the purchase order from a customer, a supplier initiates two tasks concurrently, to verify the information provided

by the customer and to check product availability in the inventory. If the purchase order is accepted, then it is fulfilled. The activity *fulfill_po* is a *compound* activity (modeled as a BPMN reusable sub-process), representing the invocation of the corresponding process, where the shipment and the invoicing are executed by a logistics provider.

In Figure 3 the core elements of the BPAL meta-model are shown in a UML class diagram for sake of readability, while the formalization, including the axiomatisation of its semantics will be presented in the next sections.

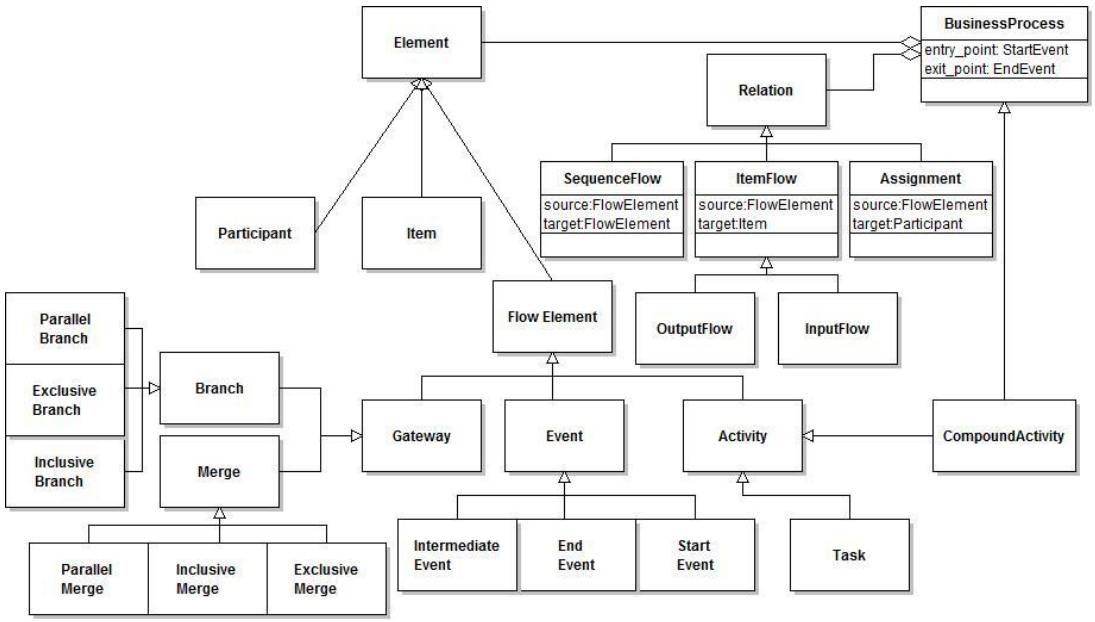


**Fig. 3.** BPAL Core Meta-Model

A *business process* is constituted by a set of *elements* and *relations* between elements appearing in the workflow graph, and it is associated to a unique *start event* and a unique *end event* that represent the entry point and the exit point, respectively, of the process. An *activity* is the key element of the business process, representing a unit of work performed within the process. A *task* represents an atomic activity (e.g., *bill_client*), i.e., no further decomposable, while a *compound activity* represents the invocation of composite (possibly remote) process, and it is associated with a workflow graph that provides the definition of its internal structure (e.g., *fulfill_po*). A process can thus be viewed as a hierarchy of activities (e.g., the composite activity *fulfill_po* occurs in the process *handle_po*). The sequencing of flow elements is specified by the *sequence flow* relation, and for branching flows, BPAL provides predicates representing *parallel* (AND), *exclusive* (XOR), and *inclusive* (OR) *branching/merging* of the control flow. An *item* represents a physical or information object (e.g., *invoice*) that is created and used during the execution of the process. An item holds the actual object instances that are produced during the process enactment and, hence, it is regarded as a variable. An *item flow* specifies that a flow element uses as *input* or produces as *output* a particular item. An item related to an item flow originating in a start event constitutes the input of the process (e.g., *handle_po* is triggered by receiving an order), while an item related to an item flow ending in an end event constitutes the output of the process (e.g., *handle_po* ends by sending back a final report). Finally, a *participant* is a generic notion representing a role within a company (e.g., *employee*), or a partner role (e.g.,

*manufacturer*) which is *assigned* to the execution of a collection of activities. It is worth noting that the *semantics* of the notions of *item* and *participant* are left intentionally underspecified, since an unambiguous definition of their meaning has to be provided in terms of a reference ontology through the semantic annotation, as shown in Section 6.

Formally a BPAL BP Schema (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 BPS elements (e.g., business activities, events, and gateways) and $p$ is a BPAL predicate. In Table 2 we list some of the BPAL predicates, while in Table 3 we report the BPAL translation of the *fulfill_po* BPS depicted in Figure 2.

**Table 2.** Excerpt of the BPAL language

| | |
|---|---|
| *bp(p,s,e)* | *p* is a process, with entry-point *s* and exit-point *e* |
| *element(x)* | *x* is an *element* occurring in some process |
| *relation(x,y,p)* | the *elements x* and *y* are in relation in the process *p* |
| *task(a)* | *a* is an atomic activity |
| *comp_act(a,s,e)* | *a* is a compound activity, with entry-point *s* and exit-point *e* |
| *seq(e1,e2,p)* | a sequence flow relation is defined between *el1* and *el2* in *p* |
| *par_branch (g)* | the execution of *g* enables all the successors flow elements |
| *par_mrg(g)* | *g* waits for the completion of all the predecessor flow elements |
| *exc_branch(g)* | the execution of *g* enables one of the successors flow elements |
| *exc_mrg(g)* | *g* waits for the completion of one of the predecessor flow elements |
| *inc_branch(g)* | the execution of *g* enables at least one of the the successors flow elements |
| *inc_mrg(g)* | *g* waits for the completion of the predecessor flow elements that will be eventually executed |
| *item(i)* | *i* is an data element |
| *input(a,i,p)* | the activity *a* uses as input the data element *i* in the process *p* |
| *output(a,i,p)* | the activity *a* uses as output the data element *i* in the process *p* |
| *participant(part)* | *part* is a participant |
| *assigned(a,part,p)* | the activity *a* is assigned to the participant *part* in the process *p* |

**Table 3.** BPAL representation of the eProcurement process

| | |
|---|---|
| *comp_act(fulfill_po, s_fpo, e_fpo)* | *item (order)* |
| *task(request_shipment)* | *item (ship_details)* |
| *task(calculate_invoice)* | *item (invoice)* |
| *task(bill_client)* | *output(s_fpo, order, fulfill_po)* |
| *task(shipment)* | *input(request_shipment, ship_details, fulfill_po)* |
| *par_branch(g5)* | *output(request_shipment, ship_details, fulfill_po)* |
| *par_merge(g6)* | *input(calculate_invoice, ship_details, fulfill_po)* |
| *seq(calculate_invoice, bill_client, fulfill_po)* | *input(shipment, ship_details, fulfill_po)* |
| *seq(s_fpo, request_shipment, fulfill_po)* | *output(calculate_invoice, invoice, fulfill_po)* |
| *seq(g5, shipment, fulfill_po)* | *input(bill_client, invoice, fulfill_po)* |
| *seq(g5, calculate_invoice, fulfill_po)* | *participant(logistics_provider)* |
| *seq(shipment, g6, fulfill_po)* | *assigned(request_shipment, logistics_provider, fulfill_po)* |
| *seq(bill_clientg, g6, fulfill_po)* | *assigned(calculate_invoice, logistics_provider, fulfill_po)* |
| *seq(request_shipment, g5, fulfill_po)* | *assigned(bill_client, logistics_provider, fulfill_po)* |
| *seq(g6, e_fpo, fulfill_po)* | *assigned(shipment, logistics_provider, fulfill_po)* |

## 3 BPAL Meta-Model Ontology

On top of the BPS modelling layer, we explicitly introduce a BP meta-modelling layer, formalized by the theory ***M*** which defines a set of structural properties of a BP schema that at this level is regarded as a labeled graph. The presence of a meta-model allows us to automatically prove the first fundamental property: the fact

that a BPAL process schema has been built in the respect of the meta-model. We will refer to such a property as *well-formedness*. Two categories of properties should be verified by a well-formed BPS: *i) local* properties related to the elementary components of the workflow graph (for instance, 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 process (for instance, every flow element must lie in a path from the start event to the end event).

A second category of properties regards the notion sub-process, that is particularly relevant for the retrieval of process fragments. We distinguish between well-formed sub-processes, with a unique entry point and a unique exit point, and *structured* sub-processes, where each branch point is matched with a merge point of the same type, and such branch-merge pairs are also properly nested.

Finally, ***M*** defines other structural properties of a BPS, related to the notions of paths and reachability between flow elements.

### 3.1 Axiomatization

In the rest of this section we describe the core of the meta-model of BPAL by means of a set of clauses constituting the theory ***M***. Hereafter, when it can be safely applied, we adopt the standard notation of set theory to ease the readability of the clauses.

**Connectivity Properties.** A first category of properties formalized in ***M*** is related to the graph structure underlining a BPS, that essentially encodes a workflow as a directed labelled graph. The verification of graph-based properties, such as the existence of a path between two activities, can be relevant for several reasons. First of all, while the analysis of the possible executions of a BPS (see next section) is of very high complexity, the retrieval and analysis of particular paths within a graph is typically polynomial. Then, the topological structure of a BPS is obviously connected to its possible executions. For instance, a path between two nodes implies that a possible execution where both occur in that order exists (assuming the BPS deadlock-free and without dead activities), or that only BP schemas that contain two given activities may potentially have executions where both occur.
The adjacency between flow elements in a BPS is defined as follows:

$$successors(X,SUCC,P) \leftarrow SUCC = \{Y \mid seq(X,Y,P)\}$$
$$predecessors(X,PRED,P) \leftarrow PRED = \{Y \mid seq(Y,X,P)\}$$

The basic form of reachability between two flow elements in a BPS is defined as the transitive closure of the *seq* relation, i.e.:

$$reachable(X,Y,P) \leftarrow seq(X,Y,P)$$
$$reachable(X,Z,P) \leftarrow seq(X,Y,P) \wedge reachable(Y,Z,P)$$

While *reachable* considers compound activities as simple nodes, *h_reachable* explores the hierarchical decomposition of a BPS, visiting also the process definitions associated to compound activities, i.e.:

$$h\_reachable(X,Y,P) \leftarrow seq(X,Y,P)$$
$$h\_reachable\ (X,Z,P) \leftarrow seq(X,Y,P) \wedge h\_reachable\ (Y,Z,P)$$
$$h\_reachable\ (X,Z,P) \leftarrow seq(X,Y,P) \wedge comp\_act(Y,S,E) \wedge h\_reachable\ (S,Z,Y)$$

Other variants of reachability may express further conditions on the workflow graph paths. For instance, *n_reachable(X,Y,N,P)*, holds if there is a path between *x* and *y* that do not traverse *n,* i.e.:

$$n\_reachable\ (X,Y,N,P) \leftarrow seq(X,Y,P) \wedge not\ Y=N$$
$$n\_reachable\ (X,Z,N,P) \leftarrow seq(X,Y,P) \wedge not\ Y=N \wedge n\_reachable\ (Y,Z,N,P)$$

An *element* (activity, data element or participant) occurs in a process (i.e., is part of the process description) if it is involved in some *relations* defined for that process, i.e.:

$$occurs(X,P) \leftarrow relation(X,Y,P)$$
$$occurs(Y,P) \leftarrow relation(X,Y,P)$$

Finally, we introduce *h_occurs(X,P),* i.e., the transitive closure of *occurs*, *onpath(X,P,S,E),* i.e., *x* occurs in a path from *s* to *e*, and *h_onpath(X,P,S,E),* i.e., a variant of the latter where the hierarchical decomposition is considered:

$$h\_occurs(X,P) \leftarrow occurs(X,P)$$
$$h\_occurs(X,P) \leftarrow occurs(X,P1) \wedge h\_occurs(P1,P)$$
$$onpath(X,S,E,P) \leftarrow reachable(S,X,P) \wedge reachable(X,E,P)$$
$$h\_onpath(X,S,E,P) \leftarrow h\_reachable(S,X,P) \wedge h\_reachable(X,E,P)$$

**Inclusion Axioms.** The set of *inclusion axioms* defines a taxonomy among the BPAL predicates, as informally described in Section 2. Some examples are reported below.

**Table 4.** BPAL Elements Inclusion Axioms

| | |
|---|---|
| *element(X) ← flow_el(X)* | *gateway(G) ← branch(G)* |
| *flow_el(X) ← event(X)* | *gateway(G) ← merge(G)* |
| *flow_el(X) ← activity(X)* | *branch(G) ← par_branch(G)* |
| *flow_el(X) ← gateway(X)* | *bp(P,X,Y) ← comp_act(P,X,Y)* |
| *event(X) ← start_ev(X)* | *element(PART) ← participant(PART)* |

**Table 5.** BPAL Relations Inclusion Axioms

| | |
|---|---|
| *flow_el(X) ← relation(X,Y,P)* | *item_flow(X,Y,P)←input(X,Y,P)* |
| *relation(X,Y,P) ← seq(X,Y,P)* | *item(Y)←item_flow(X,Y,P)* |
| *flow_el(X) ← seq(X,Y,P)* | *relation(X,Y,P) ← assignment(X,Y,P)* |
| *relation(X,Y,P) ← data_flow(X,Y,P)* | *participant(Y) ← assignment(X,Y,P)* |

**Well-formedness.** A well-formed, i.e., syntactically correct, BPS must satisfy a number of constraints. The main structural restrictions imposed to a well-formed BPS are the following:

1. every process is assigned to a unique start and a unique end event;
2. every flow element occurs on a path from the start to the end event;
3. only gateways will have multiple incoming/outgoing sequence flows;
4. there are no cycles in the hierarchy of compound activities.

Several categories of constraints are defined in ***M****:*

- ***Domain constraints*** express disjointness relationships among BPAL elements, e.g.:

$$illformed(P) \leftarrow flow\_el(X) \wedge not\ activity(X) \wedge not\ gateway(X) \wedge not\ event(X) \wedge occurs(X,P)$$
$$illformed(P) \leftarrow activity(X) \wedge event(X) \wedge occurs(X,P)$$

- ***Occurrence constraints*** impose restriction on the occurrence of elements within a BPS, e.g.:

$$illformed(P) \leftarrow bp(P,S,E) \wedge start\_ev(S1) \wedge occurs(S1,P) \wedge not\ S1=S$$

$$illformed(P) \leftarrow occurs(X,P) \wedge bp(P,S,E) \wedge not\ onpath(X,P,S,E)$$

- ***Hierarchical constraints*** impose restrictions on the hierarchical decomposition of processes through the use of compound activities, e.g.:

$$illformed(P) \leftarrow h\_occurs(P,P)$$

- ***Relation constraints*** impose restrictions on the way *elements* can participate in *relations,* e.g.*:*

$$illformed(P) \leftarrow task(X) \wedge seq(X,Y,P) \wedge seq(X,Z,P) \wedge not\ \ Y=Z$$
$$illformed(P) \leftarrow start\_ev(X) \wedge seq(Y,X,P)$$
$$illformed(P) \leftarrow end\_ev(X) \wedge seq(X,Y,P)$$
$$illformed(P) \leftarrow branch(X) \wedge occurs(X,P) \wedge successors(X,SUCC,P) \wedge |SUCC| = 1$$

Formally, a BPS $\boldsymbol{B}$ with process identifier $p$ is well-formed if the atom *illformed(p)* is false in the *Perfect model* of the stratified program $\boldsymbol{B} \cup \boldsymbol{M}$, i.e., $Perf(\boldsymbol{B} \cup \boldsymbol{M}) \not\models illformed(p)$. A well-formed sub-process is then defined as a single-entry-single-exit (SESE) region of a well-formed process *p,* i.e.:

$$wf\_sub\_proc(P,X,Y) \leftarrow bp(P,S,E) \wedge not\ n\_reachable(X,E,Y,P) \wedge not\ n\_reachable(S,Y,X,P) \wedge not\ n\_reachable(Y,Y,X,P) \wedge not\ n\_reachable(X,X,Y,P)$$

**Structuredness.** Structuredness is a syntactic restriction that guarantees important properties, such as the absence of deadlocks, livelocks and dead activities. This features make structured process schemas easier to comprehend and less error-prone than unstructured ones [12], and ease the implementation of BPMS based on languages restricted in such a way.

In order to deal with situations where only structured (sub-)processes are of interest (e.g., retrieval of sub-processes to be implemented in a structured language such as BPEL [13]), $\boldsymbol{M}$ defines the predicate *str_sub_proc(p,s,e)*, recursively defined as follows:

- a task, and similarly an event, is a structured sub-process:

$$str\_sub\_proc(P,X,X) \leftarrow task(X) \wedge occurs(X,P)$$

- a compound activity is a structured sub-process if the corresponding process definition is structured:

$$str\_sub\_proc(P,X,X) \leftarrow comp\_act(X,S,E) \wedge occurs(X,P) \wedge str\_sub\_proc(X,S,E)$$

- the concatenation of two structured sub-processes is structured:

$$str\_sub\_proc(P,X,Y) \leftarrow str\_sub\_proc(P,X,W) \wedge not\ branch(W) \wedge seq(P,W,Z) \wedge not\ merge(Z) \wedge str\_sub\_proc(P,Z,Y)$$

- a sub-process started by a branch point $x$ and ended by a merge point $y$ is structured if (i) $x$ and $y$ are of the same type, and (ii) each successor of $x$ is the entry point of a structured sub-process whose exit point is a predecessor of $y$, i.e.:

$$str\_sub\_proc(P,X,Y) \leftarrow same\_type(X,Y) \wedge successors(X,SUCC,P) \wedge predecessors(Y,PRED,P) \wedge all\_structured(SUCC,PRED,P)$$

$$all\_structured(L1,L2,P) \leftarrow X \in L1 \wedge Y \in L2 \wedge str\_sub\_proc(P,X,Y) \wedge all\_structured(L1\text{-}\{X\},\ L2\text{-}\{Y\},P)$$

# 4 BPAL Behavioral Semantics

The behavioral semantics of BPAL BP schemas is based on the Fluent Calculus, a calculus for actions and changes introduced in [14], rooted in the logic programming paradigm. More precisely, we adapt the method described in [14], where *plans* and *states* are *reified*, that is, they are represented as terms and, thus they can be manipulated by logic clauses. *States* are the basic entities of action theories, representing a snapshot of the underlying dynamic system, i.e., the part of the world being modeled, at a given instant of time. Central to the axiomatization technique of the Fluent Calculus is the representation of states as sets of *fluents*, i.e., atomic properties represented as terms. The adequate treatment of these terms requires a (domain-independent) equational theory, which, essentially, formalizes crucial properties of the set[4] data-structure.

We explain here the notation used from now on. A fluent is of the form $f(t_1,\ldots,t_n)$, where $f$ is a fluent symbol and $t_1,\ldots,t_n$ are ground terms. We then assume a closed-world interpretation of a state, considering a fluent $\varphi$ true in a state $I$ if $\varphi \in I$, *false* otherwise.

In this frame, we model the behavior of a BP by defining a set of clauses to formalize the relation *result(I,A,F)*, that holds if the *action A* can be executed in the state $I$ leading to the state $F$. Here we use the term *action* with the meaning commonly adopted in AI planning, i.e., a behavior that changes the state of the world. In the proposed axiomatization, we consider two actions: *begin(A)* which starts the enactment of $A$, and *complete(A),* which represents the completion of the enactment of $A$. The execution of a process is then modeled as an *execution trace*, corresponding to a plan, i.e., the execution of a sequence of actions leading from the initial state, where the start event is enabled, to the final state, where the end event has been executed. The predicate *trace(I,T,F)* holds if $T$ is a sequence of actions from the state $I$ to the state $F$.

$$trace(I,[],I).$$
$$trace(I,[A|TR],F) \leftarrow result(I,A,I_1) \wedge trace(I_1,TR,F)$$

A state $F$ is reachable from $I$ if there is a finite sequence of actions from $I$ to $F$:

$$reachable\_state(I,I).$$
$$reachable\_state\ (I,F) \leftarrow result(I,A,I_1) \wedge reachable\_state\ (I_1,F)$$

## 4.1 Control Flow

In this section we present a formalization of the behavioural semantics of the BPAL flow elements. As anticipated, BPAL has been strongly inspired by the BPMN language, and the proposed formalization mainly refers to the BPMN (informal) semantics. According to this, while most of the notions considered in this paper (e.g., parallel or exclusive branching/merging) are common to the great majority of workflow languages, when several interpretations are possible for a given construct, e.g., inclusive merge or data flow, we stick to the BPMN one. We introduce two fluents for the purpose of modeling the control flow:

- *enabled(A,B,P)*, means that the flow element $A$ has been executed, enabling for the execution the flow element $B$ in the enactment of the process $P$;
- *enacting(A,P),* where $A$ is an activity, means that $A$ is being enacted.

[4] Also multi-sets can be considered for the representation of states, but this extension is not treated in this report

*Activity and Event execution*

The only precondition for the execution of activities is the completion of the unique predecessor flow element, while the effects of their execution vary depending on the type. A task simply enables its only successor.

$$result(I,begin(A),F) \leftarrow task(A) \wedge enabled(X,A,P) \in I \wedge F = (I - \{enabled(X,A,P)\}) \cup \{enacting(A,P)\}$$

$$result(I,complete(A),F) \leftarrow task(A) \wedge enacting(A,P) \in I \wedge seq(A,Y,P) \wedge F = (I - \{enacting(A,P)\}) \cup \{enabled(A,Y,P)\}$$

A compound activity enabled for the execution leads to the enactment of the associated process definition, and it can complete only when the process definition associated to a compound activity is completed.

$$result(I,begin(A),F) \leftarrow comp\_act(A,S,E) \wedge enabled(X,A,P) \in I \wedge enacting(A,P) \notin I \wedge F = (I - \{enabled(X,A,P)\}) \cup \{enabled(start,S,A), enacting(A,P)\}$$

$$result(I,complete(A),F) \leftarrow comp\_act(A,S,E) \wedge enabled(E,end,A) \in I \wedge enacting(A,P) \in I \wedge seq(A,Z,P) \wedge F = I - \{enacting(A,P), enabled(E,end,A)\}) \cup \{enabled(A,Z,P)\}$$

According to BPMN, intermediate events represent "something that occur during the process execution", and hence are intended as punctual pattern of behavior that are registered at a given time-point. This reflect on the proposed axiomatization by modeling the execution of events as a single state transition.

$$result(I,complete(A),F) \leftarrow intermediate\_event(A) \wedge enabled(X,A,P) \in I \wedge seq(A,Y,P) \wedge F = (I - \{enabled(X,A,P)\}) \cup \{enabled(A,Y,P)\}$$

A special treatment is reserved to start and end events. In the initial state of a process execution the fluent *enabled(start,S,P)* holds, where *S* is the start event associated to the process *P*, and *start* is a reserved constant.

$$result(I,complete(S),F) \leftarrow start\_event(S) \wedge enabled(start,S,P) \in I \wedge seq(S,A,P) \wedge F = (I - \{enabled(start,S,P)\}) \cup \{enabled(S,A,P)\}$$

The execution of an end event leads to the final state of a process execution, in which the fluent *enabled(E,end,P)* holds, where *E* is the end event associated to the process *P*, and *end* is a reserved constant.

$result(I,complete(E),F) \leftarrow end\_event(E) \wedge$
$enabled(X,E,P) \in I \wedge$
$F = (I - \{enabled(X,E,P)\}) \cup \{enabled(E,end,P)\}$

*Branching behaviors*

When a branch point is executed, a subset of its successors are selected for the execution. We consider here three different branching behaviors. Parallel branches enables all their successors, initiating concurrent executions. Exclusive branches lead to the concurrent execution of a subset of their successors. Exclusive branches enable exactly one successor.

$result(I,complete(B),F) \leftarrow par\_branch(B) \wedge$
$enabled(X,B,P) \in I \wedge$
$successors(B,OUT,P) \wedge$
$F = (I - \{ enabled(X,B,P)\}) \cup$
$\{ enabled(B,A,P) \mid A \in OUT\}$

$result(I,complete(B),F) \leftarrow inc\_branch(B) \wedge$
$enabled(X,B,P) \in I \wedge$
$successors(B,OUT,P) \wedge$
$O \subseteq OUT \wedge$
$F = (I - \{ enabled(X,B,P)\}) \cup$
$\{enabled(B,A,P) \mid A \in O\}$

$result(I,complete(B),F) \leftarrow exc\_branch(B) \wedge$
$enabled(X,B,P) \in I \wedge$
$seq(B,A,P) \wedge$
$F = (I - \{ enabled(X,B,P)\}) \cup \{enabled(B,A,P)\}$

*Merging behaviors*

An exclusive merge can be executed every time one of its predecessors has been completed, while a parallel merge requires that each of its predecessors has been executed:

$result(I,complete(M),F) \leftarrow exc\_merge(M) \wedge$
$enabled(X,M,P) \in I \wedge$
$seq(M,Y,P) \wedge$
$F = (I - \{enabled(X,M,P)\}) \cup \{enabled(M,Y,P)\}$

$result(I,complete(M),F) \leftarrow par\_merge(M) \wedge$
$predecessor(M,IN,P) \wedge$
$\{enabled(X,M,P) \mid X \in IN\} \subseteq I \wedge$
$seq(M,Y,P) \wedge$
$F = (I - \{enabled(X,M,P) \mid X \in IN \}) \cup \{enabled(M,Y,P)\}$

For the inclusive merge several operational semantics have been proposed [15,16], due to the complexity of its non-local semantics. An inclusive merge is supposed to be able to synchronize a varying number of threads, i.e., it is executed only when at least one of its predecessors has been executed and no other will be eventually executed. Here we refer to the semantics described in [17] adopted by BPMN, stating that an inclusive merge *m* can be executed if:

- at least one predecessor has been executed,

- for each unexecuted predecessor *e*, there is no an executed element upstream *u* that may lead to the execution of *e.* An element *u* is considered upstream if:
  - there is a path from *u* to *e* that does not visit *m,* and
  - there is no a path from *u* to an executed predecessor of *m.*

Formally, an inclusive merge *M*, with predecessor elements *IN*, and completed predecessors *SYN*, has an executed element upstream if:

$$
\begin{aligned}
&el\_upstream(M,SYN,IN,P,I) \leftarrow Z \in (IN - SYN) \wedge \\
&\quad enabled(U,X,P) \in I \wedge \\
&\quad nreachable(U,Z,M,P) \wedge \\
&\quad not(\exists K \in SYN \wedge reachable(U,K,P))
\end{aligned}
$$

Hence, the semantics of an inclusive merge is formalized as follows:

$$
\begin{aligned}
&result(I,M,F) \leftarrow inc\_merge(M) \wedge \\
&\quad predecessor(M,IN,P) \wedge \\
&\quad SYN \subseteq IN \wedge \\
&\quad \{ enabled(X,M,P) \mid X \in SYN \} \subseteq I \wedge \\
&\quad not\ el\_upstream(M,SYN,IN,P,I) \wedge \\
&\quad seq(M,Y,P) \wedge \\
&\quad F = (I - \{ enabled(X,M,P) \mid X \in SYN \}) \cup \\
&\qquad \{enabled(M,Y,P)\}
\end{aligned}
$$

### 4.2 Item Flow

A requirement of BP modeling is to be able to model the physical and information items that are produced and consumed by the various activities during the execution of a process. For the formalization of the *item flow* semantics, we commit to the BPMN standard, where the so called *data objects* are used to store information created and read by activities.

In BPAL items are essentially regarded as variables, hence there is a single instance of a given item any time during the execution that may be (over-)written by some activity. We consider two main types of relationships between activities and items. First of all, an activity may *use* a particular item (input relation). This implies that the item is expected to have a value before the activity is executed. Second, an activity may produce a particular value (output relation), causing the item to gets a new value. If it did not have a value yet, it is created, otherwise it is overwritten. It is worth noting that the item flow is not necessarily overimposed over the control flow, but they interact for the definition of the process behavior. For instance, an activity expecting a value from a given item, may also cause a deadlock if this condition will not be eventually satisfied.

Item flow is modeled through the fluent *updated(a,i,p),* representing the situation in which the item *i* has been (over-)written by the flow element *a* in the enactment of the process *p.* We report the extended formulation of the axiom regarding the task execution considering the item flow semantics.

$$
\begin{aligned}
&result(I,begin(A),F) \leftarrow task(A) \wedge \\
&\quad enabled(X,A,P) \in I \wedge \\
&\quad \{updated(A1,I,P) \mid input(A,I,P)\} \subseteq I \wedge \\
&\quad F = (I - \{enabled(X,A,P)\}) \cup \{enacting(A,P)\}
\end{aligned}
$$

*result(I,complete(A),F)* ← *task(A)* ∧
*enacting(A,P)* ∈ *I* ∧
*seq(A,Y,P)* ∧
*F= ((I – {enacting(A,P)}) –{written(EL,I,P)|output(A,I,P)}) ∪ {enabled(A,Y,P)} ∪ {updated(A,I,P)|output(A,I,P)}*

### 4.3 State Updates

In the previous section we introduced a calculus to formalize a set of constraints on the possible executions of a process deriving from a workflow specification. However, not all the relevant knowledge regarding a process enactment can be captured by a workflow specification, that mainly focus on the control flow perspective. Hence, while a workflow defines the execution ordering of activities, it does not provide any information regarding how the world changes during the enactment or what are the conditions under which a task can be executed successfully. The main advantage of the proposed approach is the possibility of capturing in an AI fashion pre-conditions and effects related to the execution of actions, in terms of state updates.

Considering BPMN again, simple forms of preconditions and effects can be obtained by assigning a *status* to data objects. Each data object is assumed to be in a certain status at any time during the execution of the process. This status may be changed through the execution of activities. It can be specified which status a data element must be in before an activity can start (precondition) and which status a data element will be in after having completed an activity (effect). For instance, in our running example we could state that the task *reject_po* set the status of the *po* to cancelled, while the compound activity *fulfill_po* require the item *po* in the status *confirmed.*

We employ to this end a fairly general notion of action specification, given in terms of *action descriptions* of the form

$$action(p, a, pre^+, pre^-, eff^+, eff^-)$$

where:

- $pre^+$ is the set of fluents[5] that must hold in the state *I* to execute *a* in the process *p,* i.e.: $pre^+ \subseteq I$;
- $pre^-$ is the set of fluents that must not hold in the state *I* to execute *a* in the process *p,* i.e.: $pre^- \cap I = \emptyset$ *I;*
- $eff^+$ and $eff^-$ are the set of fluents representing the effects of *a* in the process *p,* i.e., the execution of *a* in the state *I* yields to the state $(I - eff^-) \cup eff^+$.

Examples of simple descriptions are the following:

*action(handle_po,reject_po,[],[],[status(po,cancelled)],[status(po,X)])*
*action(handle_po,fulfill_po,[status(po,confirmed)],[],[status(po,completed)],[status(po,X)])*

The execution of a task considering the state update is then defined as:

*result(I,begin(A),F)* ← *task(A)* ∧
*enabled(X,A,P)* ∈ *I* ∧
*{updated(A1,I,P) |input(A,I,P)}* ⊆ *I* ∧
$action(P, A, pre^+, pre^-, eff^+, eff^-)$ ∧
$pre^+ \subseteq I$ ∧
$pre^- \nsubseteq I$ ∧
*F= (I – {enabled(X,A,P)} ) ∪ {enacting(A,P)}*

[5] Please note that here a set of fluents essentially corresponds to their logical conjunction.

$result(I,complete(A),F) \leftarrow task(A) \wedge$
$enacting(A,P) \in I \wedge$
$seq(A,Y,P) \wedge$
$action(P,A,pre^+,pre^-,eff^+,eff^-) \wedge$
$F = (I - \{enacting(A,P)\} - \{written(EL,I,P) | output(A,I,P)\} - eff^-) \cup \{enabled(A,Y,P)\} \cup \{updated(A,I,P) | output(A,I,P)\} \cup eff^+$

### 4.4 Conditional Control Flow

Decision points (inclusive or exclusive) in a BP schema depend on conditions that usually take the form of tests on the value of the items that are passed between the activities. In general this conditions can be checked only at run-time, but in some cases they depend on particular effects of some task previously executed. Similarly to the action descriptions provided for activities, we can define guards expression to be attached to the outgoing sequence flows of exclusive and inclusive branch points. Depending on the current state, such expressions can be evaluated to select the executable flow element. A guard description is described as a pair

$$guard(seq(branch,element,proc),pre)$$

where $seq(branch,element,proc)$ is a sequence flow edge and $pre$ is the set of fluent expressions that must hold in a state $I$ to enable $element$ after the execution of $branch$ in the enactment of the process $proc$.

The execution of an exclusive branch considering guard descriptions is then defined as:

$result(I,complete(B),F) \leftarrow exc\_branch(B) \wedge$
$enabled(X,B,P) \in I \wedge$
$seq(B,A,P) \wedge$
$guard(seq(B,A,P),PRE) \wedge$
$PRE \subseteq I \wedge$
$F = (I - \{ enabled(X,B,P)\}) \cup \{enabled(B,A,P)\}$

## 5 Enriching BP Schemas with Domain Knowledge

Semantic Annotation is a common notion in the Semantic Web related research. It consists essentially in describing resources through the vocabulary defined in some ontology, that provides the underlying semantics for the terminology adopted in particular application domain. In this context, according to [18], ontologies are considered as engineering artifacts, constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words. The application of ontologies for the semantic enrichment of resources focuses on their role as sharable and reusable representation of knowledge to ease the management and exchange of information.

In [18], Guarino proposed a classification of the different kinds of ontologies that can be developed within an information system according to their level of generality and dependence on a particular task or application:

- *Top-level* ontologies describe very general concepts which are independent of a particular problem or domain;
- *Domain* ontologies and *task* ontologies capture, respectively, the knowledge within a particular domain (e.g., eProcurement, manufacturing) or related to generic tasks and activities (like purchasing or selling), by specializing the terms introduced in the top-level ontology;

- *Application* ontologies provide the specific vocabulary required to describe a certain task enactment in a particular application context. They typically make use of both domain and task ontologies.

In the BPKB, the Business Reference Ontology (BRO) is intended to describe the vocabulary relevant to a given business domain and to the related processes. Conversely, the conceptualization provided by a BPS is strongly focused on a particular application scenario, describing how activities have to be conducted to successfully realize a given goal. Then, the Semantic Annotation of a BPS w.r.t. a BRO aims at the description of the element constituting a process workflow in terms of the domain and task knowledge provided by the BRO. In this perspective, the procedural and application oriented knowledge described in a BPS can be complemented by a representation of the domain knowledge in order to, e.g.:

- Arrange activities in taxonomic and meronymic hierarchies, supporting different levels of abstraction. E.g., *Calculating_Invoice* can be described as a specialization of *DocumentProcessing* that is part of the *Invoicing* process.
- Provide an alignment of the different terminology and conceptualization used in different BP schemas. Thus, we use ontologies to align domain specific terminology used in models and identify relationships among them. E.g., the participants *Seller* and *Provider* occurring in different process schemas may actually refer to the same notion, for instance represented in the BRO by the *Supplier* concept.
- Facilitate the semantic discovery and navigation of process fragments, by querying process schemas in term of the ontology vocabulary taking advantage of the semantic relations defined in the ontology, such as *subsumption*.

In the rest of this Section we describe the representation of the BRO through the OPAL methodology and its use within the BPKB for the Semantic Annotation of BP schemas.

### 5.1 OPAL

For the definition of the BRO we consider as the reference framework OPAL, proposed in [9] as an ontology representation methodology based on UML and OWL aimed at building business-oriented domain ontologies. OPAL organizes concepts in a number of conceptual categories (referred to as *kinds*) to support the domain expert in the conceptualization process, identifying active entities (actors), passive entities (objects), and transformations (processes). The development of the ontology is guided by a use-case driven, iterative and incremental process, derived from the large experience drawn in the software engineering area, with particular reference to the UP [19] software development framework. OPAL has been tested and validated in several national and international projects and applications, showing its effectiveness and high acceptance among business experts.

Figure 4 depicts the main OPAL kinds and semantic relations, organized in an UML diagram, briefly introduced in the following:

**Actor**. An active (agentive) entity of a business domain, able to activate, perform, or monitor a business process. The domain expert, in analyzing the reality, is asked to identify relevant actors that operate producing, updating or consuming business objects. Actors can also be further specialized into *human actors*, *artificial actors* and *organizational actors*, which represents, respectively, humans, software agents and organization or organizational unit (departments, areas and divisions). Examples of actor specialization are the roles played in particular contexts by agentive entities, e.g., the manager of a department or an employee (human actors), the supplier of a B2B transaction (organizational actor), the accounting software system (artificial actor).

**Object**. A category that gathers passive entities on which one or more business processes operate. Object can be partitioned into *physical objects,* occupying a space region as long as they exists (e.g., a car), or *data objects*, representing a piece of information, independently from its concrete realization, e.g., two prints of the same purchase order represents the same data object realized into two different physical supports. Objects produced or consumed by a process are classified as *artefacts* in the scope of that process, while objects used during, or to support, the execution of processes are considered *resources.* Artefacts and resources are not disjoint classes, since they are roles that individuals play in a process, i.e., the same individual can be produced by a given process used as resource in the scope of another process. A business object document (*document* for short) is a further refinement of a *data object* that represents a category of information objects manipulated during the enactment of a process (e.g., purchase order, invoice) and related to an attribute that represent its information content.

**Process**. A business activity or operation aimed at the satisfaction of a business goal. A *process* operates on a set of artefacts (*consuming*, *modifying*, *producing* or *terminating* them), requires *resources*, depends on the participation of actors for its realization and can be decomposed into other processes. Notice that here we do not distinguish between e.g., *processes, events* or *states,* and we use the term *process* as representative of the general category usually referred in literature as *occurrent* (or *perdurant*), i.e., the class of individuals that happen in time and are constituted by temporal parts.

**Message.** A *message* represents an information exchange realized during an interaction (e.g., request, response) between actors during the enactment of a process. Such interaction is characterized by a content that is typically a *document* (e.g., a RFQ-message, carrying a request for quotation). In OPAL we adopted the FIPA[6] approach, based on 23 message types, related to different kinds of communicative acts. Examples are: *refuse,* i.e., the action of refusing to perform a given action, and explaining the reason for the refusal.; *confirm, i.e.,* the sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition*; request, i.e.,* the sender requests the receiver to perform some actions.

**Attribute**: An *attribute* represents a measurable property associated to an individual (e.g., weight, name, color), in such a way that the attribute exists as long as the individual exists. Attributes are related to individuals through the *relatedness* relation. The value of an attribute is given in term of a value space, e.g., the name of the individual *John* is represented by an instance of the attribute *name* where the value is represented by the constant "John" that has to be interpreted according to the value space defined, e.g., for the XML Schema Datatype String. Then, atomic attributes, models elementary information (e.g., street name), and complex attributes models structured information (e.g., address). Essentially, a complex attribute is defined as an aggregation of lower level complex and/or atomic attributes.

[6] Foundation for Intelligent Physical Agents(FIPA): /http://www. fipa.org.

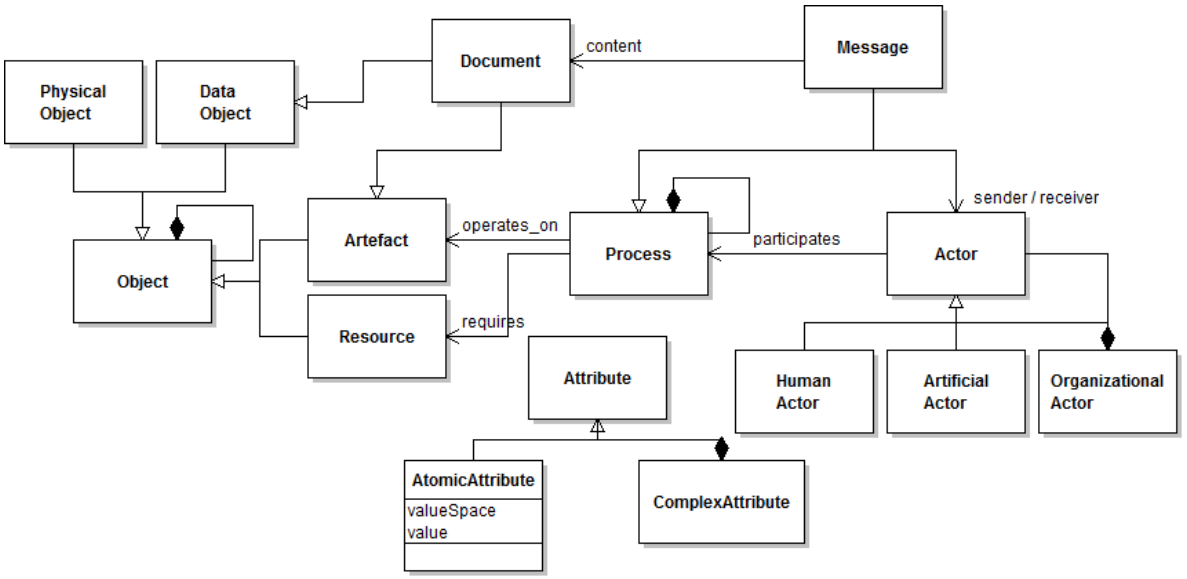


**Fig. 4.** OPAL Kinds

### 5.2 OPAL RDF Vocabulary

The OPAL Vocabulary is an application of the Resource Description Framework (RDF) derived from OPAL. Then, it consists of a set of RDF properties and RDFS classes that can be used to describe a set of concepts and their relationships according to the OPAL method. The use of RDF allows data to be linked to and/or merged with other RDF data by Semantic Web applications. In practice, this means that concepts declared within an OPAL ontology may be linked and hence defined in terms of external resources, such as existing open ontologies available on the web.

Classes in the vocabulary corresponds to *kinds* (e.g., *opal:Actor, opal:Process*), and, according to [20], the axiomatization specifies:

- ISA relations, e.g., $Artefact \sqsubseteq Object$;
- Disjointness between kinds, e.g., $Actor \sqsubseteq \neg Object$ and $Process \sqsubseteq \neg Object$;
- Property restrictions, e.g., $PhysicalObject \sqsubseteq \forall component.PhysicalObject$;
- Participation constraints, e.g., $Artifact \sqsubseteq \exists creates^{-}$ and $Process \sqsubseteq \exists participates^{-}$.

Properties in the vocabulary derives from:

1. Metadata (encoded as OWL *annotation properties*) commons to all the *kinds,* such as:
   - *label:* the preferred term to refer the concept;
   - *description:* a natural language description of the concept;
   - *references:* documental source used for the definition of the concept;
   - *terminology:* a set of terms that can be considered synonyms of the preferred term used as label;
   - *author:* the person that entered the concept specification in the ontology;
   - *defined/updated:* the date when the concept was first defined, and then updated.
2. Specific relations defined among kinds, such as the ones listed in Table 6.

**Table 6.** Basic OPAL relations

| | **subPropertyOf** | **domain** | **range** | **characteristics** |
|---|---|---|---|---|
| *participates* | *internal_process*$^-$ ∘*participant* | *Process* | *Actor* | |
| *performs* | *participant* | | | |
| *controls* | *participant* | | | *inverse functional* |
| *supports* | *participant* | | | |
| *sender* | *participant* | *Message* | | *functional* |
| *receiver* | *participant* | *Message* | | *functional* |
| *operates_on* | *internal_process*$^-$ ∘*operates_on* | *Process* | *Artifact* | |
| *consumes* | *operates_on* | | | *inverse functional* |
| *modifies* | *operates_on* | | | |
| *creates* | *operates_on* | | | *inverse functional* |
| *terminates* | *operates_on* | | | *inverse functional* |
| *content* | | *Message* | *Document* | *functional* |
| *requires* | *internal_process*$^-$ ∘*requires* | *Process* | *Resource* | |
| *relatedness* | | | *Attribute* | |
| *part_of* | | | | *irreflexive, antisimmetric, transitive* |
| *member* | *part_of* | *Actor* | *OrganizationalActor* | |
| *component* | *part_of* | *Object* | *Object* | |
| *internal_process* | *part_of* | *Process* | *Process* | |
| *sub_attribute* | *part_of* | *Attribute* | *ComplexAttribute* | |

Figure 5 shows an excerpt of an exemplary BRO related to the *handle_po* BPS, where three concepts hierarchies, having root in *opal:Process, opal:Actor* and *opal:Object* respectively, are depicted.

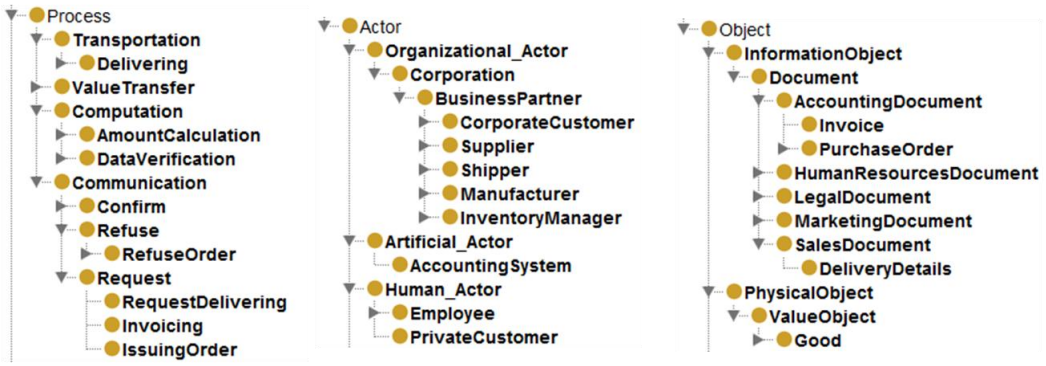


**Fig. 5.** Business Reference Ontology

### 5.3 Semantic Annotation

A *Semantic Annotation* defines a correspondence between elements of the BPS and concepts of the Reference Ontology, in order to describe the meaning of the former in terms of a suitable conceptualization of the domain of interest provided by the latter. To establish a general semantic association between the linked entities inherent in their meaning, we define the relation $\sigma \subseteq BpsEl \times Concept$, where *BpsEl* is an element of a BPS, and *Concept* is either

- a named concept defined in the BRO, e.g. *Shipper*;
- a complex concept, defined by a class expression, e.g. *Shipper* ⊓ *InventoryManager*.

Different BP elements could be annotated with respect to the same concept (since they could appear in several BPS) and not all the concepts have to be mapped with a BPS counterpart. Examples of annotation are reported in Table 7. The activity *shipment* is defined as a *Delivering* where is also realized a *ValueTransfer* related to some sort of *Good.* The s*hip_confirmation* item, input of shipment*,* is a *DeliveryDetails* document that constitute the content of a *confirmation message* exchanged between a *Supplier* and a *Shipper*. Finally, the participant *LogisticsProvider*, is a business partner able to act both as a *Shipper* and as an *InventoryManager.*

**Table 7.** Annotation Examples

| BPAL Element | Type | Annotation Expression |
|---|---|---|
| shipment | Activity | *bro:Delivering* ⊓ *bro:ValueTransfer* ⊓ ∀*opal:requires.bro:Good* |
| ship_confirmation | Item | *bro:DeliveryDetails* ⊓ ∃*opal:content*¯*.(bro:Confirm* ⊓ ∃*opal:receiver.bro:Supplier* ⊓ ∃*opal:sender.bro:Shipper)* |
| logistics_provider | Participant | *bro:Shipper* ⊓ *bro:InventoryManager* |

Even though the conceptualization introduced in a BPS differs on scope and purpose from the one provided by a reference domain ontology, some criteria may be introduced to put them in relation. To this end, we introduce a number of axioms to define constraints on the annotation. The presented axiomatization is very general and domain independent, and is intended as a starting point for further extensions where more specific constraints are formulated to accommodate the requirements of the particular domain at hand. Hereafter we present OWL expressions using the triple notation by means of the ternary predicate *t(s,p,o)*, representing a generalized RDF triple (with subject *s*, predicate *p*, and object *o*), and we assume the usual prefixes *rdfs* and *owl* for the RDFS/OWL vocabulary, plus *opal* for the introduced vocabulary and *bro* for the specific examples. For instance, the assertion *t(bro:Invoicing,rdfs:subClassOf,opal:Request)* corresponds to the inclusion axiom *Invoicing* ⊑ *Request*.

A first set of axioms formalize the $\sigma$ relation:

- the domain of $\sigma$ is restricted to elements of a BPS, while it ranges over owl classes;
  $element(El) \leftarrow \sigma(El,C)$
  $t(C,rdf:type,owl:Class) \leftarrow \sigma(El,C)$
- not all the elements of a BPS can be annotated, e.g. gateways are a pure technical construct to model the routing of the control flow, and an ontological counterpart is meaningless;
  $false \leftarrow \sigma(El,C) \wedge gateway(El)$
- the class expression used for the annotation must be satisfiable;
  $false \leftarrow \sigma(El,C) \wedge t(C,rdfs:subClassOf,A) \wedge t(C,rdfs:subClassOf,B) \wedge t(A,owl:disjointWith,B)$
- σ is preserved by the subsumption relation, since the vocabulary introduced in a BP schema is intended to be a specialization of the one introduced in the reference ontology
  $\sigma(El,A) \leftarrow \sigma(El,C) \wedge t(C,rdfs:subClassOf,A)$

The annotation is further constrained as follows, in order to relate the different kinds of BPAL elements, such as item, activities or participants to the very general concepts introduced by the top-level categories of OPAL, such as object, processes and actors:

- An *activity* (both a *task* and *compound activity*) has to be annotated in terms of a sub-class of *opal:Process*, i.e.,
  $$t(C,rdfs:subClassOf,opal:Process) \leftarrow \sigma(ACT,C) \wedge activity(ACT)$$
- An *item* has to be annotated in terms of a sub-class of *opal:Object*, i.e.,
  $$t(C,rdfs:subClassOf,opal:Object) \leftarrow \sigma(IT,C) \wedge item(IT)$$

- A *participant* has to be annotated in terms of a sub-class of *opal:Actor*, i.e.,

$$t(C,rdfs:subClassOf,opal:Actor) \leftarrow \sigma(PAR,C) \wedge participant(PAR)$$

### 5.4 Semantically Enriched BPS

A semantically enriched business process is hence a BPS defined according to BPAL, which elements are linked to concepts defined in a reference ontology through a semantic annotation, as discussed in the previous sections. In order to ease the exchange of meta-data and their reuse, we encode such semantic structure as an RDF model, as exemplified in the above listings, where are presented the examples reported in Table 7, with reference to the ontology in Figure 5

```
<rdf:Description rdf:about="bps:shipment”>
<rdf:type rdf:resource="bpal:Task"/>
<bpal:input rdf:resource="bps:ship_confirmation "/>
<bpal:assigned rdf:resource="bps:logistics_provider"/>
  <bpal:sigma>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="bro:Delivering"/>
        <owl:Class rdf:about="bro:ValueTransfer"/>
        <owl:Restriction>
          <owl:onProperty>
              <owl:ObjectProperty rdf:ID="opal:requires"/>
          </owl:onProperty>
          <owl:allValuesFrom rdf:resource="bro:Good"/>
        </owl:Restriction>
     </owl:intersectionOf>
    </owl:Class>
  </bpal:sigma>
  <bpal:model_ref> http://acme/ACME.xpdl#_123</bpal:model_ref>
</rdf:Description>
<rdf:Description rdf:about="ship_confirmation>
<rdf:type rdf:resource="bpal:Item"/>
  <bpal:sigma>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="bro:DeliveryDetails"/>
        <owl:Restriction>
          <owl:onProperty>
            <owl:inverseOf rdf:resource="opal:content"/>
          </owl:onProperty>
          <owl:someValuesFrom>
            <owl:intersectionOf rdf:parseType="Collection">
              <owl:Class rdf:about="opal:Confirm"/>
              <owl:Restriction>
                <owl:onProperty>
                  <owl:ObjectProperty rdf:ID="opal:receiver"/>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="bro:Supplier"/>
              </owl:Restriction>
              <owl:Restriction>
                <owl:onProperty>
                  <owl:ObjectProperty rdf:ID="opal:sender"/>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="bro:Shipper"/>
              </owl:Restriction>
            </owl:intersectionOf>
          </owl:someValuesFrom>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </bpal:sigma>
```

```
  <bpal:model_ref> http://acme/ACME.xpdl#789</bpal:model_ref>
</rdf:Description>
<rdf:Description rdf:about="logistics_provider”>
  <rdf:type rdf:resource="bpal:Participant"/>
  <bpal:sigma>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="bro:InventoryManager"/>
        <owl:Class rdf:about="bro:Shipper"/>
      </owl:intersectionOf>
    </owl:Class>
  </bpal:sigma>
  <bpal:model_ref> http://acme/ACME.xpdl#_456</bpal:model_ref>
</rdf:Description>
```

In this frame, other meta-data definitions can be easily handled, such as references to WSDL operations describing concrete service implementations or to data types defined in XML files. For instance, the above description also include a reference to the BP fragment (*model_ref*) in the original process schema (e.g., an XPDL file) to keep the link between the annotated BPS fragment and its annotation information, in order to allow other systems to process this piece of information.

# 6 Business Process Knowledge Base

In this section we describe our reasoning approach, based on the knowledge representation framework described so far. In this framework we are able to define a Business Process Knowledge Base (BPKB) as a collection of logical theories that formalize a repository of semantically annotated business process schemas. The interpretation of these theories as logic programs presented in Section 6.1 provides a powerful inference support, at the basis of the reasoning capabilities exemplified in Section 6.2.

## 6.1 Business Process Knowledge Base

In the previous sections we introduced the two main components of the Business Process Knowledge Base, namely *i)* a repository of BPs represented according to BPAL, *ii)* a business reference ontology defined according to OPAL that, together with the semantic annotation, provides a representation of the domain knowledge associated to the BPs. In order to achieve a uniform and formal representation, suited for reasoning on the above structures, we define a BPAL BPS repository ***BPR*** as a First Order Logic theory of the form:

$$\boldsymbol{BPR} = \boldsymbol{B} \cup \boldsymbol{M} \cup \boldsymbol{TR}$$

where ***B*** is a set of BP schemas defined in BPAL, ***M*** and ***TR*** are the BPAL core theories formalizing the meta-model and the behavioral semantics, respectively.

A relevant property of the theory ***BPR*** is that it has a straightforward interpretation as a logic program [6], which can be effectively used for reasoning within a Prolog environment. In this frame, all the properties defined in the aforementioned theories can be used for querying the theory ***BPR***. In particular the predicates defined by the meta-model theory ***M*** and by the BP schemas ***B*** allow us to query the schema level of a BP, verifying properties regarding the elements occurring in it (e.g., *activities, items, gateways*) and their relationships (e.g., *sequence flows*), while ***TR*** allow us to express queries about the behavior of a BP at execution time, i.e., verify properties regarding the execution semantics of a BPS.

For the representation of the business reference ontology we adopt a fragment of OWL, falling within the OWL 2 RL [9] profile. OWL 2 RL is an OWL subset designed for practical implementations using rule-based techniques. The semantics of OWL 2 RL is defined through a partial axiomatization of the OWL 2 RDF-Based Semantics in the form of first-order implications (OWL 2 RL/RDF rules), and constitutes an upward-compatible extension of RDF and RDFS. In the BPKB, the semantic resources encoded in OWL/RDF are represented by means of the ternary predicate *t(s,p,o)*, and reasoning is supported by including a set of clauses encoding the OWL 2 RL/RDF rule-set.

Finally, an Enterprise Knowledge Base is formalized by a logic program ***BPKB***, defined by putting together the theories introduced so far:

$$\boldsymbol{BPKB} = \boldsymbol{BPR} \cup \boldsymbol{BRO} \cup \boldsymbol{\Sigma}$$

where: *i)* ***BPR*** is a BPAL BP repository; *ii)* ***BRO*** is an OPAL Business Reference Ontology, encoded as a set of assertions of the form *t(s,p,o)* and including the OWL 2 RL/RDF rule-set; *iii)* $\boldsymbol{\Sigma}$ is a semantic annotation, including a set of assertions of the form $\sigma$*(BpsEl,Concept)*.

### 6.2 Reasoning on the BPKB

As anticipated, the logic program ***BPKB*** provides a powerful inference support, in order to reason with semantically enriched BPs, taking advantage of the tools developed in the area of logic programming. In particular, it can be used for evaluating conjunctive queries, formulated in the Prolog syntax as clauses of the form:

$$q(\vec{X}) \leftarrow p_1(\overrightarrow{X_1}) \wedge \ldots \wedge p_m(\overrightarrow{X_m}) \wedge \; not\; p_{m+1}(\overrightarrow{X_{m+1}}) \wedge \ldots \wedge not\; p_n(\overrightarrow{X_n})$$

where $p_1, \ldots, p_n$ are predicates defined in ***BPKB***, $q(\overrightarrow{X})$ is the query to be evaluated by the inference engine, $\overrightarrow{X_1}, \ldots, \overrightarrow{X_n}$ are vectors of variables such that every $X$ occurring in $\overrightarrow{X}$ occurs also in some $\overrightarrow{X_i}$.

In this Section we report some examples of reasoning task, regarding the verification of correctness criteria of BP schemas, the query of BP repositories for the retrieval of process fragment and, finally, the compliance of BPs w.r.t. business constraints.

**Verification.** The proposed framework supports the verification of both structural and behavioral properties of BP schemas, as shown in the following queries. By means of the query *q1* the processes that are not well-formed with respect to the meta-model are identified, while *q2* searches for BP schemas which are structured.

$$q1(P) \leftarrow illformed(P)$$
$$q2(P) \leftarrow bp(P,S,E) \wedge str\_sub\_proc(P,S,E)$$

The query *q3* verifies if a given process *p* has at least one correct execution, while *q4* check the presence of a deadlock, i.e., a state where no further action can be executed. Finally, *q5* verify if the process *p* is *unsafe*, i.e., does exist a state where a flow element is enabled twice, leading to multiple concurrent executions of the same piece of work.

$$q3 \leftarrow bp(p,S,E) \wedge reachable\_state([enabled(start,S,p)],[enabled(E,end,p)])$$
$$q4 \leftarrow bp(p,S,E) \wedge reachable\_state([enabled(start,S,p)],I) \wedge not\ result(I,A,Z)$$
$$q5 \leftarrow bp(p,S,E) \wedge reachable\_state([enabled(start,S,p)],I) \wedge result(I,complete(A),Z) \wedge enabled(A,B,p) \subseteq I$$

**Retrieval.** BP repositories can be queried for the retrieval of process (or fragments therein) to ease the re-use and the exchange of knowledge, e.g., for the design of new BPs. As an example we report the query *q6*.

$$q6(P,S,E) \leftarrow input(S,PO) \wedge \sigma(PO,bro{:}PurchaseOrder) \wedge h\_occurs(INV,P,S,E) \wedge \sigma(INV,R1) \wedge t(R1,owl{:}onProperty,opal{:}internal\_process) \wedge t(R1,owl{:}someVauesFrom,bro{:}Invoicing) \wedge h\_occurs(A,P,S,E) \wedge item\_flow(A,C,P) \wedge \sigma(C,R2) \wedge t(R2,owl{:}onProperty,opal{:}content^{-}) \wedge t(R2,owl{:}someVauesFrom,bro{:}Confirm) \wedge wf\_sub\_process(P,S,E)$$

The above query searches for process fragments identified by the triple <*P,S,E*> (where *P* is a process identifier, *S* the element that starts the sub-process, and *E* the element ending the sub-process) such that *i)* start with an activity that processes a *purchase order*, and *ii)* contain an activity that is part of the *invoicing* process, *iii)* contain an activity that process a document constituting the confirmation of a request. Considering the running example of Figure 1, the sub-process starting with *request_shipment* and ending with *g6* is returned.

**Compliance Verification.** In real world applications, the operations of an enterprise are regulated by a set of BPs that are often complemented by specific business rules. Enterprise policies or requirements can be often encoded as constraints that impose the presence (or absence) of a certain activity when some other activity is performed, possibly imposing also ordering conditions, i.e., precedence or subsequence of one another. For example *q7* verifies if the process *p* is compliant with a precedence constraint, that imposing that whenever a delivering is requested, the product availability has been checked before. The query is defined in terms of the negation of the predicate *neg_prec,* that holds if does exist a possible execution which violates the precedence constraint.

$$q7 \leftarrow bp(p,S,E) \wedge not\ neg\_prec(([enabled(start,S,p)],bro{:}VerifyProductAvailability,bro{:}RequestDelivering)$$

$$neg\_prec\ (I,A,B) \leftarrow result(I,start(X),F) \wedge \sigma(X,B)$$

$$neg\_prec\ (I,A,B) \leftarrow result(I,complete(A),Z) \wedge not\ \sigma(X,A) \wedge neg\_precedence\ (Z,A,B)$$

$$neg\_prec\ (I,A,B) \leftarrow result(I,start(A),Z) \wedge neg\_precedence\ (Z,A,B)$$

# 7 Implementation

This section describes the BPAL Platform, a tool implementing the proposed framework. The BPAL Platform provides an BPKB Editor, implemented as an Eclipse Plug-in[7], to assist the user in the definition of an BPKB through a graphical interface, and a Reasoner, based on a Prolog engine, able to operate on the BPKB. A functional view of the application is depicted in Figure 6.

## 7.1 BPAL Reasoner

The reasoning engine for the BPKB has been implemented as a Java application, built on top of a Prolog engine. The main components of the application are shown in Figure 6, and briefly described below.
- *BPMN2BPAL*. This module offers an interface to import BPAL process schemas into the BPKB from BPMN process models. The input BPMN processes can be acquired from both XPDL files (supported, e.g., by TIBCO[8] and Enhydra Shark[9]) and *.bpmn* files, one of the formats supported by the Eclipse SOA Tools Platform[10].

[7] http://www.eclipse.org/
[8] http://developer.tibco.com/business_studio
[9] http://www.together.at/prod/workflow/tws
[10] http://www.eclipse.org/bpmn/

- *OWL2LP*. The OPAL ontology and the semantic annotation are both imported into the BPKB from OWL/RDF files and compiled into a set of ground facts in the triple notation. The parsing of OWL/RDF files is based on the Jena2 toolkit[11].
- *BPKB Manager*. This module interacts with the Prolog engine during the set-up phase, when the BPKB is built. It is responsible for populating the Prolog database with the logic programs encoding the BPKB, i.e.: *i)* the BPAL BP schemas imported through *BPMN2BPAL, ii)* the semantic resources imported through *OWL2LP*, *iii)* the BPAL core theories (*meta-model*, *trace* and *dependency constraints* theories), *iv)* the OWL 2 RL/RDF rule-set to support reasoning over the semantic resources.
- *XSB Prolog.* The application is connected with the XSB system through the Interprolog library[12], a Java/Prolog interface. XSB extends conventional Prolog systems with an operational semantics based on *tabling,* i.e., a mechanism for storing intermediate results and avoiding to prove sub-goals more than once. XSB has several advantages over conventional Prolog systems based on SLDNF-resolution (e.g., SWI-Prolog and SICStus): (i) in many cases XSB is more efficient than conventional systems, (ii) it guarantees the termination of queries to Datalog programs, and (iii) it often avoids to return several times the same answer to a given query.
- *Query Manager*. Having populated the BPKB, inference is essentially performed by posing queries to the XSB engine. The *QueryManager* exposes functionalities to translate QuBPAL queries into Datalog and collect the results, exporting the latter both as text files and as new XPDL files, for its visualization in an external BPMS and their further reuse.

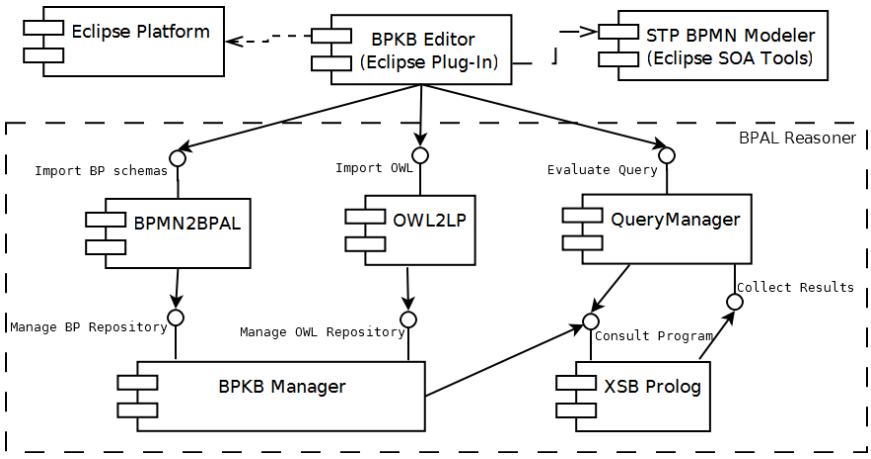


**Fig. 6.** Functional view of the BPAL Platform

## 7.2 BPKB Editor

The BPKB Editor is a prototypical semantic BP modeling environment, that provides a graphical interface to assist the user in the definition of an BPKB and in the interaction with the underlining reasoning engine. It has been implemented as an Eclipse Plug-In, and includes the STP BPMN graphical editor of the Eclipse SOA Tools Platform.

[11] http://jena.sourceforge.net/
[12] http://www.declarativa.com/interprolog

A screen-shot of the GUI is depicted in Figure 7. The left panel (Figure 7.a) provides a tree view of the resources available in the workspace, including BP schemas and ontologies. The central panel (Figure 7.b) is the process editor, provided by the STP BPMN Modeler, comprising an editor and a set of tools to model business process diagrams using the BPMN notation. On the right (Figure 7.c), the ontology browser allows for the visualization of OWL ontologies, published on the Internet or locally stored. The bottom panel (Figure 7.d) is the annotation editor, for the annotation of process elements with respect to the reference ontology. The resulting semantic annotation can be saved and loaded from RDF files. The top-central panel (Figure 7.e) is the query prompt, that provides the users with a direct access to the BPAL reasoner through the query mechanism. Results can be consulted in the 'Result Panel', (Figure 7.f) and, when process fragments are included in the query, the latter can be exported as a new XPDL file for their further re-use.

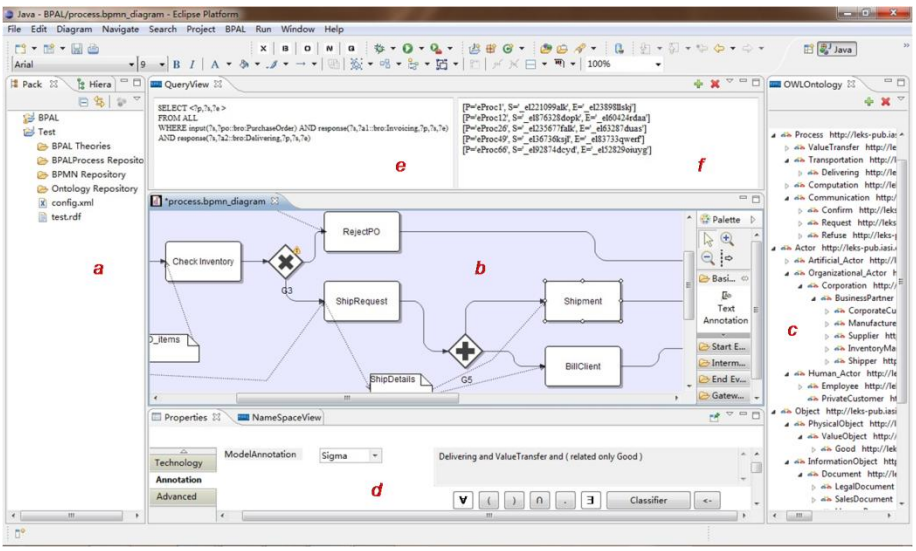


**Fig. 7.** BPKB Editor GUI

## 8 Related Works

In the literature, much attention is given to BP modeling, as its application to the management of complex processes and systems is an important issue in business organizations. It appears increasingly evident that a good support to BP management requires reliable BP modeling methods and tools. Such reliability can be achieved only if the adopted method is based on formal foundations. In this perspective, our work addresses several issues that have been extensively treated in the literature: *i)* formal languages for the specification and analysis of business processes, *ii)* semantic enrichment of process models with domain knowledge, *iii)* querying and verification of process models.

**Process Modeling**. Formal semantics of process modeling languages (e.g., the BPMN case is discussed in [21]) is usually defined in terms of a mapping to Petri nets [22]. Petri nets represent a powerful formal paradigm to support enactment and verification of BPs within a procedural approach. A logic-based approach appears more suited to manipulate, query, retrieve, compose BP diagrams, and verify structural properties at the intensional level. Furthermore, while Petri nets address the operational aspects of BPs, it is not possible to

capture and reason with domain knowledge through such approach exclusively, and further support is needed to address processes at the knowledge level.

A different approach to BP specification is represented by logic-based declarative approaches [23,24]. Here a process is modeled by a set of *constraints* that must be satisfied during execution. These proposals provide a partial representation of a BP that overlooks the procedural view, i.e., the control flow among activities. [23] proposes ConDec, a declarative flow language to define process models that can be represented as conjunction of Linear Temporal Logic formulas. This approach allows the BP designer to enact and verify processes by using model checking techniques. [24] proposes a verification method based on Event Calculus and Abductive Logic Programming. [23,24], are based on rigorous mathematical foundations but they propose a paradigm shift from traditional process modeling approaches that is difficult to be understood and, consequently, to be accepted by business people. Such approaches are mostly intended to complement and extend fully procedural languages rather than replace them, as in the case of Declare[13], which is implemented within the YAWL[14] workflow management system.

Concurrent Transaction Logic (CTR) [25] is a formalism for declarative specification, analysis, and execution of transactional processes, that has been also applied to modeling and reasoning about workflows and services [26]. CTR formulas extend Horn clauses and their standard model-theoretic and operational semantics by introducing three new connectives: *serial conjunction*, which denotes sequential executions, *concurrent conjunction,* which denotes concurrent execution, and *isolation,* which denotes transactional executions. The model-theoretic semantics of CTR formulas is then defined over paths, i.e., sequences of states. Unlike CTR formulas, the *BPKB* can be directly viewed as an executable logic program. Hence any component can be queried by any Prolog system without the need for a special purpose evaluator. This implies that both BP schemas, traces and states are explicitly represented and can be directly analyzed and manipulated, in conjunction with other knowledge representation applications, e.g., computational ontologies. Furthermore, BPAL is derived from graph based modeling languages like BPMN, and queries are written in essentially the same way as process specifications.

**Semantic enrichment.** PSL (Process Specification Language) [27], defines a logic-based neutral representation for manufacturing processes. A PSL ontology is organized into PSL-CORE and a partially ordered set of extensions. The PSL-CORE axiomatizes a set of intuitive semantic primitives (e.g., *activities*, *activity occurrences*, *time points*, and *objects*) enabling the description of the fundamental concepts of processes, while a set of extensions introduce new terminology and its logical formalization. Although PSL is defined in first order logic, which in principle makes behavioral specifications in PSL amenable to automated reasoning, we are not aware of PSL implementations for the verification or analysis of BPs, since it is intended mostly as a language to support the exchange of process information among systems.

Some techniques for semantic annotation first developed in the context of the semantic web have been extended to business process management. Within the SUPER project (http://www.ip-super.org/) several ontologies to model functional, organizational, informational and behavioral perspectives have been developed and in [28] a querying framework based on such ontologies is presented. In [29] SPARQL queries, formulated through a visual language, are evaluated against business processes represented trough a BPMN meta-model ontology annotated with respect to domain ontologies. Other approaches based on meta-model ontologies have been discussed, e.g., [30,31]. Unlike the aforementioned works, where the behavioral aspects are hidden or abstracted away, the execution semantics of a BPAL BPS is formally defined and several verification tasks can also be performed. Hence, the *BPKB* provides a homogeneous framework to effectively reason about and query the workflow structure, the behavioral semantics and the domain knowledge related to the modeled processes.

---

[13] http://www.win.tue.nl/declare/

[14] http://www.yawlfoundation.org/

Relevant work regarding the semantic enrichment of Web Services has been done within the OWL-S [32] and WSMO [33] initiatives. Both approaches make an essential use of ontologies to describe services which are accessible through a Web service interface in order to facilitate the automation of discovering, combining (mainly through AI planning and automated synthesis techniques) and invoking electronic services over the Web. Such approaches strongly differ from ours on scope and purpose and can be considered complementary to our work.

**Querying and verification.** BP-QL [34] is based on an abstract representation of the BPEL (Business Process Execution Language) standard. It is a visual language, formally based on graph grammars, that allows one to query the process *specification* (i.e. the graph representation of a process workflow) of a BPEL process ignoring the run-time semantics of certain constructs such as choice or parallel execution. A graph matching approach is also shared by BPMN-Q [35], a visual language based on BPMN. Such approaches allow the user to query the graph representation of a process workflow in an intuitive way, but they need to be combined with external tools to reason about properties of the behavioral semantics. Our framework not only provides a method for querying the structure of the workflow graph and its behavior, but due to the logic-based representation it also integrates additional reasoning services. In particular, a very relevant advantage of the BPKB is the possibility of formulating queries involving the knowledge represented in domain models formally encoded by means of ontologies. Indeed, queries can be posed in terms of the ontology vocabulary, which offers a "global view" of the processes annotated with it, hence *i)* decoupling queries from specific processes, *ii)* overcoming semantic heterogeneities deriving, e.g., from different terminologies, *iii)* allowing queries to be posed at different generality levels by taking advantage of the semantic relations defined in the ontology, such as *subsumption*.

Finally, program analysis and verification techniques have been applied to the analysis of processes behavior (see [36,37] for a sample). These works are based on the analysis of finite state models through model checking techniques where queries, formulated in some temporal logics, test if the executions of the process satisfy a certain property. Despite the efficiency of these approaches, severe limitations arise when they have to be combined with other techniques in order to verify properties encompassing also other conditions beyond the ordering/ presence/ absence of tasks in the possible execution of the process.

## 9 Conclusions

In this report we presented a framework for the semantic augmentation of process schemas, based on the synergic use of BPAL, a logic-based language adopted to capture the procedural knowledge of a business process, and domain ontologies, to capture the semantics of a business scenario. Both frameworks are seamlessly connected thanks to their grounding in first order logic (in particular, logic programming) and therefore it is possible to apply effective reasoning methods to query the knowledge base encompassing the two. A software platform has been implemented and a preliminary evaluation of the reasoning engine, not reported here, was conducted to prove the feasibility of the approach. In particular, the rule-based implementation of the OWL reasoner and the effective goal-oriented evaluation mechanism of the Prolog engine shown good response time and significant scalability.

We are working to extend the proposed knowledge representation framework in several directions. First of all, we want to increase the expressivity of the approach by supporting a larger number of workflow patterns [1], to ease its adoption in conjunction with commercial tools for BPMS. Then, the query evaluation process can be strongly optimized through more elaborated transformations achieved by exploiting sophisticated program transformation techniques [38]. On an engineering ground, we are exploring the problem of manipulating, merging and aggregating a set of business process fragments in the contexts of BP re-engineering and automatic process composition. Finally, we are working to improve the software platform, in

particular on the user interface and on the level of automation in supporting the semantic annotation of BP schemas.

## 10 References


1. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N.: Modern Business Process Automation: YAWL and its Support Environment. Springer, 2010.
2. OMG: Business Process Model and Notation, http://www.omg.org/spec/BPMN/2.0.
3. Hepp, M., Leymann, F., Domingue, J., Wahler, A., Fensel., D.: Semantic business process management: A vision towards using semantic web services for business process management. In: ICEBE 2005, pp. 535–540. IEEE Computer Society, Los Alamitos (2005).
4. De Nicola, A., Missikoff, M., Proietti, M., Smith, F.: An Open Platform for Business Process Modeling and Verification, International Conference on Database and Expert Systems Applications. Proc. DEXA 2010. LNCS 6261, pp. 66--90, Springer, 2010.
5. Grosof, B. N., Horrocks, I., Volz, R., Decker, S.: Description Logic Programs: Combining Logic Programs with Description Logic, in: Proceedings of the 12th International Conference on World Wide Web, ACM, 2003.
6. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, Berlin, 1987. 2nd Ed.
7. XPDL 2.1 Complete Specification, Oct. 2008, http://www.wfmc.org/xpdl.html.
8. OWL 2: Profiles, http://www.w3.org/TR/owl2-profiles.
9. De Nicola A., Missikoff M., Navigli R.: A software engineering approach to ontology building. Information Systems 2009, 34:258. 10.1016/ j.is.2008.07.002.
10. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P.F. editors. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2003.
11. Przymusinski, T. C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In: Jack Minker (Ed.): Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann 1988, 193-216.
12. Laue, R., and Mendling, J.: The Impact of Structuredness on Error Probability of Process Models. In: UNISCON. Volume 5 of LNBIP. (2008) 585—590.
13. Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C. K., Thatte, S., Yendluri, P., and Yiu, A., editors. Web Services Business Process Execution Language Version 2.0. Committee Draft. WS-BPEL TC OASIS, 2005.
14. Holldobler, S., and Schneeberger, J.: A new deductive approach to planning. *New Generation Computing*, 8:225—244, 1990.
15. Moe Thandar Wynn, David Edmond, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. In ICATPN, LNCS 3536, pages 423–443. Springer, 2005.
16. Ekkart Kindler. On the semantics of EPCs: Resolving the vicious circle. Data Knowl. Eng., 56(1):23–40, 2006.
17. Volzer, H.: A New Semantics for the Inclusive Converging Gateway in Safe Processes. In BPM2010, LNCS 6336, pp. 294-309, Springer, 2010.
18. Guarino, N.: Formal Ontology and Information Systems. In N. Guarino (ed.), Formal Ontology in Information Systems, Proc.of FOIS'98, Trento, Italy, 6-8 June 1998. Amsterdam, IOS Press, pp. 3-15.
19. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison Wesley, USA, 1999.
20. D'Antonio, F., Missikoff, M., Taglino, F.: Formalizing the OPAL eBusiness ontology design patterns with OWL, in: Third Interna-tional Conference on Interoperability for Enterprise Applications and Software, I-ESA 2007.
21. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. 50 (2008) 1281–1294.
22. Reisig, W. and Rozenberg, G. editors: Lectures on Petri Nets I: Basic Models, volume 1491 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1998.
23. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In BPM 2006 Workshops, LNCS 4103. pp. 169-180, 2006.

24. Montali, M., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verification from Declarative Specifications Using Logic Programming. In ICLP 2008, LNCS 5366, pp. 440–454, 2008.
25. Bonner, A. J. and Kifer, M.: Concurrency and Communication in Transaction Logic. In Joint International Conference and Symposium on Logic Programming, 1996.
26. Roman, D. and Kifer, M.: Reasoning about the Behavior of Semantic Web Services with Concurrent Transaction Logic. In VLDB, 2007.
27. Conrad, B. and Gruninger, M.: Psl: A semantic domain for flow models. Software and Systems Modeling, 4(2):209–231, May 2005.
28. Markovic, I. Advanced Querying and Reasoning on Business Process Models. Proc. BIS 2008. LNBIP 7, pp.189--200, Springer, 2008.
29. Di Francescomarino, C., Tonella, P.: Crosscutting Concern Documentation by Visual Query of Business Processes. Business Process Management Workshops 2008.
30. Haller, A. Gaaloul, W., Marmolowski, M.: Towards an XPDL Compliant Process Ontology. SERVICES I 2008, pp.83-86, 2008.
31. Haller, A., Oren, E., Kotinurmi, P.: m3po: An Ontology to Relate Choreographies to Workflow Models. Proc. Int. Conf. on Services Computing, 2006.
32. W3C: OWL-S, Semantic markup for web services. 22 november 2004, http://www.w3.org/Submission/OWL-S/.
33. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontology, 1(1): 77-106, IOS Press, 2005.
34. Beeri, C., Eyal, A., Kamenkovich, S., and Milo, T.: Querying business processes with BP-QL. *Information Systems*. 33, 6 (Sep. 2008), 477-507.
35. Awad, A., Decker, G., Weske, M.: Efficient compliance checking using BPMN-Q and temporal logic. Proc. BPM 2008. LNCS 5240, pp. 326--341. Springer, 2008.
36. Fu, X., Bultan, T., and Su, J.: Analysis of interacting BPEL web services. Proc. Int. World Wide Web Conf., pp. 621--630, ACM Press, 2004.
37. Liu, Y., Muller, S., Xu, K.: A static compliance-checking framework for business process models. *IBM Systems Journal* 46 (2007) 335--361.
38. Pettorossi, A. and Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. Journal of Logic Programming 1994: 19, 20: 261-320.