Verifying Controllability of Time-Aware Business Processes*

Emanuele De Angelis[1], Fabio Fioravanti[1], Maria Chiara Meo[1]
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] DEC, University 'G. D'Annunzio', Pescara, Italy
{emanuele.deangelis,fabio.fioravanti,cmeo}@unich.it
[2] DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Rome, Italy maurizio.proietti@iasi.cnr.it

Abstract. We present an operational semantics for *time-aware* business processes, that is, processes modeling the execution of business activities, whose durations are subject to linear constraints over the integers. We assume that some of the durations are *controllable*, that is, they can be determined by the organization that enacts the process, while others are *uncontrollable*, that is, they are determined by the external world.

Then, we consider *controllability* properties, which guarantee the completion of the enactment of the process, satisfying the given duration constraints, independently of the values of the uncontrollable durations. Controllability properties are encoded by *quantified* reachability formulas, where the reachability predicate is recursively defined by a set of Constrained Horn Clauses (CHCs). These clauses are automatically derived from the operational semantics of the process.

Finally, we present two algorithms for solving the so called *weak* and *strong* controllability problems. Our algorithms reduce these problems to the verification of a set of quantified integer constraints, which are simpler than the original quantified reachability formulas, and can effectively be handled by state-of-the-art CHC solvers.

1 Introduction

A business process model is a procedural, semi-formal specification of the order of execution of the activities in a business process (or BP, for short) and of the way these activities must coordinate to achieve a goal [18,35]. Many notations for BP modeling, and in particular the popular BPMN [26], allow the modeler to express time constraints, such as deadlines and activity durations. However, time related aspects are neglected when the semantics of a BP model is given through the standard Petri Net formalization [18], which focuses on the control flow only. Thus, formal reasoning about time related properties, which may be very important in many analysis tasks, is not possible in that context.

* This work has been partially funded by INdAM-GNCS (Italy). E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at IASI-CNR, Rome, Italy.

In order to overcome this difficulty, various approaches to BP modeling with time constraints have been proposed in the literature (see [5] for a recent survey). Some of these approaches define the semantics of *time-aware* BP models by means of formalisms such as *time Petri nets* [24], *timed automata* [33], and *process algebras* [36]. Properties of these models can then be verified by using very effective reasoning tools available for those formalisms [3,17,23].

In this paper we address the problem of verifying the *controllability* of time-aware business processes. This notion has been introduced in the context of scheduling and planning problems over *Temporal Networks* [32], but it has not received much attention in the more complex case of time-aware BP models. We assume that some of the durations are *controllable*, that is, they can be determined by the organization that enacts the process, while others are *uncontrollable*, that is, they are determined by the external world. Properties like *strong controllability* and *weak controllability* guarantee, in different senses, that the process can be completed, satisfying the given duration constraints, for all possible values of the uncontrollable durations. Controllability properties are particularly relevant in scenarios (e.g., healthcare applications [9]) where the completion of a process within a certain deadline must be guaranteed even if the durations of some activities cannot be fully determined in advance.

We propose a method for solving controllability problems by extending a logic-based approach that has been recently proposed for modeling and verifying time-aware business processes [11]. This approach represents both the BP structure and the BP behavior in terms of *Constrained Horn Clauses* (CHCs) [4], also known as *Constraint Logic Programs* [19], over Linear Integer Arithmetics. (Here we will use the 'Constrained Horn Clauses' term, which is more common in the area of verification.) In our setting, controllability properties can be defined by *quantified reachability formulas.*

An advantage of the logic-based approach over other approaches is that it allows a seamless integration of the various reasoning tasks needed to analyze business processes from different perspectives. For instance, logic-based techniques can easily perform ontology-related reasoning about the business domain where processes are enacted [29,34] and reasoning on the manipulation of data objects of an infinite type, such as databases or integers [2,10,28]. Moreover, for the various logic-based reasoning tasks, we can make use of very effective tools such as CHC solvers [15] and Constraint Logic Programming systems.

For reasons of simplicity, in this paper we consider business process models where the only time-related elements are constraints over task durations. However, other notions can be modeled by following a similar approach.

The main contributions of this paper are the following. (1) We define an operational semantics of time-aware BP models, which modifies the semantics presented in [11] by formalizing the synchronization of activities at parallel merge gateways and we prove some properties of this new semantics (see Section 3). Our semantics is defined under the assumption that the process is *safe*, that is, during its enactment there are no multiple, concurrent executions of the same task [1]. (2) We provide formal definitions of strong and weak controllability properties as quantified reachability formulas (see Section 4). (3) We present a transformation technique for automatically deriving the CHC representation of the reachability relation starting from the CHC encoding of the semantics of time-aware processes, and of the process and property under consideration (see Section 5). (4) Finally, we propose two algorithms that solve strong and weak controllability problems for time-aware BPs. These algorithms avoid the direct verification of quantified reachability formulas, which often cannot be handled by state-of-the-art CHC solvers, and they verify, instead, a set of simpler Linear Integer Arithmetic formulas, whose satisfiability can effectively be worked out by the Z3 constraint solver [15] (see Section 6). Detailed proofs of the results presented here can be found in a technical report [12].

2 Preliminaries

In this section we recall some basic notions about the constrained Horn clauses (CHCs) and the Business Process Model and Notation (BPMN).

Let *RelOp* be the set $\{=, \neq, \leq, \geq, <, >\}$ of predicate symbols denoting the familiar relational operators over the integers. If p_1 and p_2 are linear polynomials with integer variables and integer coefficients, then $p_1 R\, p_2$, with $R \in \mathit{RelOp}$, is an *atomic constraint.* A *constraint* c is either *true* or *false* or an atomic constraint or a conjunction or a disjunction of constraints. Thus, constraints are formulas of Linear Integer Arithmetics (*LIA*). An *atom* is a formula of the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol not in *RelOp* and $t_1, \ldots, t_m$ are terms constructed as usual from variables, constants, and function symbols. A *constrained Horn clause* (or simply, a *clause*, or a CHC) is an implication of the form $A \leftarrow c, G$ (comma denotes conjunction), where the conclusion (or *head*) A is either an atom or *false*, the premise (or *body*) is the conjunction of a constraint c and a (possibly empty) conjunction G of atoms. The empty conjunction is identified with *true*. A *constrained fact* is a clause of the form $A \leftarrow c$, and a *fact* is a clause whose premise is *true*. We will write $A \leftarrow \mathit{true}$ also as $A \leftarrow$. A clause is *ground* if no variable occurs in it. A clause $A \leftarrow c, G$ is said to be *function-free* if no function symbol occurs in (A, G), while arithmetic function symbols may occur in c. For clauses we will use a Prolog-like syntax (in particular, '_' stands for an anonymous variable).

A set S of CHCs is said to be *satisfiable* if $S \cup \mathit{LIA}$ has a model, or equivalently, $S \cup \mathit{LIA} \not\models \mathit{false}$. Given two constraints c and d, we write $c \sqsubseteq d$ if $\mathit{LIA} \models \forall(c \rightarrow d)$, where $\forall(F)$ denotes the universal closure of formula F. The *projection* of a constraint c onto a set X of variables is a new constraint c', with variables in X, which is equivalent, in the domain of *rational* numbers, to $\exists Y. c$, where Y is the set of variables occurring in c and not in X. Clearly, $c \sqsubseteq c'$.

A BPMN model of a business process consists of a diagram drawn by using graphical notations representing: (i) *flow objects* and (ii) *sequence flows* (also called *flows*, for short).

A flow object is: either (i.1) a *task*, depicted as a rounded rectangle, or (i.2) an *event*, depicted as a circle, or (i.3) a *gateway*, depicted as a diamond. A sequence flow is depicted as an arrow connecting a source flow object to a target flow object (see Figure 1).

Tasks are atomic units of work performed during the enactment (or execution) of the business process. An events is either a *start* event or an *end* event, which denote the beginning and the completion, respectively, of the activities of the process. Gateways denote the branching or the merging of activities. In this paper we consider the following four kinds of gateways: (a) the *parallel branch*, that simultaneously activates all the outgoing flows, if its single incoming flow is activated (see $g1$ in Figure 1), (b) the *exclusive branch*, that (non-deterministically) activates exactly one out of its outgoing flows, if its single incoming flow is activated (see $g3$ in Figure 1), (c) the *parallel merge*, that activates the single outgoing flow, if all the incoming flows are simultaneously activated (see $g4$ in Figure 1), and (d) the *exclusive merge*, that activates the single outgoing flow, if any one of the incoming flows is activated (see $g2$ in Figure 1). The diamonds representing parallel gateways and exclusive gateways are labeled by '**+**' and '**×**', respectively. Branch and merge gateways are also called *split* and *join* gateways, respectively.

A sequence flow denotes that the execution of the process can pass from the source object to the target object. If there is a sequence flow from a_1 to a_2, then a_1 is a *predecessor* of a_2, and symmetrically, a_2 is a *successor* of a_1. A *path* is a sequence of flow objects such that every pair of consecutive objects in the sequence is connected by a sequence flow.

We will consider models of business processes that are *well-formed*, in the sense that they satisfy the following properties: (1) every business process contains a single start event and a single end event, (2) the start event has exactly one successor and no predecessors, and the end event has exactly one predecessor and no successors, (3) every flow object occurs on a path from the start event to the *end* event, (4) (parallel or exclusive) branch gateways have exactly one predecessor and at least one successor, while (parallel or exclusive) merge gateways have at least one predecessor and exactly one successor, (5) tasks have exactly one predecessor and one successor, and (6) no cycles through gateways only are allowed. Note that we do not require BP models to be block-structured.

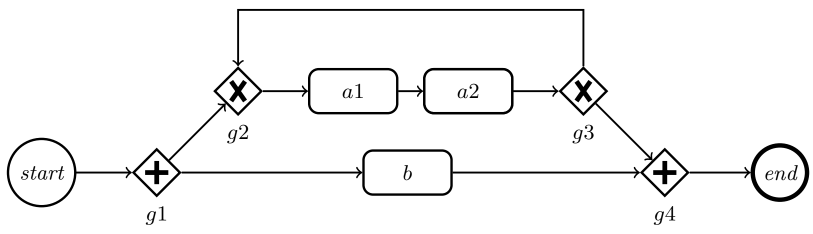

Fig. 1. A business process *Proc*.

In Figure 1 we show the BPMN model of a business process, called *Proc*. After the *start* event, the parallel branch $g1$ simultaneously activates the two flow objects $g2$ and b. The exclusive merge $g2$ activates the sequential execution of the task $a1$ which in turn activates the execution of the task $a2$ which is followed by the exclusive branch $g3$. After $g3$, the execution can either return to $g2$ or proceed to $g4$. If $g3$ and b both complete their executions simultaneously, then the parallel merge $g4$ is executed, the *end* event occurs, and the process *Proc* terminates.

3 Specification and Semantics of Business Processes

In this section we introduce the notion of a *Business Process Specification* (BPS), which formally represents a business process by means of CHCs. Then we define the operational semantics of a BPS.

Business Process Specification via CHCs A BPS contains: (i) a set of ground facts that specify the flow objects and the sequence flows between them, and (ii) a set of clauses that specify the duration of each flow object and the controllability (or uncontrollability) of that duration.

For the flow objects we will use of the following predicates: $\mathit{task}(X)$, $\mathit{event}(X)$, $\mathit{gateway}(X)$, $\mathit{par\_branch}(X)$, $\mathit{par\_merge}(X)$, $\mathit{exc\_branch}(X)$, $\mathit{exc\_merge}(X)$ with the expected meaning. For the sequence flows we will use the predicate $\mathit{seq}(X,Y)$ meaning that there is a sequence flow from X to Y. For every task X we specify its duration by the constrained fact $\mathit{duration}(X,D) \leftarrow d_{min} \leq D \leq d_{max}$, where d_{min} and d_{max} are positive integer constants representing the minimal and the maximal duration of X, respectively. Events and gateways, being instantaneous, have duration 0. For every task X and its duration, we also specify that they are controllable, or uncontrollable, by the fact $\mathit{controllable}(X) \leftarrow$, or $\mathit{uncontrollable}(X) \leftarrow$, respectively.

In Figure 2 we show the BPS of process *Proc* of Figure 1. For reasons of space, in that specification we did not list all the facts for the tasks, events, gateways, and sequence flows of the diagram of *Proc*.

$$\begin{array}{llll}
\mathit{task}(a1) \leftarrow & \mathit{event}(\mathit{start}) \leftarrow & \mathit{gateway}(g1) \leftarrow & \ldots \\
\mathit{par\_branch}(g1) \leftarrow & \mathit{seq}(\mathit{start}, g1) \leftarrow & \mathit{seq}(g1, b) \leftarrow & \ldots \\
\mathit{uncontrollable}(a1) \leftarrow & \mathit{controllable}(a2) \leftarrow & \mathit{controllable}(b) \leftarrow &
\end{array}$$

$$\mathit{duration}(a1,D) \leftarrow 2 \leq D \leq 4 \quad \mathit{duration}(a2,D) \leftarrow 1 \leq D \leq 2 \quad \mathit{duration}(b,D) \leftarrow 5 \leq D \leq 6$$

$$\mathit{duration}(X,D) \leftarrow \mathit{event}(X), D=0 \qquad \mathit{duration}(X,D) \leftarrow \mathit{gateway}(X), D=0$$

Fig. 2. The CHCs of the Business Process Specification of process *Proc* of Figure 1.

Operational Semantics We define the operational semantics of a business process under the assumption that the process is *safe*, that is, during its enactment there are no multiple, concurrent executions of the same flow object [1]. By using this assumption, we represent the *state* of a process enactment as a set of properties, called *fluents* holding at a time instant. We borrow the notion of fluent from *action languages* such as the *Situation Calculus* [25], the *Event Calculus* [20], or the *Fluent Calculus* [30], but we will present our semantics by following the *structural operational* approach often adopted in the field of programming languages.

Formally, a state $s \in \mathit{States}$ is a pair $\langle F, t \rangle$, where F is a set of fluents and t is a time instant, that is, a non-negative integer. A fluent is a term of the form:

(i) $\mathit{begins}(x)$, which represents the beginning of the enactment of the flow object x, (ii) $\mathit{completes}(x)$, which represents the completion of the enactment of x, (iii) $\mathit{enables}(x, y)$, which represents that the flow object x has completed its enactment and it enables the enactment of its successor y, and (iv) $\mathit{enacting}(x, r)$, which represents that the enactment of x requires r units of time to complete (for this reason r is called the *residual time* of x). We have that $\mathit{begins}(x)$ is equivalent to $\mathit{enacting}(x, r)$, where r is the duration of x, and $\mathit{completes}(x)$ is equivalent to $\mathit{enacting}(x, 0)$. (This redundant representation allows us to write simpler rules for the operational semantics below.)

The operational semantics is defined by a rewriting relation $\longrightarrow$ which is a subset of $\mathit{States} \times \mathit{States}$. This relation is specified by rules S_1–S_7 below, where we use the following predicates, besides the ones introduced in Section 3 for defining the BPS: (i) $\mathit{not\_par\_branch}(x)$, which holds if x is *not* a parallel branch, and (ii) $\mathit{not\_par\_merge}(x)$, which holds if x is *not* a parallel merge.

$$(S_1)\quad \frac{\mathit{begins}(x) \in F \qquad \mathit{duration}(x, d)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{begins}(x)\}) \cup \{\mathit{enacting}(x, d)\},\ t\rangle}$$

$$(S_2)\quad \frac{\mathit{completes}(x) \in F \qquad \mathit{par\_branch}(x)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{completes}(x)\}) \cup \{\mathit{enables}(x, s) \mid \mathit{seq}(x, s)\},\ t\rangle}$$

$$(S_3)\quad \frac{\mathit{completes}(x) \in F \qquad \mathit{not\_par\_branch}(x) \qquad \mathit{seq}(x, s)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{completes}(x)\}) \cup \{\mathit{enables}(x, s)\},\ t\rangle}$$

$$(S_4)\quad \frac{\forall p\ \mathit{seq}(p, x) \rightarrow \mathit{enables}(p, x) \in F \qquad \mathit{par\_merge}(x)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{enables}(p, x) \mid \mathit{enables}(p, x) \in F\}) \cup \{\mathit{begins}(x)\},\ t\rangle}$$

$$(S_5)\quad \frac{\mathit{enables}(p, x) \in F \qquad \mathit{not\_par\_merge}(x)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{enables}(p, x)\}) \cup \{\mathit{begins}(x)\},\ t\rangle}$$

$$(S_6)\quad \frac{\mathit{enacting}(x, 0) \in F}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{\mathit{enacting}(x, 0)\}) \cup \{\mathit{completes}(x)\},\ t\rangle}$$

$$(S_7)\quad \frac{\mathit{no\_other\_premises}(F) \qquad \exists x\, \exists r\ \mathit{enacting}(x, r) \in F \qquad m > 0}{\langle F, t\rangle \longrightarrow \langle F \ominus m \setminus \mathit{Enbls},\ t + m\rangle}$$

where: (i) $\mathit{no\_other\_premises}(F)$ holds iff none of the premises of rules S_1–S_6 holds, (ii) $m = \mathit{min}\{r \mid \mathit{enacting}(x, r) \in F\}$, (iii) $F \ominus m$ is the set F of fluents where every $\mathit{enacting}(x, r)$ is replaced by $\mathit{enacting}(x, r - m)$, and (iv) $\mathit{Enbls} = \{\mathit{enables}(p, s) \mid \mathit{enables}(p, s) \in F\}$.

We assume that, for every flow object x, there exists a *unique* value d of its duration, satisfying the given constraint, which is used for *every* application of rule S_1. Note that S_7 is the only rule that formalizes the passing of time, as it infers rewritings of the form $\langle F, t\rangle \longrightarrow \langle F', t + m\rangle$, with $m > 0$. In contrast, rules S_1–S_6 infer state rewritings of the form $\langle F, t\rangle \longrightarrow \langle F', t\rangle$, where time does not pass. Here is the explanation of rules S_1–S_7.

(S_1) If the execution of a flow object x begins at time t, then, at the same time t, x is enacting and its residual time is its duration d;

(S_2) If the execution of the parallel branch x completes at time t, then x enables *all its successors* at time t;

(S_3) If the execution of x completes at time t and x is not a parallel branch, then x enables *precisely one of its successors* at time t (in particular, this case occurs when x is a task);

(S_4) If *all* the predecessors of x enable the parallel merge x at time t, then the execution of x begins at time t;

(S_5) If *at least one* predecessor p of x enables x at time t and x is not a parallel merge, then the execution of x begins at time t (in particular, this case occurs when x is a task);

(S_6) If a flow object x is enacting at time t with residual time 0, then the execution of x completes at the same time t;

(S_7) Suppose that: (i) none of rules S_1–S_6 can be applied for computing a state rewriting $\langle F,t\rangle \longrightarrow \langle F',t'\rangle$, (ii) at time t at least one task is enacting with *positive* residual time (note that flow objects different from tasks cannot have positive residual time), and (iii) m is the least value among the residual times of all the tasks enacting at time t. Then, (i) every task x that is enacting at time t with residual time r, is enacting at time $t+m$ with residual time $r-m$ and (ii) all $\mathit{enables}(p,s)$ fluents are removed.

Due to rules S_4 and S_5, if a fluent of the form $\mathit{enables}(p,s)$ is removed by applying rule S_7, then s necessarily refers to a parallel merge that is not enabled at time t by some of its predecessors. Thus, a parallel merge is executed if and only if it gets *simultaneously* enabled by all its predecessors. For lack of space we omit to model the asynchronous version of the parallel merge [11], which does not require the simultaneity condition. Note also that, if desired, tasks can be added for modeling delays in an explicit way.

We say that state $\langle F',t'\rangle$ is *reachable* from state $\langle F,t\rangle$, if $\langle F,t\rangle \longrightarrow^* \langle F',t'\rangle$, where $\longrightarrow^*$ denotes the reflexive, transitive closure of the rewriting relation $\longrightarrow$.

The *initial state* is the state $\langle\{\mathit{begins}(\mathit{start})\},0\rangle$. The *final state* is the state of the form $\langle\{\mathit{completes}(\mathit{end})\},t\rangle$, for some time instant t.

Properties of the Operational Semantics Now we first introduce the notions of: (i) a derivation, which is a sequence of states, and (ii) a selection function, which is a rule providing the order in which fluents are rewritten according to the relation $\longrightarrow$. In Theorem 1 below we will prove that the relation $\longrightarrow$ is independent of that order.

Definition 1 (Derivation). *A derivation from a state s_0 in a BPS is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that for all $i \geq 0$, $s_i \longrightarrow s_{i+1}$.*

Let States_{sat} be the subset of *States* where $\mathit{no\_other\_premises}(F)$ holds.

Definition 2 (Selection function). *Let δ be a finite derivation whose last state is $\langle F,t\rangle$, with $F \neq \emptyset$. A selection function $\mathcal{R}$ is a function that takes the derivation δ and returns: either (i) a subset of F whose elements satisfy the conditions in the premise of exactly one rule among S_1–S_6, or (ii) the union of* *the set of the 'enacting' fluents in F and the set of the 'enables' fluents in F, if $\langle F, t\rangle \in States_{sat}$ and at least one 'enacting' fluent belongs to F.*

The selection function is well-defined, because each fluent can be rewritten by at the most one rule and the rules S_1–S_6 are not overlapping, that is, the sets of fluents which can fire two distinct rules in S_1–S_6 (or two different instances of the same rule) are disjoint.

Definition 3 (Derivation via $\mathcal{R}$). *Given a selection function $\mathcal{R}$, we say that a derivation δ is via $\mathcal{R}$ iff for each proper prefix δ' of δ ending with a state s, if $s \longrightarrow s'$ and $(\delta'\, s')$ is a prefix of δ, then $\mathcal{R}(\delta')$ are the fluents of s that are rewritten when deriving s'.*

Theorem 1. *For every derivation δ from a state s_0, and selection function $\mathcal{R}$, there exists a derivation δ' from s_0 via $\mathcal{R}$ such that if $\langle F,t\rangle$ is a state in δ and f is a fluent in F, then there exists a state $\langle F',t\rangle$ in δ' such that f is a fluent in F'.*

4 Encoding Controllability Properties into CHCs

In this section we show how weak and strong controllability properties are formalized by defining a CHC *interpreter*, that is, a set of CHCs that encodes the operational semantics of business processes. Then, the interpreter is *specialized* with respect to the business process and property to be verified.

4.1 Encoding the Operational Semantics in CHCs

A state of the operational semantics is encoded by a term of the form $s(F, T)$, where F is a set of fluents and T is the time instant at which the fluents in F hold. The rewriting relation $\longrightarrow$ between states and its reflexive, transitive closure $\longrightarrow^*$ are encoded by the predicates *tr* and *reach*, respectively. The clauses defining these predicates are shown in Table 4.1. In the body of the clauses, the atoms that encode the premises of the rules of the operational semantics have been underlined.

The predicate *select*(L, F) encodes a selection function (see Definition 2). We assume that *select*(L, F) holds iff L is a subset of the set F of fluents such that: (i) there exists a clause in $\{C1, \ldots, C6\}$ that updates F by replacing the subset L of F by a new set of fluents, and (ii) among all such subsets of F, L is the one that contains the first fluent, in textual order (in this sense *select*(L, F) is deterministic). The predicate *task_duration*(X, D, U, C) holds iff *duration*(X, D) holds and D belongs to either the list U of durations of the uncontrollable tasks or the list C of durations of the controllable tasks. The predicate *update*(F, R, A, FU) holds iff FU is the set obtained from the set F by removing all the elements of R and adding all the elements of A. The predicate *no_other_premises*(F) holds iff the premise of every rule in $\{C1, \ldots, C6\}$ is unsatisfiable. The predicate *mintime*$(Enacts, M)$ holds iff *Enacts* is a set of fluents of the form *enacting*(X, R) and M is the minimum value of R for the elements of *Enacts*. The predicate *decrease_residual_times*$(Enacts, M, EnactsU)$ holds iff

$C1.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\{\mathit{begins}(X)\},F)},\ \mathit{task\_duration}(X,D,U,C),$
$\quad \mathit{update}(F,\{\mathit{begins}(X)\},\{\mathit{enacting}(X,D)\},FU)$

$C2.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\{\mathit{completes}(X)\},F)},\ \underline{\mathit{par\_branch}(X)},$
$\quad \mathit{findall}(\mathit{enables}(X,S),(\mathit{seq}(X,S)),\mathit{Enbls}),\ \mathit{update}(F,\{\mathit{completes}(X)\},\mathit{Enbls},FU)$

$C3.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\{\mathit{completes}(X)\},F)},\ \underline{\mathit{not\_par\_branch}(X)},\ \mathit{seq}(X,S),$
$\quad \mathit{update}(F,\{\mathit{completes}(X)\},\{\mathit{enables}(X,S)\},FU)$

$C4.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\mathit{Enbls},F)},\ \underline{\mathit{par\_merge}(X)},$
$\quad \underline{\mathit{findall}(\mathit{enables}(P,X),(\mathit{seq}(P,X)),\mathit{Enbls})},\ \mathit{update}(F,\mathit{Enbls},\{\mathit{begins}(X)\},FU)$

$C5.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\{\mathit{enables}(P,X)\},F)},\ \underline{\mathit{not\_par\_merge}(X)},$
$\quad \mathit{update}(F,\{\mathit{enables}(P,X)\},\{\mathit{begins}(X)\},FU)$

$C6.\ \mathit{tr}(s(F,T),s(FU,T),U,C) \leftarrow \underline{\mathit{select}(\{\mathit{enacting}(X,R)\},F),\ R=0},$
$\quad \mathit{update}(F,\{\mathit{enacting}(X,R)\},\{\mathit{completes}(X)\},FU)$

$C7.\ \mathit{tr}(s(F,T),s(FU,TU),U,C) \leftarrow \underline{\mathit{no\_other\_premises}(F)},\ \underline{\mathit{member}(\mathit{enacting}(\_,\_),F)},$
$\quad \underline{\mathit{findall}(Y,(Y=\mathit{enacting}(X,R),\ \mathit{member}(Y,F)),\mathit{Enacts})},$
$\quad \underline{\mathit{mintime}(\mathit{Enacts},M)},\ \underline{M>0},\ \mathit{decrease\_residual\_times}(\mathit{Enacts},M,\mathit{EnactsU}),$
$\quad \mathit{findall}(Z,(Z=\mathit{enables}(P,S),\mathit{member}(Z,F)),\mathit{Enbls}),$
$\quad \mathit{set\_union}(\mathit{Enacts},\mathit{Enbls},\mathit{EnactsEnbls}),\ \mathit{update}(F,\mathit{EnactsEnbls},\mathit{EnactsU},FU),$
$\quad TU=T+M$

$R1.\ \mathit{reach}(S,S,U,C).\qquad R2.\ \mathit{reach}(S,S2,U,C) \leftarrow \mathit{tr}(S,S1,U,C),\ \mathit{reach}(S1,S2,U,C)$

Table 4.1. The CHC interpreter for time-aware business processes.

EnactsU is the set of fluents obtained by replacing every element of *Enacts*, of the form $\mathit{enacting}(X,R)$, with the fluent $\mathit{enacting}(X,\mathit{RU})$, where $\mathit{RU}=R-M$. The predicates $\mathit{member}(\mathit{El},\mathit{Set})$ and $\mathit{set\_union}(A,B,\mathit{AB})$ are self-explanatory. The predicate $\mathit{findall}(X,G,L)$ holds iff X is a term whose variables occur in the conjunction G of atoms, and L is the set of instances of X such that $\exists Y.\,G$ holds, where Y is the tuple of variables occurring in G different from those in X.

We denote by *Sem* the set consisting of clauses $C1,\ldots,C7,R1,R2$, together with the clauses encoding the business process specification.

Theorem 2 (Correctness of Encoding). *Let init be the term that encodes the initial state* $\langle\{\mathit{begins}(\mathit{start})\},0\rangle$, *and* $\mathit{fin}(t)$ *be the term that encodes the final state* $\langle\{\mathit{completes}(\mathit{end})\},t\rangle$. *Then,* $\langle\{\mathit{begins}(\mathit{start})\},0\rangle \longrightarrow^* \langle\{\mathit{completes}(\mathit{end})\},t\rangle$ *iff there exist tuples of integers* x *and* y *such that* $\mathit{Sem}\cup\mathit{LIA} \models \mathit{reach}(\mathit{init},\mathit{fin}(t),x,y)$.

4.2 Encoding Controllability Properties

A *reachability property* is defined by a clause of the form:

$RP.\ \mathit{reachProp}(U,C)\ \leftarrow\ c(T,U,C),\ \mathit{reach}(\mathit{init},\ \mathit{fin}(T),\ U,\ C)$

where: (i) U and C denote tuples of uncontrollable and controllable durations, respectively, and (ii) $c(T,U,C)$ is a constraint.

We say that the duration D of task X is *admissible* iff $\mathit{duration}(X,D)$ holds. The *strong controllability* problem for a BPS consists in checking whether or not there exist durations C such that, for all admissible durations U, the property *reachProp*(U, C) holds. The *weak controllability* problem for a BPS consists in checking whether or not, for all admissible durations U, there exist durations C such that *reachProp*(U, C) holds. Note that, if *reachProp*(U, C) holds, then all durations used to reach the final state are admissible, and hence in the definition of controllability there is no need to require that the existentially quantified durations C are admissible. We denote by I the set $Sem \cup \{RP\}$.

Definition 4 (Strong and weak controllability). *Given a BPS $\mathcal{B}$,*
- $\mathcal{B}$ is strongly controllable iff $I \cup LIA \models \exists C\, \forall U.\, adm(U) \rightarrow reachProp(U, C)$
- $\mathcal{B}$ is weakly controllable iff $I \cup LIA \models \forall U. adm(U) \rightarrow \exists C\, reachProp(U, C)$
where $adm(U)$ holds iff U is a tuple of admissible durations.

If a business process is weakly controllable, in order to determine the durations of the controllable tasks, we need to know in advance the actual durations of all the uncontrollable tasks. This might not be realistic in practice, as uncontrollable tasks may occur after controllable ones. Strong controllability implies weak controllability and guarantees that suitable durations of the controllable tasks can be computed, before the enactment of the process, by using the constraints on the uncontrollable durations, which are provided by the process specification.

5 Specializing Constrained Horn Clauses

The clauses in I make use of complex terms, and in particular lists of variable length, to represent states (see clauses $C1$–$C7$). Now we present a transformation that *specializes* I to the particular BPS under consideration and derives an equivalent set I_{sp} of function-free CHCs, on which CHC solvers are much more effective. The specialization algorithm is a variant of the ones for the so called *Removal of the Interpreter* proposed in the area of verification of imperative programs [14]. The specialization algorithm makes use of the following transformation rules: *unfolding*, *definition introduction*, and *folding* [16].

The specialization algorithm (see Figure 3) starts off by unfolding clause RP, thereby performing a symbolic exploration of the space of the reachable states. The unfolding rule is defined as follows.

Unfolding Rule. Let C be a clause of the form $H \leftarrow c, L, A, R$, where H and A are atoms, L and R are (possibly empty) conjunctions of atoms, and c is a constraint. Let $\{K_i \leftarrow c_i, B_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart) clauses in I such that, for $i = 1, \ldots, m$, A is unifiable with K_i via the most general unifier ϑ_i and $(c, c_i)\, \vartheta_i$ is satisfiable. We define the following function:

$$Unf(C, A, I) = \{\, (H \leftarrow c, c_i, L, B_i, R)\, \vartheta_i \mid i = 1, \ldots, m\,\}$$

Each clause in $Unf(C, A, I)$ is said to be derived by *unfolding A in C* using I.

Unrestricted unfolding may cause the nontermination of the UNFOLDING phase. Now we introduce the notion of *unfoldable atom* that guarantees that the UNFOLDING phase does terminate. Every atom whose predicate is different from *reach* is unfoldable. Every atom of the form *reach*$(s(F1, T1), s(F2, T2), U, C)$ is *unfoldable* if $I \cup LIA \not\models \exists X.\, no\_other\_premises(F1)$, where X is the tuple of variables occurring in $F1$ (that is, rule S_7 is not applicable in state $s(F1, T1)$).

```
Input: A set $I$ of CHCs defining a reachability property.
Output: A set $I_{sp}$ of function-free CHCs such that, for all (tuples of) integer values $x$
and $y$, $I \cup LIA \models reachProp(x, y)$ iff $I_{sp} \cup LIA \models reachProp(x, y)$.

INITIALIZATION:
$I_{sp} := \emptyset$;  $InCls := \{RP\}$;  $Defs := \emptyset$;
while in $InCls$ there is a clause $C$ with an atom in its body do
  UNFOLDING:
  $SpC := Unf(C, A, I)$   where $A$ is the atom in the body of $C$;
    while in $SpC$ there is a clause $D$ whose body contains an occurrence of an unfoldable atom $A$ do  $SpC := (SpC - \{D\}) \cup Unf(D, A, I)$
    end-while;
  DEFINITION-INTRODUCTION & FOLDING:
    while in $SpC$ there is a clause $E$ of the form:
      $H \leftarrow e,\ reach(s(fl(Rs), T), fin(Tf), U, C)$
    where: $H$ is an atom, $e$ is a constraint, $fl(Rs)$ is a term denoting a set of fluents,
    $Rs$ is a tuple of variables denoting residual times, $T$ is a variable denoting the
    absolute time, $U$ is a tuple of variables denoting the uncontrollable durations, and
    $C$ is a tuple of variables denoting the controllable durations do
      if   in $Defs$ there is a clause $D$ of the form (up to variable renaming):
             $newp(Rs, T, Tf, U, C) \leftarrow d(Rs),\ reach(s(fl(Rs), T), fin(Tf), U, C)$
           where: $d(Rs)$ is a constraint such that $e \sqsubseteq d(Rs)$
      then $SpC := (SpC - \{E\}) \cup \{ H \leftarrow e,\ newp(Rs, T, Tf, U, C)\}$;
      else
           let $F$ be the clause:
             $newr(Rs, T, Tf, U, C) \leftarrow f(Rs),\ reach(s(fl(Rs), T), fin(Tf), U, C)$
           where: $newr$ is a predicate symbol not occurring in $I \cup Defs$, and $f(Rs)$
           is the projection of $e$ onto $Rs$;
           $InCls := InCls \cup \{F\}$;  $Defs := Defs \cup \{F\}$;
           $SpC := (SpC - \{E\}) \cup \{ H \leftarrow e,\ newr(Rs, T, Tf, U, C) \}$
    end-while;
  $InCls := InCls - \{C\}$;   $I_{sp} := I_{sp} \cup SpC$;
end-while;
```

Fig. 3. The Specialization Algorithm.

At the end of the UNFOLDING phase, for every *reach* atom occurring in the body of a clause, a new predicate definition is introduced by a clause of the form:

$$newr(Rs, T, Tf, U, C) \leftarrow f(Rs),\ \ reach(s(fl(Rs), T), fin(Tf), U, C)$$

where $f(Rs)$ is obtained by projecting the constraint occurring in the body of the clause where the *reach* atom occurs, onto the tuple Rs of variables representing the residual times, and $fl(Rs)$ is a term representing a set of fluents. Then, *reach* atoms with complex arguments representing states, are replaced by function-free calls to the newly introduced predicates, by applying the folding rule. Thus, at the end of the DEFINITION-INTRODUCTION & FOLDING phase, we derive a set of function-free CHCs, which are added to I_{sp}. The specialization algorithm proceeds by adding the clause defining *newr* to the set *InCls* of the clauses to be specialized and to the set *Defs* of the clauses introduced by the definition introduction rule. The strategy terminates when all clauses in *InCls* have been processed.

The termination of the specialization algorithm is a consequence of the following two facts: (i) the Unfolding phase terminates and (ii) the set of the new predicates introduced in all executions of the body of the loop is finite, because the given task durations determine a finite set of non-equivalent constraints. Moreover, since the unfolding, definition introduction, and folding rules preserve satisfiability [16], we have the following result.

Theorem 3 (Total Correctness of the Specialization Algorithm). *For any input set I of CHCs, the specialization algorithm terminates. Suppose that the specialization algorithm returns the set I_{sp} of CHCs. Then (i) I_{sp} is a set of function-free CHCs, and (ii) for all (tuples of) integer values x and y,*
$I \cup LIA \models reachProp(x,y)$ *iff* $I_{sp} \cup LIA \models reachProp(x,y)$.

Example 1. Let I be the set of clauses defining the reachability property for the process *Proc* of Figure 1, and let clause *RP* be:
$reachProp(A1,A2,B) \leftarrow reach(init, fin(T), A1, (A2,B))$
where $A1$ denotes the duration of the uncontrollable task $a1$ and $(A2, B)$ denotes the durations of the controllable tasks $a2$ and b. By applying the specialization algorithm with I as input, we derive the following function-free clauses as output:

$reachProp(A1,A2,B) \leftarrow A=A1, B=B1, A1\geq 2, A1\leq 4, B\geq 5, B\leq 6,$
$\quad new2(A, B1, F, G, A1, A2, B)$
$new2(A,B1,C,D,A1,A2,B) \leftarrow H=A+C, I=B1-A, J=0, A\geq 1, I\geq 0, A+I\geq 1,$
$\quad new2(J, I, H, D, A1, A2, B)$
$new2(A,B1,C,D,A1,A2,B) \leftarrow H=B1+C, I=A-B1, J=0, A\geq 1, I\geq 0, A-I\geq 1,$
$\quad new2(I,J,H,D,A1,A2,B)$
$new2(A,B1,C,D,A1,A2,B) \leftarrow H=A2, A=0, H\geq 1, H\leq 2,\ new5(H,B1,C,D,A1,A2,B)$
$new5(A,B1,C,C,A1,A2,B) \leftarrow A=0, B1=0$
$new5(A,B1,C,D,A1,A2,B) \leftarrow H=A+C, I=B1-A, J=0, A\geq 1, I\geq 0, A+I\geq 1,$
$\quad new5(J,I,H,D,A1,A2,B)$
$new5(A,B1,C,D,A1,A2,B) \leftarrow H=B1+C, I=A-B1, J=0, A\geq 1, I\geq 0, A-I\geq 1,$
$\quad new5(I,J,H,D,A1,A2,B)$
$new5(A,B1,C,D,A1,A2,B) \leftarrow H=A1, A=0, H\geq 2, H\leq 4,\ new2(H,B1,C,D,A1,A2,B)$

6 Solving Controllability Problems

State-of-the-art CHC solvers are often not effective in solving controllability problems defined by a direct encoding of the formulas in Definition 4, where nested universal and existential quantifiers occur. The main problem is that performing quantifier elimination on formulas defined by, possibly recursive, Constrained Horn Clauses is very expensive, and often unsuccessful. Thus, we propose an alternative method that is based on verifying a series of simpler properties, where quantification is restricted to LIA constraints.

We assume the existence of a solver that is sound and complete for Horn clauses with LIA constraints. The solver interface is a procedure $solve(P, Q)$

such that, for any set P of CHCs and for any *query* Q, which is a conjunction of atoms and LIA constraints, returns an *answer* A, that is, a satisfiable constraint A such that $P \models \forall(A \to Q)$, if such an answer exists, and *false* otherwise.

The method we propose solves controllability problems by looking for a satisfiable constraint $a(U,C)$, where U and C are tuples of variables denoting the durations of the uncontrollable and controllable tasks, respectively, such that $I \cup \mathit{LIA} \models \forall U\, \forall C.\, a(U,C) \to \mathit{reachProp}(U,C)$ and either
(†) $\mathit{LIA} \models \exists C\, \forall U.\, \mathit{adm}(U) \to\ a(U,C)$ (for strong controllability), or
(‡) $\mathit{LIA} \models \forall U.\, \mathit{adm}(U) \to \exists C.\, a(U,C)$ (for weak controllability).

In particular, we introduce the Strong and Weak Controllability algorithms (SC and WC for short, respectively) that, given a set of function-free CHCs defining $\mathit{reachProp}(U,C)$ (that is, a set of clauses generated by the specialization transformation of Section 5), produce a solution for the controllability problem by constructing $a(U,C)$ as a disjunction of the answer constraints provided by the solver until either condition (†) holds (respectively, condition (‡) holds) or there are no more answers (see Figure 4). In order to avoid repeated answers, at each iteration of the *do-while* loop, the solver is invoked on a query containing the negation of the (disjunction of the) answers obtained so far.[4]

Since the durations of the tasks belong to finite integer intervals, the set of answers that can be returned by the *solve* procedure is finite. Hence the SC and WC algorithms always terminate. The following theorem states that SC and WC are sound and complete methods for solving strong and weak controllability problems, respectively.

Theorem 4 (Soundness and Completeness of SC and WC). *Let I be a set of CHCs defining a reachability property for a BPS $\mathcal{B}$. Then,*
(i) SC returns a satisfiable constraint if and only if $\mathcal{B}$ is strongly controllable
(ii) WC returns a satisfiable constraint if and only if $\mathcal{B}$ is weakly controllable.

Input: A set I of CHCs defining a reachability property.
Output: A satisfiable constraint $a(U,C)$, if the process is (strongly or weakly) controllable; *false* otherwise.

SC: *Strong Controllability*	WC: *Weak Controllability*
$a(U,C) := \mathit{false}$ *do* { $Q := (\mathit{reachProp}(U,C) \wedge \neg a(U,C))$; *if* $(\mathit{solve}(I,Q) = \mathit{false})$ *return false*; $a(U,C) := a(U,C) \vee \mathit{solve}(I,Q)$; } *while* $(\mathit{LIA} \not\models \exists C\, \forall U.\, \mathit{adm}(U) \to a(U,C))$ *return* $a(U,C)$;	$a(U,C) := \mathit{false}$ *do* { $Q := (\mathit{reachProp}(U,C) \wedge \forall C.\, \neg a(U,C))$; *if* $(\mathit{solve}(I,Q) = \mathit{false})$ *return false*; $a(U,C) := a(U,C) \vee \mathit{solve}(I,Q)$; } *while* $(\mathit{LIA} \not\models \forall U.\, \mathit{adm}(U) \to \exists C.\, a(U,C))$ *return* $a(U,C)$;

Fig. 4. The SC and WC algorithms for verifying strong and weak controllability.

We now illustrate how the WC algorithm works by applying it to the clauses obtained by specialization in Example 1. During the first iteration of the *do-while*

[4] In WC we introduce a small optimization by using a query that avoids obtaining multiple answers with the same values of U.

loop the CHC solver is invoked by executing $\mathit{solve}(I, \mathit{reachProp}(A1, A2, B) \wedge \forall A2, B. \neg \mathit{false})$, which returns the answer constraint $a1(A1, A2, B)$: $A1 \geq B - 2, A1 \leq 4, A2 = B - A1, B \geq 5, B \leq 6$. In our example, the constraint $\mathit{adm}(A1)$ is $A1 \geq 2, A1 \leq 4$. Now we have that $\mathit{LIA} \not\models \forall A1.\, \mathit{adm}(A1) \rightarrow \exists A2, B.\, a1(A1, A2, B)$, and hence the algorithm executes the second iteration of the *do-while* loop. Next, the CHC solver is invoked by executing $\mathit{solve}(I, \mathit{reachProp}(A1, A2, B) \wedge \forall A2, B.\, \neg a1(A1, A2, B))$, which returns the answer constraint $a2(A1, A2, B)$: $A1 = 2, A2 = 1, B = 6$. Now the condition of the *do-while* loop is false, because $\mathit{LIA} \models \forall A1.\, \mathit{adm}(A1) \rightarrow \exists A2, B.\, (a1(A1, A2, B) \vee a2(A1, A2, B))$. Thus, the WC algorithm terminates and we can conclude that the considered weak controllability property holds.

We have used the VeriMAP transformation and verification system for CHCs [13] to implement the specialization transformation of Section 5, and SICStus Prolog and the Z3 solver to implement the SC and WC algorithms. We have applied our method to verify the weak controllability of the process *Proc*. The timings are as follows[5]: the execution of the specialization transformation requires 0.04 seconds and the execution of the WC algorithm requires 0.03 seconds.

We have also solved controllability problems for other small-sized processes, not shown here for reasons of space, whose reachability relation, like the one for process *Proc*, contains cycles that may generate an unbounded proof search, and hence may cause nontermination if not handled in an appropriate way. In particular, in all the examples we have considered, we noticed that Z3 is not able to provide a proof within a time limit of one hour for a direct encoding of the controllability properties as they are formulated in Definition 4.

7 Related Work and Conclusions

Controllability problems arise in all contexts where the duration of some tasks in a business process cannot be determined in advance by the process designer. We have presented a method for checking strong and weak controllability properties of business processes. The method is based upon well-established techniques and tools in the field of computational logic.

Modeling and reasoning about time in the field of business process management has been largely investigated in the recent years [5]. The notion of controllability, extensively studied in the context of scheduling and planning problems over temporal networks [6,7,8,27,31,32], has been considered as a useful concept for supporting decisions in business process management and design [9,21,22].

Algorithms for checking strong and weak controllability properties were first introduced for Simple Temporal Networks with Uncertainty [32]. Later, sound and complete algorithms were developed for both strong [27] and weak [31] controllability of Disjunctive Temporal Problems with Uncertainty (DTPU). More

[5] The experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under GNU/Linux OS.

recently, a general and effective method for checking strong [7] and weak [8] controllability of DTPU's via SMT has been developed.

The task of verifying controllability of BP models we have addressed in this paper is similar to the task of checking controllability of temporal workflows addressed by Combi and Posenato [9]. These authors present a workflow conceptual framework that allows the designer to use temporal constructs to express duration, delays, relative, absolute, and periodic constraints. The durations of tasks are uncontrollable, while the delays between tasks are controllable. The controllability problem, which arises from relative constraints that limit the duration of two non-consecutive tasks, consists in checking whether or not the delays between tasks enforce the relative constraints for all possible durations of tasks. The special purpose algorithms for checking controllability presented in [9] enumerate all possible choices, and therefore are computationally expensive.

Our approach to controllability of BP models exhibits several differences with respect to the one considered by Combi and Posenato in [9]. In our approach the designer has the possibility of explicitly specifying controllable and uncontrollable durations. We also consider workflows with minimal restrictions on the control flow, and unlike the framework in [9], we admit loops. We automatically generate the clauses to be verified from the formal semantics of the BP model, thus making our framework easily extensible to other classes of processes and properties. Finally, we propose concrete algorithms for checking both strong and weak controllability, based on off-the-shelf CHC specializers and solvers.

As future work we plan to perform an extensive experimental evaluation of our method and to apply our approach to extensions of time-aware BP models, whose properties also depend on the manipulation of data objects.

References

1. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
2. B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. *Proc.* (*PODS '13*), pp. 163–174. ACM, 2013.
3. B. Berthomieu and F. Vernadat. Time Petri nets analysis with TINA. *Proc. QEST '06*, pp. 123–124. IEEE Computer Society, 2006.
4. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich*, LNCS 9300, pp. 24–51. Springer, 2015.
5. S. Cheikhrouhou, S. Kallel, N. Guermouche, and M. Jmaiel. The temporal perspective in business process modeling: a survey and research challenges. *Service Oriented Computing and Applications*, 9(1):75–85, 2015.
6. A Cimatti, L. Hunsberger, A. Micheli, R. Posenato, and M. Roveri. Dynamic controllability via timed game automata. *Acta Informatica*, 53(6):681–722, 2016.
7. A. Cimatti, A. Micheli, and M. Roveri. Solving strong controllability of temporal problems with uncertainty using SMT. *Constraints*, 20(1):1–29, 2015.
8. A. Cimatti, A. Micheli, and M. Roveri. An SMT-based approach to weak controllability for disjunctive temporal problems with uncertainty. *Artif. Intell.*, 224:1–27, 2015.

9. C. Combi and R. Posenato. Controllability in Temporal Conceptual Workflow Schemata. *Proc. BPM '09*, LNCS 5701, pp. 64–79. Springer, 2009.
10. E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Transactions on Database Systems*, 37(3):1–36, 2012.
11. E. De Angelis, F. Fioravanti, M. C. Meo, A. Pettorossi, and M. Proietti. Verification of time-aware business processes using Constrained Horn Clauses. *Preliminary Proc. LOPSTR'16*, CoRR, `http://arxiv.org/abs/1608.02807`, 2016.
12. E. De Angelis, F. Fioravanti, M. C. Meo, A. Pettorossi, and M. Proietti. Verifying controllability of time-aware business processes. Tech. Rep. IASI-CNR 16-08, 2016.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. *Proc. TACAS '14,* LNCS 8413, pp. 568–574. Springer, 2014.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. *Proc. PPDP '15*, pp. 91–102. ACM, 2015.
15. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *Proc. TACAS '08* LNCS 4963, pp. 337–340. Springer, 2008.
16. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
17. Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual. `http://www.fsel.com`, 1998.
18. A. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, and N. Russell Eds. *Modern Business Process Automation: YAWL and its Support Environment.* Springer, 2010.
19. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
20. R.A. Kowalski and M.J. Sergot. A Logic-based Calculus of Events. *New Generation Comput.*, 4(1):67–95, 1986.
21. A. Kumar, S.R. Sabbella, and R.R. Barton. Managing controlled violation of temporal process constraints. *BPM '15*, LNCS 9253, pp. 280–296. Springer, 2015.
22. A. Lanz, R. Posenato, C. Combi, and M. Reichert. Controlling time-awareness in modularized processes. *Proc. BPMDS/EMMSAD '16*, LNBIP 248, pp. 157–172. Springer, 2016.
23. K.G. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
24. M. Makni, S. Tata, M.M. Yeddes, and N. Ben Hadj-Alouane. Satisfaction and coherence of deadline constraints in inter-organizational workflows. *Proc. OTM '10*, LNCS 6426, pp. 523-539. Springer, 2010.
25. J. McCarthy and P.J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: B. Meltzer and D. Michie (eds.), *Machine Intelligence* 4, pp. 463–502, Edinburgh University Press, 1969.
26. OMG. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/`.
27. B. Peintner, K. B. Venable, and N. Yorke-Smith. Strong Controllability of Disjunctive Temporal Problems with Uncertainty. *Proc. CP '07*, LNCS 4741, pp. 856-863. Springer, 2007.
28. M. Proietti and F. Smith. Reasoning on data-aware business processes with constraint logic. *Proc. SIMPDA '14*, *CEUR*, Vol. 1293, pp. 60–75, 2014.
29. F. Smith and M. Proietti. Rule-based behavioral reasoning on semantic business processes. In: *Proc. ICAART '13, Vol. II*, pp. 130–143. SciTePress, 2013.
30. M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artif. Intell.*, 111(1-2):277-299, 1999.

31. K. B. Venable, M. Volpato, B. Peintner, and N. Yorke-Smith. Weak and dynamic controllability of temporal problems with disjunctions and uncertainty. *Proc. COPLAS '10* , pp. 50–59, 2010.
32. T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *J. Exp. Theor. Artif. Intell.*, 11(1):23-45, 1999.
33. K. Watahiki, F. Ishikawa, and K. Hiraishi. Formal verification of business processes with temporal and resource constraints. *Proc. SMC '11*, pp. 1173–1180. IEEE, 2011.
34. I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: On the verification of semantic business process models. *Distrib. Parallel Databases*, 27:271–343, 2010.
35. M. Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer, 2007.
36. P.Y.H. Wong and J. Gibbons. A relative timed semantics for BPMN. *Electronic Notes Theoretical Computer Science*, 229(2):59–75, 2009.

A Proofs of Section 3

The following lemma states that two derivation steps can be switched.

Lemma 1 (Switching Lemma). *Consider a state s, with two sets of fluents F and F' which activate two different (instances of the) rules in S_1–S_6. Suppose that $\delta = s \longrightarrow s_1 \longrightarrow s_2$ is a derivation where F are the rewritten fluents in s and F' are the rewritten fluents in s_1.*
Then there exists a derivation $\delta' = s \longrightarrow s'_1 \longrightarrow s_2$ where F' are the rewritten fluents in s and F are the rewritten fluents in s'_1. Moreover a fluent f occurs in δ iff f occurs in δ'.

Proof. The proof is immediate by observing that the premises of rules in S_1–S_6 are not overlapping, that is, the application of a rule in S_1–S_6 does not erase fluents for the application of a different rule in S_1–S_6.

Lemma 2. *Let δ be an infinite derivation. Then there exists a flow object x, such that $begins(x)$ is rewritten infinitely many times in δ.*

Proof. First let us observe that, since δ be an infinite derivation and the set of flow objects in a BPS is finite, there exists a flow object y, such that a fluent of the form either $begins(y)$, or $completes(y)$, or $enables(p, y)$, or $enacting(y, d)$ is rewritten infinitely many times in δ. If $begins(y)$ is rewritten infinitely many times, then we have the thesis by taking $x = y$. Otherwise $begins(y)$ is not rewritten infinitely many times. In this case neither $enacting(y, d)$ nor $completes(y)$ are rewritten infinitely many times in δ. This follows immediately by observing that:

– $enacting(y, d)$ is produced by S_7 from a set F of fluents including $enacting(y, d')$, with $d < d' \leq d_y$, where $duration(y, d_y)$ holds and by S_1 which has $begins(y)$ and $duration(y, d_y)$ as premise, and
– $completes(y)$ is produced by S_6, which has $enacting(y, 0)$ (and therefore by the previous point, $begins(y)$) as premise.

By previous observations, we have that $enables(p, y)$ is rewritten infinitely many times in δ and then we take $x = p$. Now, the lemma follows from the observation that $enables(p, y)$ has $completes(p)$ as premise (by using S_2 or S_3) and (analogously to the previous case) $completes(p)$ is produced by S_6, which has $enacting(p, 0)$ (and therefore $begins(p)$) as premise.

Lemma 3 (Finiteness Lemma). *Let δ be a derivation that uses rules S_1–S_6 only. Then δ is finite.*

Proof. The proof is by contradiction. Assume that there exists an infinite derivation δ which uses only the rules S_1–S_6. Then δ is of the form

$$\delta = \langle F_0, t\rangle \longrightarrow \ldots. \longrightarrow \langle F_n, t\rangle \longrightarrow \ldots.$$

since each rule in S_1–S_6 infers instantaneous state rewritings. The following points hold.

(i) There exists a flow object x such that the fluent $begins(x)$ is rewritten infinitely many times in δ. This is a consequence of the fact that by definition every BPS $\mathcal{B}$ is always finite, and therefore, since δ is infinite, there exists a flow object x such that a fluent of the form either $begins(x)$, or $completes(x)$, or $enables(p,x)$, or $enacting(x,d)$ is rewritten infinitely many times in δ. Then by Lemma 2 we have the thesis. In the following let us denote by o such a flow object.

(ii) There exists a gateway y (thus, $duration(y,0)$ holds) such that the fluent $begins(y)$ is rewritten infinitely many times in δ. If $duration(o,0)$ holds then we have the thesis by taking $y=o$. Otherwise, $duration(o,d)$ with $d>0$ (namely o is a task) and there exists a unique y such that $seq(y,o)$ holds. Since $begins(o)$ is rewritten infinitely many times in δ, we have that $enables(y,o)$ is rewritten infinitely many times in δ. Then by definition of the operational semantics $completes(y)$ is rewritten infinitely many times in δ and by Lemma 2, we have that $begins(y)$ is rewritten infinitely many times in δ. Moreover, since $completes(y)$ is rewritten infinitely many times in δ and δ uses only the rules S_1–S_6, we have that $enacting(y,0)$ is rewritten infinitely many times in δ (by using the rule S_6) and (by the rule S_1) $duration(y,0)$ holds. In the following let us denote by o' such a flow object. By definition o' is a gateway.

(iii) There exists a cycle made out of gateways only. Since $begins(o')$ is rewritten infinitely many times in δ, we have that there exists a flow object z such that $seq(z,o')$ holds and $enables(z,o)$ is rewritten infinitely many times in δ. We have the following two possibilities:
- $z=o'$. In this case there exists a cycle through the gateway o' and then we get the thesis.
- $z\neq o'$. The same argument of the previous point leads to conclude that z is a gateway and the fluent $begins(z)$ is rewritten infinitely many times in δ. By iterating this argument and since the number of gateway in every BPS $\mathcal{B}$ is always finite, we get the thesis.

Therefore $gateway\_path(o',o')$ holds. This contradicts the hypothesis that $\mathcal{B}$ is a BPS and the proof of the lemma is completed.

In the following, given the two derivations $\delta_1 = s_0 \longrightarrow s_1 \ldots \longrightarrow s_n$ and $\delta_2 = s_n \longrightarrow s_{n+1} \ldots \longrightarrow s_m$, we denote by $\delta_1 \cdot \delta_2$ the derivation $\delta_1 = s_0 \longrightarrow s_1 \ldots \longrightarrow s_n \longrightarrow s_{n+1} \ldots \longrightarrow s_m$.

Theorem 5. *For every derivation δ from a state s_0 ending in a state in $States_{sat}$, there exists a derivation δ' from s_0 via $\mathcal{R}$ such that: (i) δ and δ' end in the same state, and (ii) for each state $\langle F,t\rangle$ in δ such that $f\in F$, there exists a state $\langle F',t\rangle$ in δ' such that $f\in F'$.*

Proof. Let l the number of applications of rule S_7 in δ. The proof is by induction on l.

Case $l=0$. In this case δ uses the rules S_1–S_6 only and then it is of the form $s_0 \longrightarrow s_1 \ldots \longrightarrow s_n$, where $s_0=\langle F_0,t\rangle$ and $s_n=\langle F,t\rangle$.

Consider the first i such that in s_i the set of fluents selected in δ differs from the set of fluents F'_i selected by $\mathcal{R}$ in δ'. δ ends with a state in $States_{sat}$ and the rules are not overlapping, so for some $j>0$, F'_i is the set of fluents selected in the state s_{i+j} of δ. If such an i does not exists (namely δ is via $\mathcal{R}$), then we take $i=n$ and $j=0$.

Intuitively, i is the first place where δ deviates from the selection function $\mathcal{R}$ and j is the 'delay' of δ with respect to $\mathcal{R}$. We call $(n-i, j)$ the (deviate, delay) pair of δ with respect to $\mathcal{R}$. Now, we prove the claim by induction using the lexicographic ordering $\prec$ on the (deviate, delay) pairs, where $\prec$ is defined as usual, that is, $(k,j) \prec (h,\ell)$ iff $k<h$ or ($k=h$ and $j<\ell$).

If the (deviate, delay) pair is of the form $(0,0)$, then δ is via $\mathcal{R}$ and then we get the thesis. Otherwise, by Lemma 1, the $i+j$-th and $i+j-1$-th steps can be switched yielding a derivation δ'' from s_0 such that δ and δ'' end with the same state and a fluent f occurs in δ'' iff f occurs in δ. The (deviate, delay) pair of δ'' with respect to $\mathcal{R}$ is $(n-i, j-1)$ if $j>1$, and it is $(n-i-1, \ell)$, for some $\ell \geq 0$, if $j=1$. So, in both the cases the (deviate, delay) pair is below $(n-i, j)$ and then we get the thesis.

Case $l>0$. In this case δ is of the form $s_0 \longrightarrow s_1 \ldots \longrightarrow s_n \longrightarrow s_{n+1} \ldots \longrightarrow s_m$, where the number of applications of S_7 in $\delta_1 = s_0 \longrightarrow s_1 \ldots \longrightarrow s_n$ and in $\delta_2 = s_{n+1} \ldots \longrightarrow s_m$ is $l-1$ and 0, respectively and $s_n \longrightarrow s_{n+1}$ is obtained by applying S_7. By hypothesis $s_n \in \mathit{States}_{sat}$.

By inductive hypothesis there exists a there exists a derivation $\delta_1' = s_0 \longrightarrow s_1' \longrightarrow \ldots \longrightarrow s_n$ from s_0 via $\mathcal{R}$ such that: (i) δ_1 and δ_1' end in the same state and (ii) for each state $\langle F, t\rangle$ in δ_1 such that $f \in F$, there exists a state $\langle F', t\rangle$ in δ_1' such that $f \in F'$. Moreover, since $s_n \in \mathit{States}_{sat}$, only the rule S_7 can be applied by otaining the state s_{n+1} and then $\delta_1'' = \delta_1' \cdot s_n \longrightarrow s_{n+1}$ is a derivation via $\mathcal{R}$. Now, let $\mathcal{R}'$ be the selection rule such that for any derivation γ, $\mathcal{R}'$ returns the same set of fluents of $\mathcal{R}$ applied to $\delta_1'' \cdot \gamma$. By the previous point and since the number of application of rule S_7 in δ_2 is equal to 0, there exists a derivation $\delta_2' = s_{n+1} \ldots \longrightarrow s_m$ via $\mathcal{R}'$, such that: (i) δ_2 and δ_2' end in the same state and (ii) for each state $\langle F, t\rangle$ in δ_2 such that $f \in F$, there exists a state $\langle F', t\rangle$ in δ_2' such that $f \in F'$.

Then we get the thesis by previous results, since by construction $\delta_1'' \cdot \delta_2'$ is a derivation via $\mathcal{R}$.

Proof of Theorem 1

Proof. Without loss of generality, we assume that δ is of the form $s_0 \longrightarrow s_1 \ldots \longrightarrow s_n$, where for $i = 0, \ldots n$, $s_i = \langle F_i, t\rangle$. If $s_n \in \mathit{States}_{sat}$ then the proof follows by Theorem 5.

Otherwise, by Lemma 3, δ can be extended to a derivation δ_1 ending in a state in States_{sat}. Then, by Theorem 5, there exists a derivation δ' from s_0 via $\mathcal{R}$ such that for each state $\langle F, t\rangle$ in δ_1 such that $f \in F$, there exists a state $\langle F', t\rangle$ in δ', such that $f \in F'$ and then we get the thesis.

B Proofs of Termination of the Specialization Algorithm (Section 5)

In order to show the termination of the specialization algorithm we prove the following two lemmas.

Lemma 4. *For every BPS there exists a positive integer k such that, for every state $\langle F, t\rangle$ which is reachable from the initial state, we have that $|F| < k$.*

Proof. We use the assumption that the BPS is safe, that is, during its enactment there are no multiple concurrent executions of the same flow object. Let $\langle F, t\rangle$ be a state which is reachable from the initial state. By the safety assumption and the definition of the operational semantics, for every flow object x in F, there is at most one fluent $\mathit{enacting}(x, r)$, for some residual time r. The other fluents in which x may occur are among $\mathit{begins}(x)$, $\mathit{completes}(x)$, and $\mathit{enables}(x, y)$, where y is a successor of x. Thus, the thesis follows from the fact that in any BPS there are finitely many flow objects. □

Definition 5. *Let m be a non-negative integer. A constraint $g(X_1, \ldots, X_n)$ is* bounded *by m if, for $i = 1, \ldots, n$, $g(X_1, \ldots, X_n) \sqsubseteq 0 \leq X_i \leq m$.*

Lemma 5. *All new definitions introduced by the specialization algorithm are clauses of the form :*
$\quad \mathit{newr}(\mathit{Rs}, T, \mathit{Tf}, U, C) \leftarrow f(\mathit{Rs}),\ \mathit{reach}(s(\mathit{fl}(\mathit{Rs}), T), \mathit{fin}(\mathit{Tf}), U, C)$
where $f(\mathit{Rs})$ is bounded by the largest value among the maximal task durations specified by I.

Proof. Let m be the largest value among the maximal task durations specified by I. The variables in the tuple Rs will be called the *residual time variables.*

A constraint on a residual time variable can be introduced during the UNFOLDING phase by unfolding a *tr* atom using a clause among $C1$, $C6$, and $C7$, and then unfolding the atoms occurring in the bodies of these clauses.

If the *tr* atom is unfolded using $C1$, then the subsequent unfolding of the atom $\mathit{duration}(X, D)$ will add a constraint of the form $d_{min} \leq D \leq d_{max}$ on the residual time variable D. This constraint is bounded by m, as d_{max} is the maximal task duration of a task specified by I.

If the *tr* atom is unfolded using $C6$, then the constraint $R = 0$ is added to the residual time variable R. Clearly, this constraint is bounded by m.

The case where the *tr* atom is unfolded using $C7$ is slightly more elaborated. Suppose that the body of the clause to be unfolded contains a constraint $b(R_1, \ldots, R_n)$, where $R_1, \ldots, R_n$ are residual time variables, and $b(R_1, \ldots, R_n)$ is bounded by m. By unfolding the clause with respect to *tr* using $C7$, and subsequently with respect to the atoms in the body of $C7$, we derive (zero or more) clauses whose body has a constraint of the following form, for some $j \in \{1, \ldots, n\}$:

$$b(R_1, \ldots, R_n), R_j > 0, R_j \leq R_1, \ldots, R_j \leq R_n, R'_1 = R_1 - R_j, \ldots, R'_n = R_n - R_j \qquad (\dagger)$$

The constraint $R_j \leq R_1, \ldots, R_j \leq R_n$ is derived by unfolding the *mintime* atom and the constraint $R'_1 = R_1 - R_j, \ldots, R'_n = R_n - R_j$ is derived by unfolding the *decrease_residual_times* atom. Since $b(R_1, \ldots, R_n)$ is bounded by m, we have that, for $i = 1, \ldots, n$, the constraint (†) entails $R'_j \leq m$, and hence (†) is bounded by m.

Thus, every constraint on residual time variables which is derived during the UNFOLDING phase is bounded by m. The thesis follows from the facts that the constraint $f(\mathit{Rs})$ is derived by projection onto the residual time variables Rs and this projection operation preserves boundedness. □

We are now ready to show the termination of the specialization algorithm.

Proof. We need to show that the outer loop of the specialization algorithm terminates, that is, the new predicate definitions introduced in *Defs* by the algorithm constitute a finite set. Every new definition is a clause of the form:

$\quad newr(Rs, T, Tf, U, C) \leftarrow f(Rs),\ reach(s(fl(Rs), T), fin(Tf), U, C)$

By Lemma 4, the atom $reach(s(fl(Rs), T), fin(Tf), U, C)$ is taken from the finite set (modulo variable names) such that $fl(Rs) < k$, where k is an integer bound that depends on the BPS specified by I. (Recall that the term $fl(Rs)$ encodes the set F of fluents in a state $\langle F, t \rangle$.)

By Lemma 5, *f(Rs)* is bounded by the greatest upper bound m of the task duration intervals specified by I, and the set of pairwise non-equivalent constraints bounded by m is finite. Since the specialization algorithm does not introduce two new definitions with equal *reach* atoms and equivalent constraints, the set of new predicates introduced in *Defs* is finite. □

C Proof of Soundness and Completeness of the Controllability Algorithms

Proof. We present the proof for SCC. The proof for WCC is similar, and we omit it.

Soundness (1) *If SCC returns a satisfiable constraint $a(U, C)$ then $\mathcal{B}$ is strongly controllable.*

If SCC returns a satisfiable constraint $a(U, C)$ then the exit condition of the *do-while* loop guarantees that $LIA \models \exists C\, \forall U.\, adm(U) \rightarrow a(U, C)$. Since $a(U, C)$ is a disjunction of answers to queries containing $reachProp(U, C)$ as a conjunct, we have that $I \cup LIA \models \forall U\, \forall C.\, a(U, C) \rightarrow reachProp(U, C)$. Thus, we have that $I \cup LIA \models \exists C\, \forall U.\, adm(U) \rightarrow reachProp(U, C)$, and hence $\mathcal{B}$ is strongly controllable.

Soundness (2) *If SCC returns false, then $\mathcal{B}$ is not strongly controllable.*

If SCC returns *false*, then let Q be the query of the form $reachProp(U, C) \wedge \neg a(U, C)$ such that $solve(I, Q) = false$. By the condition of the *do-while* loop we have that: (*) $LIA \not\models \exists C\, \forall U.\, adm(U) \rightarrow a(U, C)$. Let us assume, by contradiction, that $\mathcal{B}$ is strongly controllable, that is, $I \cup LIA \models \exists C\, \forall U.\, adm(U) \rightarrow reachProp(U, C)$. By the completeness of the CHC solver, we have that $I \cup LIA \models \forall U\, \forall C.\, a(U, C) \leftrightarrow reachProp(U, C)$, and hence $LIA \models \exists C\, \forall U.\, adm(U) \rightarrow a(U, C)$ which contradicts (*).

Completeness (1) *If $\mathcal{B}$ is strongly controllable, then SCC returns a satisfiable constraint $a(U, C)$.*

Suppose that $\mathcal{B}$ is strongly controllable. Let us assume, by contradiction, that SCC returns *false*. Then, by soundness of SCC, we have that $\mathcal{B}$ is not strongly controllable, which contradicts the hypothesis.

Completeness (2) *If $\mathcal{B}$ is not strongly controllable, then SCC returns false.*

Suppose that $\mathcal{B}$ is not strongly controllable. Let us assume, by contradiction, that SCC returns a satisfiable constraint $a(U, C)$. Then, by soundness of SCC, we have that $\mathcal{B}$ is strongly controllable, which contradicts the hypothesis. □