Reasoning on Data-Aware Business Processes with Constraint Logic

Maurizio Proietti and Fabrizio Smith

National Research Council, IASI "Antonio Ruberti" - Via dei Taurini 19, 00185 Roma, Italy
{maurizio.proietti, fabrizio.smith}@iasi.cnr.it

Abstract. We propose a framework grounded in Constraint Logic Programming for representing and reasoning about business processes from both the workflow and data perspective. In particular, our goal is twofold: (1) define a logical language and a formal semantics for process models where data object manipulation and interactions with an underlying database are explicitly represented; (2) provide an effective inference mechanism that supports the combination of reasoning services dealing with process behavior and data properties. To this end we define a rule-based process representation coping with a relevant fragment of the popular BPMN modeling notation, extended with annotations that model data manipulation. The behavioral semantics of a process is defined as a state transition system by following an approach similar to the Fluent Calculus, and allows us to specify state change in terms of preconditions and effects of the enactment of activities. Our framework provides a wide range of reasoning services, which can be performed by using standard Constraint Logic Programming inference engines.

Keywords: Business Process, Constraints, Logic Programming, Analysis, Verification

1 Introduction

The penetration of Business Process (BP) Management solutions into production realities is constantly growing, due to its potential for an effective support to enterprise actors and business stakeholders along the entire BP life-cycle. In this frame, modeling languages such as BPMN [1] are largely adopted by the stakeholders (designers, analysts, business men) to develop conceptual models to be used for the design and reengineering of BPs. One of the main advantages of having a machine-processable representation of BPs available is that it enables the automation of tasks dealing with process analysis, simulation and verification.

However, standard process-centric approaches focus on the procedural representation of a BP as a workflow graph that specifies the planned order of operations, while the interactions of individual operations with the underlying *data* layer is often left implicit or abstracted away. Indeed, the automated analysis issue is addressed in the workflow community mainly from a control flow perspective (see, for instance, the notion of soundness [2]), and most of the tools today available aim at verifying whether the behavior of the modeled system enforces requirements specified without considering the data perspective.

In order to provide an integrated account of the workflow and data modeling, several approaches have been proposed both in industrial realities (e.g., [3, 4]), as well as in the database (e.g., [5]) and workflow (e.g., [6]) research communities. A *data-aware* BP representation explicitly models the manipulation of data objects operated by individual tasks and their interactions with databases, with the aim of enabling the automated analysis of behavioral properties of the resulting system.

In this paper we propose a logic-based framework for representing and reasoning about data-aware BP models, where the workflow perspective, specified according to BPMN, is enriched by annotations defining *preconditions* and *effects* of individual process elements in terms of *data objects*, used to store information that is read and modified by the process enactment. We also consider the existence of an underlying database, which can be queried during the enactment, retrieving values to be used for data object manipulation. The behavioral semantics of a process is defined as a state transition system by following an approach derived from the *Fluent Calculus* [7], and by integrating into the framework a *symbolic* representation of data object values, given in terms of arithmetic constraints over the real numbers. The proposed rule-based formalization supports a relevant fragment of BPMN in addition to expressive data modeling, and its grounding into Constraint Logic Programming (CLP) [8] provides a uniform and formal framework that enables automated reasoning.

In this work we do *not* propose yet another business process modeling language, but we assume a pragmatic perspective aiming at supporting process-related knowledge expressed by means of de-facto standards for BP modeling, like BPMN. To the best of our knowledge, this is one of the first attempts to provide a formal execution semantics for expressive BPMN workflows in the presence of data and arithmetic constraints. Notably, the CLP formalization directly provides an executable semantics, which enables the implementation of analysis and verification tasks relying on established automated reasoning methods and tools. Due to the presence of data, the state space of the modeled systems is potentially infinite and most verification problems are undecidable. However, the symbolic representation of data values by means of constraints achieves the termination of a number of reasoning services in many cases of practical relevance also in the presence of an infinite state space.

The paper is organized as follows. After presenting a motivating scenario and introducing the modeling framework in Section 2, we provide a formal account of the behavioral semantics in Section 3. In Section 4 we show how automated reasoning methods developed in the field of CLP can be directly applied to perform analysis and verification tasks. In Section 5 we present a proof-of-concept prototype, and, finally, in the concluding section we give a critical discussion of our approach, along with directions for future work.

2 Modeling Data-Aware Business Processes

A data-aware Business Process Schema (*DAPS*) is an activity workflow model, where each task can additionally operate on *data objects* used to store information that is read and modified by process enactment. In our approach, data objects are essentially regarded as variables, and hence at any time during execution there is a single instance

of a given data object that may be read or (over-)written by some activity. We consider two main types of relationships between activities and data objects. Firstly, the enactment of an activity may be guarded by a condition involving a number of data objects. Secondly, the enactment of an activity can modify the value of a data object, hence producing a new value, possibly related to other data objects' values by an *arithmetic constraint* over the real numbers $\mathbb{R}$. We also consider the presence of a database (*DB*) that can be queried during process enactment, hence retrieving values to be used for data object manipulation. For the scope of this work, we assume that *DB* cannot be updated by activity executions. Furthermore, we assume to deal with BPs whose state space, considering the control flow only, is finite. However, since the data objects can assume infinitely many values, the state space of the overall system is in general infinite.

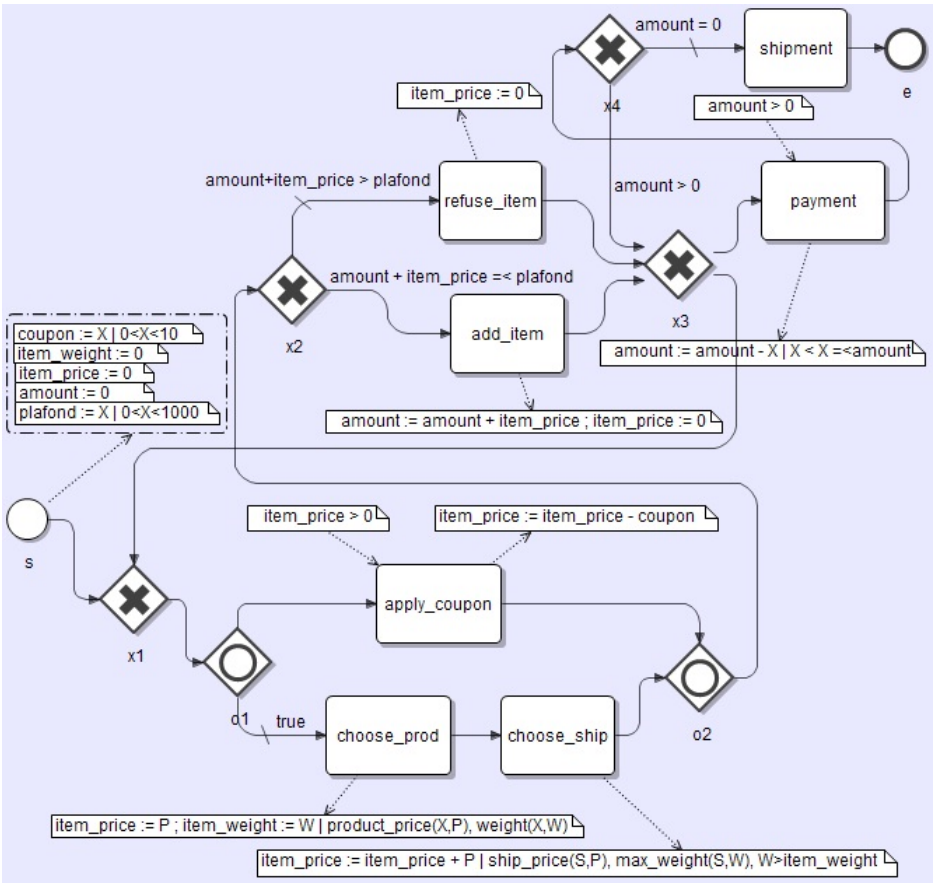

Fig. 1: Example DAPS for an eProcurement Scenario

For the representation of the workflow-related perspective, we mainly refer to the BPMN standard [1], which will be used throughout as a reference notation. We show how a DAPS is specified by means of the example depicted in Figure 1, which deals with the handling of a purchase order in an eProcurement scenario.

A DAPS consists of a set of *flow elements* and *relations* between them, and it is associated with a unique *start event* (e.g., s) and a unique *end event* (e.g., e), which are flow elements that represent the entry point and the exit point, respectively, of the process. An *activity* is a flow element that represents a unit of work performed within the process. It can be modeled as a *task*, representing an atomic activity no further decomposable (e.g., *choose_prod*), or as a *compound activity*, representing the execution of a sub-processes (not exemplified here). The sequencing of flow elements is specified by the *sequence flow* relation (corresponding to solid arrows), and the *branching/merging* of the control flow is specified by using three types of *gateways*: *exclusive* (XOR, e.g., $x1$), *inclusive* (OR, e.g., $o1$), and *parallel* (AND, not exemplified here).

The *item flow* relation (corresponding to dotted arrows) specifies that a flow element uses as *input* or manipulates as *output* particular data objects. In our setting, the input and output of a flow element can be enriched by declarative descriptions of *preconditions* and *effects*, respectively, formulated in terms of arithmetic constraints, value assignments (denoted by :=), and database *queries* (following the $|$ symbol). For instance, the task *apply_coupon* requires (precondition) a value of *item_price* greater than 0, while upon its execution (effect), the value of *item_price* is decreased by the amount of *coupon*. More complex effects can be specified, as in the case of *choose_prod*, where the values assigned to *item_price* and *item_weight* are retrieved by performing the conjunctive database query $product\_price(X,P), weight(X,W)$. Uppercase letters denote *variables*, representing values not known at design-time that are introduced during the enactment, e.g., retrieved by database queries or produced after interactions with external systems or users. The values that variables are allowed to assume at run-time can be characterized in terms of arithmetic constraints, as exemplified by the *payment* effect where, due to the constraint $0 < X \leq amount$, the possible values of X after each execution of *payment* range from 0 to the current value of *amount* (representing any admissible paid amount). Similarly to activity preconditions, *guards* can be attached to outgoing sequence flows of inclusive and exclusive gateways, as in the case of $o1$ and $x2$. Whenever a guard is not defined, a non-deterministic behavior is assumed.

The depicted BP is started by the user log-in, which triggers the execution of the start event leading to the initialization of the involved data objects. The user can select a number of products, by choosing for each of them a shipment compatible with the item weight and optionally applying a coupon, which decreases the item price. When the amount due exceeds the plafond associated with the user, a selected product cannot be added to the order and it is refused. In order to proceed with the shipment, the full amount due has to be paid, possibly through several subsequent payments.

3 Formal Semantics

Now we present a formal definition of the behavioral semantics, or *enactment*, of a DAPS, by following an approach derived from the *Fluent Calculus*, a well-known rule-based calculus for action and change (see [7] for an introduction), which is formalized in Logic Programming (LP). In [9] we have proposed a specialized version of the Fluent Calculus, developed to specifically deal with BPs. Here, in order to cope with the data perspective, our formalization is enhanced by using Constraint Logic Programming

(CLP) [8], which extends LP with constraints over specific domains and, in particular, over the domain of the real numbers with the usual arithmetic operations.

3.1 Constraint Logic Programming

We will consider CLP programs with constraints over the set $\mathbb{R}$ of the real numbers. We will denote variables by upper case letters $X_1, X_2, \ldots$, while we will denote constants, predicate and function symbols by lower case letters, $a, p, f, \ldots$ Constraints are inductively defined as follows. An *atomic constraint* is either a formula of the form $p_1 \geq p_2$ or a formula of the form $p_1 > p_2$, where p_1 and p_2 are polynomials with real variables. We will also use the equality '$=$' and the inequalities '$\leq$' and '$<$' defined in terms of '$\geq$' and '$>$' as usual. A *constraint* is either *true*, or *false*, or an atomic constraint, or a *conjunction* of constraints.

An *atom* is an atomic formula of the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol not in $\{\geq, >\}$ and $t_1, \ldots, t_m$, with $m \geq 0$, are terms. A *literal* is either an atom A or a negated atom $\neg A$. A *goal* is a (possibly empty) conjunction of atoms. A *constrained goal* $c \wedge G$ is a conjunction of a constraint c and a goal G. A CLP *program* is a finite set of *rules* of the form $A \leftarrow c \wedge G$ (to be understood as "A if c and G"), where A is an atom and $c \wedge G$ is a constrained goal. Given a rule $A \leftarrow c \wedge G$, A is the *head* of the rule and $c \wedge G$ is the *body* of the rule. A rule with empty body is called a *fact*. A term or a formula is *ground* if no variable occurs in it.

Let $T_{\mathbb{R}}$ denote the set of ground terms built from $\mathbb{R}$ and from the set of function symbols in the language. An $\mathbb{R}$-*interpretation* is an interpretation that: (i) has universe $T_{\mathbb{R}}$, (ii) assigns to $+, \times, >, \geq$ the usual meaning in $\mathbb{R}$, and (iii) is the standard *Herbrand interpretation* [10] for function and predicate symbols different from $+, \times, >, \geq$. We can identify an $\mathbb{R}$-interpretation I with the set of ground atoms (with arguments in $T_{\mathbb{R}}$) which are true in I. We write $\mathbb{R} \models \varphi$ if φ is true in every $\mathbb{R}$-interpretation. A constraint c is *satisfiable* if $\mathbb{R} \models \exists X_1, \ldots, X_n.c$, where $X_1, \ldots, X_n$ are all variables occurring in c. A constraint c *entails* a constraint d, denoted $c \sqsubseteq d$, if $\mathbb{R} \models \forall X_1, \ldots, X_n.c \rightarrow d$, where $X_1, \ldots, X_n$ are all variables occurring in c or d. An $\mathbb{R}$-*model* of a CLP program P is an $\mathbb{R}$-interpretation that makes true every rule of P. Every CLP program P has a *least* (with respect to set inclusion) $\mathbb{R}$-model, denoted $M(P)$.

We consider the standard operational semantics of CLP programs based on *resolution* extended with *constraint solving* [8]. For reasons of simplicity we assume that no negated atom appears in the body of a CLP rule. However, we consider negated atoms in queries. A *query* has the form $\leftarrow c \wedge C$, where c is a constraint and C is a conjunction of literals. A query $\leftarrow c \wedge C$ *succeeds* if it is possible to derive from it, possibly in many steps, a query of the form $\leftarrow c' \wedge true$, where c' is a satisfiable constraint and *true* is the empty conjunction of literals. The constraint c' is also called an *answer* to the query $\leftarrow c \wedge C$. As usual in CLP systems, we assume the left-to-right selection strategy of literals during resolution. A query $\leftarrow c \wedge C$ *(finitely) fails* if the set of derivations from it is finite and no query of the form $\leftarrow c' \wedge true$, where c' is satisfiable, is derivable. The operational semantics is sound with respect to the least model semantics in the sense that, if a query $\leftarrow Q$ succeeds with answer c' then $M(P) \models \forall X_1, \ldots, X_n.c' \rightarrow Q$, where $X_1, \ldots, X_n$ are the variables occurring in $c' \rightarrow Q$. If $\leftarrow Q$ fails then $M(P) \models \forall X_1, \ldots, X_n.\neg Q$, where $X_1, \ldots, X_n$ are the variables occurring in Q.

3.2 Data-Aware BP Schema Representation

A DAPS is formally represented by a triple $\langle WF, DC, DBS \rangle$, where *WF* is a workflow specification, *DC* (*data constraints*) is a specification of the *preconditions* and *effects* of activities on data objects, and *DBS* is a database schema.

The workflow specification *WF* is a set of ground facts of the form $p(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n$ are constants denoting individual flow elements and p is a predicate symbol representing a BPMN construct (e.g., activity, gateway, sequence flow). For instance, the formal specification of the eProcurement BP in Figure 1 will contain facts such as: $bp(eProc, s, e)$, stating that *eProc* is a process starting with s and ending with e; $task(choose\_prod)$, stating that *choose_prod* is an atomic activity within the workflow; $exc\_branch(x2)$, stating that $x2$ is an exclusive branch point, i.e., a decision point; $seq(choose\_prod, choose\_ship, eProc)$, stating that a sequence flow relation is defined between *choose_prod* and *choose_ship* in *eProc*.

The data constraints *DC* specify the way data objects are manipulated during process enactment, by means of the following relations:

Precondition: $pre(A, C, Proc)$, which specifies an *enabling condition* C that the data objects must satisfy to enable the execution of an activity A in the process *Proc*;

Effect: $eff(A, E, Proc)$, which specifies the *effect expression* E describing the effect on the data objects of executing A in the process *Proc*;

Guard: $guard(C, B, Y, Proc)$, which specifies a *conditional sequence flow* used to select the set of successors of decision points, where the *enabling condition* C must hold in order to enable the flow element Y, successor of B in the process *Proc*.

Enabling conditions and effect expressions are formally defined as follows. Let *DO* denote the set of data objects occurring in the DAPS. An *arithmetic expression* is an expression constructed from *DO*, (CLP) variables, real numbers, $+$, and $\times$. A *constraint expression* is an expression of the form a_1 *rel* a_2, where a_1, a_2 are arithmetic expressions and $rel \in \{\geq, >\}$ (i.e., a constraint expression is an atomic constraint on data objects and CLP variables). A *db-query* is an atom whose predicate is defined in the database schema DBS. A *data update* is an expression of the form $o := a$, where $o \in DO$ and a is an arithmetic expression. An *enabling condition* is the conjunction of $n \geq 0$ constraint expressions. An *update condition* is either a constraint expression or a db-query. An *effect expression* is a pair *data-updates* [] *conds*, where *data-updates* is a sequence of data updates and *conds* is a conjunction of update conditions.

Returning to the eProcurement example of Section 2, a precondition associated with the *payment* activity is:

$$pre(payment, [amount > 0]), eProc)$$

meaning that the task *payment* can be executed only if the data object *amount* has a positive value. In the above example and in the sequel, both sequences and conjunctions appearing as predicate arguments are represented using lists. The specification of the effect associated with the *choose_ship* activity is:

$$eff(choose\_ship, [item\_price := item\_price + P] \; [] \\ [ship\_price(S, P), max\_weight(S, W), W > item\_weight], eProc)$$

meaning that the effect of the execution of the task *choose_ship* is that the value of the data object *item_price* is incremented by a quantity P, where P is the price of a shipment type S and the value of *item_weight* is below the maximum weight allowed for S.

The specification of the guard associated with the flow from $x2$ to *refuse_item* is:

$guard([amount + item\_price > plafond], x2, refuse\_item, eProc)$

meaning that the control flow can proceed from the gateway $x2$ to the *refuse_item* task only if the sum of the values of *amount* and *item_price* exceeds the value of *plafond*.

The database schema *DBS* consists of a set of predicate symbols (with arity) representing the relations stored in the database, together with a set of formulas of the form $p(X_1,\ldots,X_n) \rightarrow c$, where p is a predicate symbol in the schema and c is a constraint whose variables are among $X_1,\ldots,X_n$, representing *integrity constraints* (for simplicity we do not consider more complex integrity constraints). A *database instance* of *DBS* is a finite set of ground facts $p(a_1,\ldots,a_n)$ that satisfies all formulas in *DBS*.

3.3 Behavioral Semantics of Data-Aware Processes

Similarly to the Fluent Calculus, we represent the state of the world as a set of *fluents*, i.e., terms denoting atomic properties that hold at a given instant of time. The execution of a flow element may cause a change of state, i.e., an update of the collection of fluents associated with it. In particular, a change of state can be determined by the effects of activities specified by *DC*. A fluent is represented by an expression of the form $f(a_1,\ldots,a_n)$, where f is a fluent symbol and $a_1,\ldots,a_n$ are constants or variables. We take a closed-world interpretation of states, that is, we assume that a fluent F holds in a state S iff $F \in S$. Our set-based representation of states relies on the assumption that the DAPS is *safe*, that is, during its enactment there are no concurrent executions of the same flow element [2]. This assumption enforces that, in absence of data objects, the set of states reachable by a given DAPS is finite.

We will consider the following three kinds of fluents:

- $cf(E_1,E_2,Proc)$, which means that the flow element E_1 has been executed and the successor flow element E_2 is waiting for execution, during the enactment of the process *Proc* (*cf* stands for *control flow*);
- $en(A,Proc)$, which means that the activity A is being executed during the enactment of the process *Proc* (*en* stands for *enacting*).
- $val(O,V)$, which means that the data object $O \in DO$ has value V.

The truth value of a fluent or update condition depends on the state where it is evaluated. For this reason we introduce the satisfaction relation $holds(C,S)$, which holds if C is true in the state S, where C is either a fluent or a conjunction of update conditions.

1. $holds([\,],S)$
2. $holds([C|Cs],S) \leftarrow holds(C,S) \wedge holds(Cs,S)$
3. $holds(p(X_1,\ldots,X_n),S) \leftarrow p(X_1,\ldots,X_n)$ for every p declared in *DBS*
4. $holds(F,S) \leftarrow fluent(F) \wedge F \in S$
5. $holds(val(X,V),S) \leftarrow constant(X) \wedge V = X$
6. $holds(val(X,V),S) \leftarrow variable(X) \wedge V = X$
7. $holds(A_1 > A_2,S) \leftarrow V_1 > V_2 \wedge holds(val(A_1,V_1),S) \wedge holds(val(A_2,V_2),S)$
8. $holds(val(A_1 + A_2,V),S) \leftarrow V = V_1 + V_2 \wedge holds(val(A_1,V_1),S) \wedge holds(val(A_2,V_2),S)$

Recall that in this paper we assume that the database does not change during process enactment, and hence in Rule 3 the evaluation of the db-query $p(X_1,\ldots,X_n)$ does not depend on the state S. Rule 4 has one instance for each kind of fluents, and a particular instance is the following rule for retrieving the value of a data object O in a state S: $holds(val(O,V),S) \leftarrow val(O,V) \in S$. Rules 5 and 6 express the fact that the value of logical variables and constants is independent of the state. Rules 7 and 8 are needed to define the value of constraint expressions by structural induction. We have omitted the rules for $\geq$ and $\times$, which are similar to rules 7 and 8, respectively.

The change of state determined by the execution of an action will be formalized by a relation $result(S_1,A,S_2)$, which holds if action A can be executed in state S_1 leading to state S_2. For the definition of $result(S_1,A,S_2)$, we assume that an instance of the database schema *DBS* is provided. We also assume that the execution of an activity has a beginning and a completion (although we do not associate a *duration* with activity execution), while the other flow elements execute instantaneously. Thus, we will consider two kinds of actions: $begin(A)$ which starts the execution of an activity A, and $complete(E)$, which represents the completion of the execution of a flow element E (possibly, an activity). The following auxiliary predicate will be used: $update(S_1,T,U,S_2)$, which holds if $S_2 = (S_1 - T) \cup U$, where S_1,T,U, and S_2 are sets of fluents.

Let us now present some of the rules that define the behavioral semantics of a DAPS. The state change determined by the execution of a task is defined by the following two rules, corresponding to the start and the completion of the task, respectively:

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

$$result(S_1,complete(A),S_2) \leftarrow task(A) \wedge holds(en(A,P),S_1) \wedge seq(A,Y,P) \\ \wedge eff(A,DU \,[\!]\, C,P) \wedge holds(C,S) \wedge apply(DU,S_1,S') \\ \wedge update(S',\{en(A,P)\},\{cf(A,Y,P)\},S_2)$$

The first rule states that the execution of task A is started if the control flow has reached it ($holds(cf(X,A,P),S_1)$) and the enabling conditions associated with it hold in the current state ($pre(A,C,P) \wedge holds(C,S_1)$). The successor state is obtained by asserting that the process is enacting A ($update(S_1,\{cf(X,A,P)\},\{en(A,P)\},S_2)$). The second rule states that the execution of task A can be completed if the update conditions associated with A hold in the current state ($eff(A,DU \,[\!]\, C,P) \wedge holds(C,S)$). The successor state is obtained by applying the sequence DU of data updates, hence updating the values of the data objects ($apply(DU,S_1,S')$), and moving the control flow to the next flow element Y ($update(S',\{en(A,P)\},\{cf(A,Y,P)\},S_2)$). The relation $apply(DU,S_1,S')$, meaning that state S' is obtained from state S_1 by performing the sequence DU of data updates is defined as follows:

$$apply([\,],S,S).$$

$$apply([DU|DUs],S,T) \leftarrow apply(DU,S,S') \wedge apply(DUs,S',T)$$

$$apply(O := A,S,S') \leftarrow holds(val(A,V),S) \wedge update(S,\{val(O,X)\},\{val(O,V),S'\})$$

The following two rules formalize the state changes determined by the execution of an exclusive branch (with a guard associated with an outgoing flow) and an exclusive merge, respectively.

$$result(S_1,complete(B),S_2) \leftarrow exc\_branch(B) \wedge holds(cf(X,B,P),S_1) \wedge seq(B,Y,P) \\ \wedge guard(C,B,Y,P) \wedge holds(C,S_1) \wedge update(S_1,\{cf(X,B,P)\},\{cf(B,Y,P)\},S_2)$$

$$result(S_1, complete(M), S_2) \leftarrow exc\_merge(M) \wedge holds(cf(A,M,P), S_1) \wedge seq(M,Y,P) \\ \wedge update(S_1, \{cf(A,M,P)\}, \{cf(M,Y,P)\}, S_2)$$

Note that, in particular, in order to proceed from the exclusive branch B to the next flow element Y, the guard C associated with the flow from B to Y should hold in the current state ($guard(C,B,Y,P) \wedge holds(C,S_1)$).

The behavioral semantics of other flow elements, e.g., parallel or inclusive gateways, can be formalized by rules defined in a similar style. For lack of space we omit those rules and we refer to [9] for more details in the simpler case where data object manipulation (which is the main contribution of this paper) is not considered[1].

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

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

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

4 Reasoning Services

The formal semantics of data-aware BP schemas introduced in Section 3 is the basis for developing automated reasoning techniques for the analysis and verification of business processes that manipulate data objects. A major point is that our formal semantics is a CLP program, and hence we can directly apply automated reasoning methods and tools developed in the field of Constrained Logic Programming to perform analysis and verification tasks. Indeed, by using standard CLP systems we are able to provide a framework that supports several reasoning services and, in particular, in this section we will demonstrate some of them and their use for analyzing process *enactment*, for *testing* process executions, and for *verifying* behavioral properties.

Given a DAPS $\langle WF, DC, DBS \rangle$ and a database instance D of the schema DBS, let $\mathcal{KB}$ be the CLP program consisting of: (1) the ground facts WF specifying the workflow, (2) the *pre*, *eff*, and *guard* facts in DC specifying the enabling conditions, effects, and guards associated with the workflow, (3) the ground facts in D, and (4) the rules (introduced in Section 3) that define the behavioral semantics of the DAPS.

Reasoning services will be realized by evaluating queries to the CLP program $\mathcal{KB}$. One major advantage of our approach is that query evaluation relies on a *symbolic* representation of states, which often avoids the actual exploration of the whole, in general infinite, state space, by covering that space by means of a finite set of constraints. More specifically, a symbolic state is represented as a set of the form:

$\{f_1, \ldots, f_k, val(o_1,V_1), \ldots, val(o_m,V_m)\}$ satisfying a constraint c on $V_1, \ldots, V_m$.

In the above symbolic state $f_1, \ldots, f_k$ are ground *cf* or *en* fluents and $val(o_1,V_1), \ldots, val(o_m,V_m)$ are fluents that associate data objects $o_1, \ldots, o_m$ with their values $V_1, \ldots, V_m$, respectively. Thus, a symbolic state represents the, possibly infinite, set of concrete

[1] The semantics presented in [9] supports the definition of unstructured workflows with arbitrary cycles, exceptional flows, and inclusive merge points, under the safeness assumption [2].

states that satisfy the given constraint. We say that a symbolic state S with associated constraint c is *subsumed* by another symbolic state T with associated constraint d, if S is equal to T, modulo variable renaming, and $c \sqsubseteq d$. We can often reduce the state space by avoiding to consider symbolic states that are *subsumed* by previously visited ones.

4.1 Enactment

We model the enactment of a DAPS as an *execution trace* (corresponding to a *plan* in the Fluent Calculus), i.e., a sequence of actions of the form $[act(a_1),\ldots,act(a_n)]$ where *act* is either *begin* or *complete*.

The predicate $trace(S_1,T,S_2,N)$ defined below holds if T is a sequence of actions of maximum length N that leads from state S_1 to state S_2:

$trace(S_1,[\,],S_2,N) \leftarrow N = 0 \wedge S_1 = S_2$
$trace(S_1,[A|T],S_2,N) \leftarrow N > 0 \wedge N_1 = N-1 \wedge result(S_1,A,U) \wedge trace(U,T,S_2,N_1).$

In the following we use the abbreviation s_{0Proc} to denote the *initial* state of a process *Proc*, where the start event of *Proc* is enabled to fire. Furthermore, we introduce the following rule to characterize the set of *final* states, where the end event of *Proc* is enabled to fire

$holds(final(Proc),S) \leftarrow bp(Proc,E_{start},E_{end}) \wedge holds(en(E_{end},Proc),S).$

Our framework provides two services for analyzing process enactment, namely *trace compliance* and *simulation*.

(1) Trace compliance is the task of verifying whether an execution trace of a process is correct with respect to a given DAPS specification. Execution traces are commonly stored by BP management systems as process logs, representing the evolution of the process instances that have been enacted. Formally, a *correct trace* T of length N of a process *Proc* is a trace that leads from the initial state to the final state of *Proc*, that is:

$ctrace(T,Proc,N) \leftarrow trace(s_{0Proc},T,S_f,N) \wedge holds(final(Proc),S_f)$

The compliance of a trace with respect to a given DAPS can then be verified by evaluating a query of the form $\leftarrow ctrace(t,p,n)$ with respect to program $\mathcal{KB}$, where t is a ground list of length at most n and p is a process name. It is easy to see that such query terminates for every ground t, as the length of the second argument of the *trace* predicate decreases at each recursive call. An example of correct trace related to our running example is reported below.

```
[comp(s),comp(x1),comp(o1),beg(choose_prod),comp(choose_prod),
beg(choose_ship),comp(choose_ship), comp(o2),comp(x2),beg(add_item),
comp(add_item),comp(x3),beg(payment),comp(payment),
comp(x4),beg(shipment),comp(shipment),comp(e)]
```

The trace is guaranteed to be compliant also with respect to the data constraints associated with the DAPS, even if the information about the actual values of the data objects is not explicitly represented. A more complex representation of traces that also includes information about the data object values can easily be defined, but we omit it for reasons of simplicity.

(2) Simulation is the task of *generating* execution traces that represent possible process enactments. In order to analyze the dynamic behavior of the process in various situations, the process designer can analyze test cases where: (i) data objects are initialized to ground values in $\mathbb{R}$, and (ii) a subset of the database instance D is selected. A query of the form $\leftarrow trace(s_{0Proc}, T, S, n)$, where T is a free variable, can be used to generate the execution traces T of length not larger than n. The query terminates for every fixed integer n, as the last argument of *trace* decreases at each recursive call, and for each successful derivation from the query, the unification mechanism employed by CLP will bind T to a ground list of actions. Similarly, to the case of trace compliance, we can easily extend our definitions so as to generate traces that also contain explicit information about data values.

4.2 Symbolic Testing

Simulation is performed by selecting a finite set of test cases, that is, by fixing values for initializing the data objects and taking into consideration a specific database instance. However, the generation of test cases is not always straightforward. The mechanisms of symbolic computation provided by CLP (notably, unification and constraint solving) enable us to generate execution traces by only specifying constraints that those values are required to satisfy. Thus, we can initialize a data object to a value in a range, rather than to a concrete value. Furthermore, we can exploit the integrity constraints in the database schema *DBS* to perform a symbolic evaluation of a *trace* query without considering a fixed database instance.

For instance, in our eProcurement example, we can evaluate a *trace* query by initializing the data object *coupon* to a value X with $0 < X < 10$, as specified in Figure 1. Similarly, we can initialize *plafond* to a value X with $0 < X < 1000$. Furthermore, suppose that in our running example the integrity constraints relative to *product_price* and *ship_price* are:

$product\_price(X,P) \rightarrow P > 5 \wedge P < 100$
$ship\_price(X,P) \rightarrow P > 0 \wedge P < 15.$

Then we replace the database instance D in $\mathcal{KB}$ by the inverse implications of these integrity constraints, that is, by the rules:

$product\_price(X,P) \leftarrow P > 5 \wedge P < 100$
$ship\_price(X,P) \leftarrow P > 0 \wedge P < 15.$

In general, for performing a symbolic testing task, we replace the ground facts that constitute the database instance D in $\mathcal{KB}$, by the inverse implications $p(X_1,\ldots,X_n) \leftarrow c$ of all integrity constraints $p(X_1,\ldots,X_n) \rightarrow c$ specified by *DBS*, hence deriving a new CLP program $\mathcal{KB}'$. Due to the least model semantics of CLP, we will have that $M(\mathcal{KB}') \models p(X_1,\ldots,X_n) \leftrightarrow c_1 \wedge \ldots \wedge c_k$, where $c_1,\ldots,c_k$ are the constraints implied by $p(X_1,\ldots,X_n)$ in *DBS*. $\mathcal{KB}'$ is an over-approximation of $\mathcal{KB}$, i.e., $M(\mathcal{KB}) \subseteq M(\mathcal{KB}')$. Then we evaluate a query of the form $\leftarrow trace(s_{0Proc}, T, S, n)$, for a given integer n. This query always terminates and, if it succeeds and returns an answer $T = t$, then there exists a database instance of *DBS* such that the DAPS $\langle WF, DC, DBS\rangle$ has t as a possible execution trace. The converse in not necessarily true, i.e., there may exist database instances that do not generate the trace t.

$\mathcal{KB}'$ can be used to test reachability properties of the DAPS. For instance, the reachability of a *deadlock state* in n steps can be verified trough a query of the form:

$\leftarrow trace(s_{0Proc}, T, S_d, n) \wedge \neg r(S_d, S_n)$

If the query succeeds, then the DAPS can reach, for some database instance, a potential deadlock state S_d and T is bound to a trace that leads to that state. If the query fails then, for any database instance satisfying the given database schema *DBS*, no deadlock is reachable in at most n steps.

In our example, two potential deadlock states are reachable by taking $n = 20$, both occurring when the control flow reaches *payment* (i.e., $holds(en(payment, eProc), S_d)$). In the first case the potential deadlock is due to a negative value of *amount* caused by a *coupon* value higher than the price of the product chosen by *choose_prod*, which prevents the execution of *payment*. Indeed, the following constraint associated with S_d is computed as an answer to the above query:

$V_{amount} < 0 \ \wedge \ V_{item\_price} < 0 \ \wedge \ V_{coupon} > 5$

where V_o is the logical variable associated to the data object o in the state S_d through the fluent *val* (i.e., $holds(val(o, V_o), S_d)$). In the second case we have that the following constraint associated with S_d is computed as another answer to the above query:

$V_{amount} = 0 \ \wedge \ V_{item\_price} < V_{plafond} < 115$

Here the potential deadlock is due to a value of *item_price* that exceeds the plafond granted to the user. For this reason, no item is added to the order, causing a zero value *amount* and preventing the execution of *payment*.

In order to prevent the above potential deadlocks it is sufficient to fix different ranges for the data objects *plafond* and *coupon* that make the two constraints unsatisfiable, e.g., $0 < coupon \leq 5$ and $115 \leq plafond < 1000$. In this way, the lowest plafond always covers the purchase of at least one product, and the highest coupon cannot exceed the price of any product.

4.3 Verification

Verification aims at checking whether a temporal property holds for all enactments of a given DAPS. Unlike the trace conformance, simulation, and symbolic testing tasks, for verification we do not assume a bounded trace length. Thus, due to the presence of data objects with values in $\mathbb{R}$ and arbitrary, unstructured workflow graphs, the state space is infinite. Most verification problems, and in particular reachability, are undecidable in this setting. However, since we encode verification tasks as CLP queries, whose evaluation is based on a symbolic representation of states, by applying state subsumption we can avoid the actual exploration of the whole state space and terminate in many concrete verification examples.

(1) A very relevant behavioral property of a DAPS p is that, from any reachable state, it is possible to complete the process, i.e., reach the final state. This property, also known as *option to complete* [2], holds if the following query fails:

$\leftarrow reachable\_state(s_{0p}, S) \wedge \neg reachable\_final(S)$

where *reachable_final*(S) holds if a final state can be reached from S, i.e.,

$reachable\_final(S) \leftarrow\ reachable\_state(S,S_f) \wedge holds(final(p),S_f)$

In our running example, modified as suggested at the end of Section 4.2, the above query fails, hence enforcing the option to complete property.

(2) Another property that may reveal potential flaws in a DAPS is *executability* [11], according to which no activity reached by the control flow should be unable to execute due to some unsatisfied enabling condition. In our framework we can verify non-executability by the following query, which succeeds if it can be reached a state where some activity A is waiting for execution but its precondition is not enforced.

$$\leftarrow reachable\_state(s_{0p},S) \wedge holds(cf(A_1,A,p),S) \wedge activity(A) \\ \wedge pre(C,A,p) \wedge \neg holds(C,S)$$

In our running example we have a case of non-executability whenever *apply_coupon* and *choose_prod* are both scheduled for execution after $o1$, according to the inclusive gateways semantics. In this case, since *apply_coupon* requires a value of *item_price* greater than 0, it will not begin its execution until the completion of *apply_coupon*.

(3) Temporal queries can also be used for the verification of *compliance rules*, i.e., directives expressing internal policies and regulations aimed at specifying the way an enterprise operates. We report below two such compliance rules related to our running example. The first one requires that in any possible enactment the amount never reaches a negative value. The following query succeeds if it is possible to reach a state of the process where $amount < 0$.

$$\leftarrow reachable\_state(s_{0eProc},S) \wedge holds(amount < 0,S)$$

In our running example, modified to avoid deadlock as indicated at the end of Section 4.2, this query fails, thus enforcing the compliance rule.

A second example is a compliance rule requiring that it is possible to add an item to the order after the payment of part of the due amount. This property is encoded by the following query, which succeeds in our running example.

$$\leftarrow reachable\_state(s_{0eProc},S_1) \wedge holds(en(payment,eProc),S_1) \\ \wedge\ reachable\_state(S_1,S_2) \wedge holds(en(add\_item,eProc),S_2)$$

5 Implementation

We implemented the proposed framework in a proof-of-concept tool, extending the system discussed in [12, 9]. This extension provides an integrated environment to: *i)* edit BPs using the graphical BPMN editor shown in Figure 1, *ii)* annotate BP elements in terms of preconditions, effects, and guards *iii)* translate the annotated BPs into Constraint Logic Programming, and *iv)* handle the communication with an underlying inference engine that compiles the CLP program representing the DAPS and performs reasoning services through the querying mechanism.

The inference engine is built upon SWI Prolog[2] and is based on a suitable encoding of the set $\mathcal{KB}$ of rules defined in Section 4. The constraints are handled by the built-in SWI solver for equality and inequality constraints over the reals. While the translation

[2] http://www.swi-prolog.org/

of $\mathcal{KB}$ is straightforward, additional considerations are needed when dealing with the evaluation of queries that involve state reachability (i.e., the relation *reachable_state*). Indeed, similarly to many other Prolog engines, SWI generates derivations from a query by using a depth-first search strategy that never looks at the queries derived before the current one. For this reason query evaluation may enter an infinite loop in the presence of recursive rules. To mitigate this difficulty, at least partially, the definition of the predicate *reachable_state* (reported below) has been implemented through *memoing* [13], i.e., a strategy for storing intermediate results and avoiding to prove sub-goals more than once.

```
reachable_state(X,X).
reachable_state(X,Z) :-
      result(X,_,Y),
      constrained_state(Y,cs(Y1,CY)),
      \+ subsumed_by_visited(cs(Y1,CY)),
      assert(visited(cs(Y1,CY))),
      reachable_state(Y,Z).
```

where: `constrained_state(Y,cs(Y1,CY))` holds if `Y1` is the set of fluents in `Y` and `CY` is the constraint associated with the variables in `Y`; `subsumed_by_visited(cs(Y1,CY))` holds if a fact `visited(cs(W1,CW))` belongs to the program, such that `Y1` is an instance of `W1` for some substitution and `CW` entails `CY`; `assert(visited(cs(Y1,CY)))` adds the fact `visited(cs(Y1,CY))` to the program.

Essentially, during the evaluation of a goal including a *reachable_state* atom (i.e., requiring the state space exploration), for every reached state a subsumption test is performed, in order to verify whether a state that subsumes the current one has been already considered. If it is the case, the sub-goal fails, avoiding redundant computations. Otherwise, the state is added to the set of the visited states and the exploration proceeds.

By exploiting this simple memoing mechanism, we were able to verify many reachability properties of various versions of the eProcurement example, and in particular all properties presented in Section 4.3, which in principle require the exploration of an infinite state space.

6 Conclusions and Discussion

In this paper we presented a framework, grounded in Constraint Logic Programming, for reasoning on BP models represented through a *data-aware* extension of the popular BPMN notation. The behavioral semantics of a process is defined as a (possibly infinite) state transition system by following an approach derived from the Fluent Calculus. Data values have a symbolic representation based on arithmetic constraints, which allow us to specify data objects manipulations and database interactions in terms of preconditions and effects of the enactment of activities. We discussed the reasoning tasks the framework enables, dealing with enactment, testing and verification, and how they can be implemented through CLP engines.

Our work is related to a growing stream of research ([14–17]) dealing with an integrated view of the process and data perspective in BP modeling and verification

(see, e.g., [5] for a survey). In [14] decidability and complexity results are provided for the verification of artifact-systems, specified according to a variant of the artifact-centric model introduced by IBM [3, 4] and extended with data dependencies and arithmetic. The main result is the identification of classes of decidable and tractable artifact-systems, under particular restrictions on the interactions between control flow and artifacts. In [15] Data-Centric Dynamic Systems are introduced, through a formalism whose expressive power is equivalent to artifact-centric models, where arithmetic constraints among data objects are not explicitly addressed. Here, a process is described in terms of condition-action rules and the data layer is a relational database that is updated by actions execution. The verification of temporal properties given μ-calculus variants is addressed, and some decidability and computational complexity results are provided. With respect to the above works, our objective is more pragmatic, in that our formalization enables the implementation of a number of reasoning services by taking advantage of CLP engines specifically designed to deal with constraint solving. Since we do not pose any restrictions on the procedural description of processes, directly corresponding to BPMN diagrams, nor to data manipulation, in our setting verification is in general undecidable. However, by considering process runs of bounded length, many tasks with a practical relevance terminate, in particular when related to simulation and testing.

Several approaches have been recently proposed in the workflow community to take into account process data, extending consolidated results originally conceived for dealing with control-flow only. Among them, some related problems addressed in literature concern: run-time support for the enactment of data-aware process models [18]; verification of workflow models where the data-flow is also represented [19]; conformance checking of the execution logs of a BP with respect to its modeled behavior [20]. In contrast, we focus on a logic-based formalization of data-aware BPs to enable static analysis tasks in the presence of arithmetic constraints and data dependencies.

Finally, other approaches based on LP that are worth mentioning are [17, 16]. In [17] it is discussed a formalization of constraints dealing with the data-flow perspective, designed to extend declarative workflow specifications. The framework is grounded in the Event Calculus, and its LP implementation is intended to address a-posteriori analysis of process logs and runtime monitoring. In [16] it is presented an approach to BP verification based on an extension of answer set programming with temporal logic and constraints, where the compliance of business rules is checked by bounded model checking techniques extended with constraint solving for dealing with conditions on numeric data. With respect to our setting, the framework assumes finite domains for variables besides several restrictions on the workflow, resulting in a less expressive (decidable) language, whose verification can be reduced to finite-state analysis.

The preliminary results presented in this paper open up several directions for future research. First of all, we plan to push forward the empirical evaluation of our proposal in each application scenario reported in Section 4. To this end, a relevant aspect to be further elaborated regards the adoption of program optimization techniques to enhance the performances of the reasoning approach. On a more theoretical perspective, we are investigating the class of BPs for which a symbolic finite-state space can be computed, in order to characterize the decidability and complexity of temporal verification tasks.

References

1. OMG: Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0 (2011)
2. van der Aalst, W.M.P.: The application of Petri nets to workflow management. Journal of Circuits, Systems, and Computers **8**(1) (1998) 21–66
3. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Syst. J. **42**(3) (July 2003) 428–445
4. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: On the Move to Meaningful Internet Systems: OTM 2008, Part II. Volume 5332 of LNCS. Springer (2008) 1152–1163
5. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data-aware process analysis: A database theory perspective. In: Proceedings of the 32nd Symposium on Principles of Database Systems. PODS '13, ACM (2013) 1–12
6. van der Aalst, W., Weske, M.: Case handling: A new paradigm for business process support. Data Knowl. Eng. **53**(2) (2005) 129–162
7. Thielscher, M.: Introduction to the Fluent Calculus. Electron. Trans. Artif. Intell. **2** (1998) 179–192
8. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581
9. Smith, F., Proietti, M.: Rule-based behavioral reasoning on semantic business processes. In: Proceedings of the 5th Int. Conf. on Agents and Artificial Intelligence, Volume II, SciTePress (2013) 130–143
10. Lloyd, J.W.: Foundations of logic programming. Springer-Verlag New York, Inc. (1987)
11. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: On the verification of semantic business process models. Distrib. Parallel Databases **27** (2010) 271–343
12. Smith, F., Missikoff, M., Proietti, M.: Ontology-Based Querying of Composite Services. In: Business System Management and Engineering. Volume 7350 of LNCS. Springer (2010) 159–180
13. Dietrich, S., Fan, C.: On the completeness of naive memoing in Prolog. New Generation Computing **15**(2) (1997) 141–162
14. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM Trans. Database Syst. **37**(3) (September 2012) 22:1–22:36
15. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proceedings of the 32nd Symposium on Principles of Database Systems. PODS '13, ACM (2013) 163–174
16. Giordano, L., Martelli, A., Spiotta, M., Dupré, D.T.: Business process verification with constraint temporal answer set programming. TPLP **13**(4-5) (2013) 641–655
17. Montali, M., Chesani, F., Mello, P., Maggi, F.M.: Towards data-aware constraints in Declare. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC '13, ACM (2013) 1391–1396
18. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Proceedings of 11th International Conference on Business Process Management, BPM 2013. Volume 8094 of LNCS. Springer (2013) 171–186
19. Sidorova, N., Stahl, C., Trcka, N.: Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. Inf. Syst. **36**(7) (2011) 1026–1043
20. de Leoni, M., van der Aalst, W.M.P.: Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In: Proceedings of 11th International Conference on Business Process Management, BPM 2013. Volume 8094 of LNCS. Springer (2013) 113–129