Fundamenta Informaticae Submitted (2018) 1–39
DOI 10.3233/FI-2012-0000
IOS Press

Semantics and Controllability of Time-Aware Business Processes*

Emanuele De Angelis, Fabio Fioravanti, Maria Chiara Meo
University of Chieti-Pescara, Viale Pindaro 42, 65127, Pescara, Italy,
{emanuele.deangelis,fabio.fioravanti,cmeo}@unich.it

Alberto Pettorossi
University of Rome Tor Vergata, Via del Politecnico 1, 00133 Rome, Italy,
pettorossi@info.uniroma2.it

Maurizio Proietti
IASI-CNR, Via dei Taurini 19, 00185 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 executes 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 execution 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 means 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, also called tasks, in a business process (or BP, for short) and of the way these

*This work has been partially supported by the National Group of Computing Science (GNCS-INdAM).
E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at IASI-CNR, Rome, Italy. All authors are members of the INdAM Research group GNCS.

activities must coordinate to achieve a goal [20, 40]. Many notations for BP modeling, and in particular the popular BPMN [30], 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 [20], which focuses on the control flow only. Thus, formal reasoning about time related properties, which may be very important in many applications, is not possible in that context.

In order to overcome this difficulty, various approaches to BP modeling with time constraints have been proposed in the literature (see [6] 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* [27], *timed automata* [38], and *process algebras* [41]. Properties of these models can then be verified by using the effective reasoning tools that are available for those formalisms [3, 19, 26].

In this paper we address the problem of verifying the *controllability* of time-aware business processes. This notion has been introduced for dealing with scheduling and planning problems over *Temporal Networks* [37], but it has not received much attention in the more complex case of time-aware BP models. We assume that some of the durations of the tasks of the process are *controllable*, that is, they can be determined by the organization that executes (or *enacts*) the process, while other durations are *uncontrollable*, that is, they are determined by the external world. The properties of *weak controllability* and *strong controllability*, guarantee, in two different senses, that all process tasks 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 [11]) where the completion of the whole process within a certain deadline must be guaranteed, even if the exact durations of some of its tasks cannot be 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 [13]. This approach represents both the BP structure and the BP semantics in terms of *Constrained Horn Clauses* (CHCs) [4], also known as *Constraint Logic Programs* [21], 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 are defined in terms of properties, called *reachability properties*, which establish the possibility for a process to complete its execution under some given time constraints. More specifically, controllability will be defined in terms of quantified reachability properties.

An advantage of the logic-based approach with respect to other approaches is that it allows a smooth integration of the various forms of reasoning needed to analyze business processes from different perspectives. These forms of reasoning include, for instance, (i) the ontology-related reasoning on the business domain where processes are executed [34, 39], and (ii) the reasoning on the manipulation of data objects such as databases or integer values [2, 12, 33]. Moreover, in order to perform those kinds of logic-based reasoning, one can make use of effective tools such as CHC solvers [16] and Constraint Logic Programming systems.

For reasons of simplicity, in this paper we consider business process models where the only time-related entities are the constraints that task durations should satisfy. However, by following a similar approach also other entities which refer to time can be modeled, if so desired.

The main contributions of this paper are the following. (1) We define and study the basic properties of a novel operational semantics for *safe* time-aware business process models [1]. This

semantics is a variant of the one presented in [13] (see Section 3). In particular, we provide a new formalization of the synchronization occurring at the parallel merge gateways. (2) We provide the formal definitions of weak and strong controllability properties using 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 the weak and the strong controllability problem, respectively, for time-aware business processes. 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 [16] (see Section 6).

2. Preliminaries

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

Let *RelOp* be the set $\{=, \neq, \leq, \geq, <, >\}$ of predicate symbols denoting the usual relations 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*). Note that constraints are closed under negation. 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* over the constraint theory *LIA* (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.

A set S of CHCs is said to be *satisfiable* if $S \cup \mathit{LIA}$ has a *LIA*-model (that is, a model where the function symbols and predicate symbols of *LIA* are interpreted as expected), 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. Recall that the satisfiability of quantified *LIA* formulas is decidable [5]. 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, $c \sqsubseteq c'$. This notion of a projection is adequate for our use in the specialization algorithm (see Section 5), and it will be sufficient to prove the correctness of that algorithm [18].

A BPMN model of a business process consists of a diagram drawn by using a graphical notation for depicting: (i) *flow objects*, and (ii) *sequence flows* (also called *flows*, for short) [30].

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 that connects a flow object, called the *source*, to a flow object, called the *target* (see Figure 1).

Tasks are atomic units of work that are performed during the execution, also called the *enactment*, 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 $g2$ in Figure 1),

(b) the *exclusive branch*, that (non-deterministically) activates exactly one of its outgoing flows, if its single incoming flow is activated (see $g4$ in Figure 1),

(c) the *parallel merge*, that activates the single outgoing flow, if all the incoming flows are simultaneously activated (see $g3$ in Figure 1), and

(d) the *exclusive merge*, that activates the single outgoing flow, if any one of the incoming flows is activated (see $g1$ in Figure 1).

The diamonds representing parallel gateways (a) and (c) are labeled '+', and the diamonds representing exclusive gateways (b) and (d) are labeled by '×'.

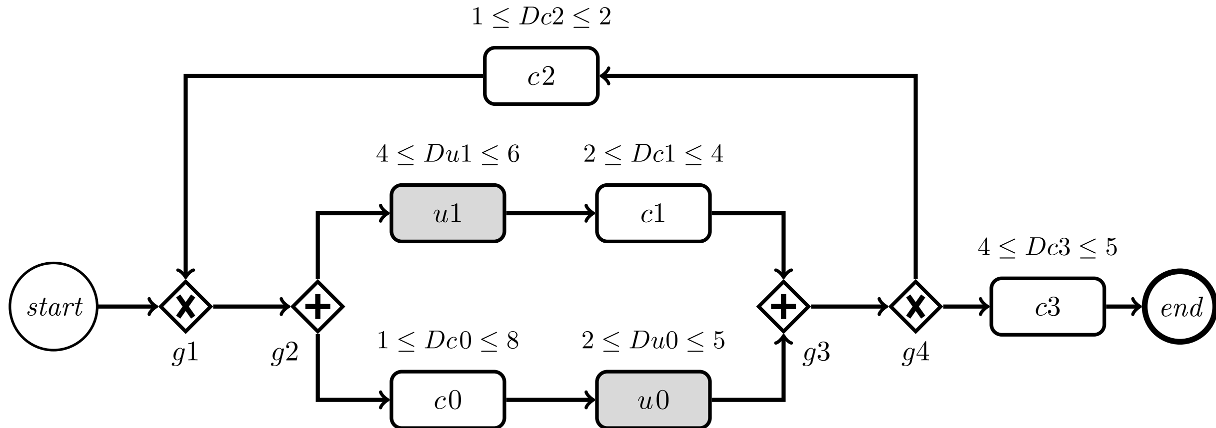

Figure 1. A business process *CarDealer*. Tasks $u0$ and $u1$ (with grey background) have uncontrollable durations. The other tasks $c0$–$c3$ (with white background) have controllable durations.

Given a business process, a sequence flow denotes the fact that the execution of the process can pass from the source object to the target object. If there is a sequence flow from an object u to an object v, then u is said to be a *predecessor* of v, and symmetrically, v is said to be a *successor* of u. A *path* is a sequence of flow objects such that every pair of consecutive objects in the sequence is connected by a sequence flow. A *cycle* is a path with at least two flow objects (not necessarily distinct) such that the first flow object of the path is equal to the last flow object of the path.

In Figure 1 we show the BPMN model of a business process, called *CarDealer*, of a car dealer

company. The company buys cars from a supplier and sets them up before shipping them to the customers together with certificates of compliance. More than one car may be shipped during a single execution of the process. After the *start* event, the exclusive merge $g1$ activates the parallel branch $g2$, which in turn activates the tasks $u1$ (*Buy-Car*) and $c0$ (*Prepare-Documents*). When the tasks $c1$ (*Setup-Car*) and $u0$ (*Obtain-Certificate*) have terminated, the parallel merge $g3$ activates the exclusive branch $g4$ which activates either (i) the task $c2$ (*Get-Car-Orders*) or (ii) the task $c3$ (*Load-on-Carrier*). In Case (i), at the completion of the task $c2$ a new instance of the task $u1$ and a new instance of the task $c0$ are activated (via the gateways $g1$ and $g2$) and the execution of the process proceeds as indicated above. In Case (ii), the task $c3$ is activated and at its completion the whole process *CarDealer* terminates with the *end* event.

We will consider models of business processes that are *well-formed*, in the sense that they satisfy the following properties: (1) every business process is made out of a *finite* number of flow objects, and for any flow object there is only a *finite* number of incoming and outgoing sequence flows, (2) for every sequence flow, the source and the target flow objects are distinct, (3) there is a single *start* event having exactly one successor and no predecessors, and there is a single *end* event having exactly one predecessor and no successors, (4) every flow object occurs in a path from the *start* event to the *end* event, (5.1) parallel branch gateways and exclusive branch gateways have exactly one predecessor and at least one successor, (5.2) parallel merge gateways and exclusive merge gateways have at least one predecessor and exactly one successor, (5.3) tasks have exactly one predecessor and one successor, and (6) there are no cycles whose flow objects are gateways only.

Note that business process models need not be block-structured. This means that, for instance, in a model whose flow objects are a, b_1, b_2, and c, and whose set of paths includes the paths $a \to b_1 \to c$ and $a \to b_2 \to c$, there may be the sequence flow $b_1 \to b_2$.

3. Specification and Semantics of Business Processes

In this section we introduce the notion of a *Business Process Specification*, which formally represents a business process by means of CHCs (see Section 3.1), we define the operational semantics of a business process (see Section 3.2), and we state some of its basic properties (see Section 3.3).

3.1. Business Process Specification via CHCs

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

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 irreflexive predicate $\mathit{seq}(X,Y)$ meaning that there is a sequence flow from X to Y.

For every task X we specify its duration D by a constrained fact of the form $\mathit{duration}(X,D) \leftarrow d_{min} \leq D \leq d_{max}$, where d_{min} and d_{max}, with $d_{min} \leq d_{max}$, are positive integer constants repre-

senting the minimal and the maximal duration of X, respectively. Events and gateways, being instantaneous, are assumed to have duration 0. (Note that, on the contrary, tasks have positive duration.) For every task X we also specify whether or not its duration is controllable by stating the fact either $controllable(X) \leftarrow$ or $uncontrollable(X) \leftarrow$, respectively. We will say that a task is controllable (or uncontrollable) if its duration is controllable (or uncontrollable, respectively).

In Figure 2 we show the constrained Horn clauses which specify process *CarDealer* of Figure 1.

$$
\begin{array}{llll}
task(c0) \leftarrow & task(c1) \leftarrow & task(c2) \leftarrow & task(c3) \leftarrow \\
task(u0) \leftarrow & task(u1) \leftarrow & event(start) \leftarrow & event(end) \leftarrow \\
exc_merge(g1) \leftarrow & par_branch(g2) \leftarrow & par_merge(g3) \leftarrow & exc_branch(g4) \leftarrow \\
seq(start, g1) \leftarrow & seq(g1, g2) \leftarrow & seq(g2, u1) \leftarrow & seq(u1, c1) \leftarrow \\
seq(c1, g3) \leftarrow & seq(g2, c0) \leftarrow & seq(c0, u0) \leftarrow & seq(u0, g3) \leftarrow \\
seq(g3, g4) \leftarrow & seq(g4, c3) \leftarrow & seq(g4, c2) \leftarrow & seq(c2, g1) \leftarrow \\
seq(g4, end) \leftarrow & uncontrollable(u0) \leftarrow & uncontrollable(u1) \leftarrow & \\
controllable(c0) \leftarrow & controllable(c1) \leftarrow & controllable(c2) \leftarrow & controllable(c3) \leftarrow
\end{array}
$$

$$
\begin{array}{ll}
duration(u0, Du0) \leftarrow 2 \leq Du0 \leq 5 & duration(u1, Du1) \leftarrow 4 \leq Du1 \leq 6 \\
duration(c0, Dc0) \leftarrow 1 \leq Dc0 \leq 8 & duration(c1, Dc1) \leftarrow 2 \leq Dc1 \leq 4 \\
duration(c2, Dc2) \leftarrow 1 \leq Dc2 \leq 2 & duration(c3, Dc3) \leftarrow 4 \leq Dc3 \leq 5 \\
duration(X, D) \leftarrow event(X),\ D = 0 & duration(X, D) \leftarrow gateway(X),\ D = 0
\end{array}
$$

Figure 2. The CHCs of the Business Process Specification of *CarDealer* of Figure 1.

3.2. Operational Semantics of Business Processes

We will define the operational semantics of a business process under the assumption that the process is *safe*, that is, during its execution there are no multiple, simultaneous executions of the same flow object [1] (we will formalize the *safeness* notion in Section 3.3.) As a consequence of this assumption, we will represent the *state* of a process, during its execution, as a *set* (not a *multiset*) of facts, called *fluents*, holding at a time instant. We borrow the notion of a fluent from *action languages* such as the *Situation Calculus* [29], the *Event Calculus* [23], or the *Fluent Calculus* [35], and we will present our operational semantics by means of rules that define a transition relation, denoted '$\longrightarrow$', between states, as often done in the theory of programming languages. We will assume that time is *discrete* and the first time instant is 0.

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

$$f \ ::= \ begins(x) \ \mid \ enacting(x, r) \ \mid \ completes(x) \ \mid \ enables(x, y)$$

where: (i) $begins(x)$ represents the beginning of the enactment of the flow object x, (ii) $enacting(x,r)$ represents the enactment of the flow object x which at the current instant requires r extra units of time to complete (for this reason r is called the *residual time* of x), (iii) $completes(x)$ represents the completion of the enactment of the flow object x, and (iv) $enables(x,y)$ represents that the flow object x has completed its enactment and it enables the enactment of its successor y. We have that $begins(x)$ is equivalent to $enacting(x, d)$, where d is the duration of x, and

$enacting(x,0)$ is equivalent to $completes(x)$. This redundant representation of the enactment of an object allows us to write simpler rules for the operational semantics (see rules S_1–S_7 below).

The operational semantics is defined by means of a transition relation '$\longrightarrow$' which is a subset of $States \times States$. We will refer to this relation also as a *rewriting* relation when we want to stress that the set fluents of the current state is rewritten into a new set of fluents in the new state. The relation '$\longrightarrow$' is generated by the rules S_1–S_7 listed below. In these rules we use the predicates introduced in Section 3.1 and also the following two predicates: (i) $not_par_branch(x)$, which holds if the flow object x is *not* a parallel branch, and (ii) $not_par_merge(x)$, which holds if the flow object x is *not* a parallel merge.

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

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

$$(S_3) \quad \frac{not_par_branch(x) \qquad completes(x) \in F \qquad seq(x, s)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{completes(x)\}) \cup \{enables(x, s)\},\ t\rangle}$$

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

$$(S_5) \quad \frac{not_par_merge(x) \qquad enables(p, x) \in F}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{enables(p, x)\}) \cup \{begins(x)\},\ t\rangle}$$

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

$$(S_7) \quad \frac{no_other_premises(F) \qquad \exists x\, \exists r\ enacting(x, r) \in F \qquad m > 0}{\langle F, t\rangle \longrightarrow \langle (F \setminus (Enacts \cup Enbls)) \cup (Enacts \ominus m),\ t + m\rangle}$$

where: (i) $no_other_premises(F)$ holds iff none of the rules S_1–S_6 can be applied in a state whose set of fluents is F, (ii) $m = \min\{r \mid enacting(x, r) \in F\}$, (iii) $Enacts = \{enacting(x, y) \mid enacting(x, y) \in F\}$, (iv) $Enbls = \{enables(x, y) \mid enables(x, y) \in F\}$, and (v) $Enacts \ominus m$ denotes the set $Enacts$, where every $enacting(x, y)$ has been replaced by $enacting(x, y - m)$.

From the definitions of rules S_1–S_6 we have that $no_other_premises(F)$ holds iff the following holds (we have indicated to the right of each conjunct the rule whose premise does not hold):

$$\begin{array}{lr}
\forall x.\ \ begins(x) \notin F & [S_1] \\
\quad \wedge \text{ if } par_branch(x) \text{ then } completes(x) \notin F & [S_2] \\
\qquad\qquad\qquad\qquad \text{else } completes(x) \notin F \vee \forall s. \neg seq(x, s) & [S_3]
\end{array}$$

$$\begin{array}{lll} \wedge \text{ if } \mathit{par_merge}(x) & \text{then } \exists p.\, \mathit{seq}(p,x) \wedge \mathit{enables}(p,x) \notin F & [S_4] \\ & \text{else } \forall p.\, \mathit{enables}(p,x) \notin F & [S_5] \\ \wedge\ \mathit{enacting}(x,0) \notin F & & [S_6] \end{array}$$

In the conjunct for rule S_3 we have that $\forall s. \neg seq(x, s)$ iff $x = end$, because for the business processes we consider the only flow object with no successor is the *end* event.

We assume that, for every flow object x, there exists a *unique duration*, denoted d_x, such that the atom $duration(x, d_x)$ holds. (This assumption is relevant for tasks only, because by definition the duration of events and gateways is 0.) For every flow object x, the value d_x is specified via a constraint fact (see Figure 2), and in every application of an instance of rule S_1, the same value d_x is used. Note that S_7 is the only rule that formalizes the passing of time, as it allows the generation of state rewritings of the form $\langle F, t\rangle \longrightarrow \langle F', t+m\rangle$, with $m > 0$. In contrast, rules S_1–S_6 generate state rewritings of the form $\langle F, t\rangle \longrightarrow \langle F', t\rangle$, where time does not pass. Here is a brief 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 with the residual time d_x which is the duration of x;

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

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

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

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

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

(S_7) Suppose that: (i) none of the rules S_1–S_6 can be applied to generate a state rewriting of the form $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$, (ii) at time t at least one task is enacting with *positive* residual time (note that flow objects different from tasks do not 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 $enables(p, s)$ fluents are removed. Note that, after the application of rule S_7, there exists at least one task x such that $enacting(x, 0) \in F'$, and thus after rule S_7, we can apply rule S_6.

Due to rules S_4 and S_5, if a fluent of the form $enables(p, s)$ 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 at time t if and only if it gets *simultaneously* enabled at time t by all its predecessors. In this sense rules S_1–S_7 formalize the *synchronous* semantics of the parallel merge gateway.

Note that the operational semantics of the parallel merge gateway can be provided in a different manner, by stipulating that its execution takes place when and only when all its predecessors are enabled, but not necessarily at the same time t. For this different, *asynchronous*

semantics of the parallel merge the reader may refer to a previous paper of ours [13]. It is not difficult to see that the asynchronous semantics may be obtained from the synchronous semantics we have formalized above by rules $S1$–$S7$, by adding to the business process under consideration suitable extra tasks whose durations realize delays.

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 $\langle \{ begins(start) \}, 0 \rangle$ (recall that time starts at the time instant 0).

A *final state* is any state of the form $\langle \{ completes(end) \}, t \rangle$, for some time instant t.

3.3. Properties of the Operational Semantics

Having defined in the previous section the operational semantics of a business process by means of a rewriting relation '$\longrightarrow$', now we will introduce four notions associated with that relation, namely, the notions of: (i) a derivation, (ii) a removed set, (iii) an added set, and (iv) a selection function. Then, we will prove an important property of the derivations (see Theorem 3.7 below).

Definition 3.1. (Derivation)
A derivation from a state s_0 is a (possibly infinite) sequence $s_0 \longrightarrow \ldots \longrightarrow s_i \longrightarrow s_{i+1} \longrightarrow \ldots$ of states, also written as $s_0 \longrightarrow^* s_i \longrightarrow s_{i+1} \longrightarrow \ldots$ We say that a derivation *uses rules* S_1–S_n, for some n, with $1 \leq n \leq 7$, if each of its rewritings is generated by an instance of one of the rules in the set $\{S_1, \ldots, S_n\}$.

Definition 3.2. (Removed set and added set)
Every rewriting generated by an instance σ of the rule S_k, with $1 \leq k \leq 7$, is of the form $\langle F, t \rangle \longrightarrow \langle (F \setminus R) \cup S, t' \rangle$, where R and S are called the *removed* set and the *added set*, respectively, of that rewriting. The sets R and S are non-empty sets uniquely determined by the conclusion of the instance σ (see the listing of rules S_1–S_7 above). A fluent in R (or in S) is said to be a *removed fluent* (or an *added fluent*, respectively).

Example 3.3. If the rewriting $\langle F, t \rangle \longrightarrow \langle (F \setminus R) \cup S, t \rangle$ is generated by the instance of rule S_4 which binds the free variable x to the flow object a, then in that rewriting the removed set R is $\{ enables(y, a) \mid enables(y, a) \in F \}$ and the added set S is $\{ begins(a) \}$.

In any rewriting of the form $\langle F, t \rangle \longrightarrow \langle (F \setminus R) \cup S, t' \rangle$, where R is the removed set and S is the added set, we have that $R \subseteq F$ and $R \cap S = \emptyset$.

Now we can formalize the notion of a safe process, which has been introduced at the beginning of Section 3.2. A process is said to be *safe* iff for every rewriting of the form $\langle F, t \rangle \longrightarrow \langle F', t' \rangle$, with $F' = (F \setminus R) \cup S$, where R and S are the removed set and the added set, respectively, we have that: (i) $F \cap S = \emptyset$, and (ii) it does not exist any flow object x such that $\{ enacting(x, r1), enacting(x, r2) \} \subseteq F'$, for two distinct integers $r1$ and $r2$. Thus, in particular, every fluent of a state s of a safe process is not an added fluent in any rewriting of the form $s \longrightarrow s'$, for some state s'.

Definition 3.4. (Selection function)
Let δ be a derivation of the form $s_0 \longrightarrow^* \langle F, t \rangle$, with $F \neq \emptyset$. A selection function $\mathcal{R}$ is a function that takes δ and returns: *either* (i) the empty set, if no state s' exists such that $\langle F, t \rangle \longrightarrow s'$,

(that is, the derivation δ cannot be extended) *or* (ii) the removed set of a rewriting $\langle F, t\rangle \longrightarrow s'$, if at least one such rewriting exists (that is, the derivation δ can be extended into the longer derivation $s_0 \longrightarrow^* \langle F, t\rangle \longrightarrow s'$).

Definition 3.5. (Derivation via a selection function)
We say that a derivation δ is via a selection function $\mathcal{R}$ iff for each proper prefix γ of δ, where γ is of the form $s_0 \longrightarrow^* \langle F, t\rangle$, *if* $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$, and the derivation $s_0 \longrightarrow^* \langle F, t\rangle \longrightarrow \langle F', t'\rangle$ is a prefix of δ, *then* $\mathcal{R}(\gamma)$ is the removed set of $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$.

From the definition of the rules we have the following.

Fact 3.6. Every non-empty derivation via any selection function that starts from the initial state is generated by a sequence of applications of the rules S_1–S_7 which in the regular expression notation is denoted by $(S_1 + \ldots + S_6)^+\,(S_7\,(S_1 + \ldots + S_6)^+)^*$.

Proof:
It is enough to note that: (i) the only fluent in the initial state is *begins*(*start*), (ii) rule S_7 can be applied only if none of the rules in S_1–S_6 can be applied, and (iii) after any rewriting $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$ generated by an application of rule S_7, one can apply rule S_6 because, by construction, at least one fluent of the form $enacting(x, 0)$ belongs to F'. □

Now we show that, given an initial state s_0 and a selection function $\mathcal{R}$, it may exist more than one derivation from s_0 via $\mathcal{R}$. Let us assume that in the last state $\langle F, t\rangle$ of a derivation γ from s_0, we have that $\mathcal{R}(\gamma) = completes(a)$, where a is an exclusive branch gateway. On that last state we can apply the instance of rule S_3 where x is a and s is a successor b of a (that is, $seq(a,b)$ holds), and we get the rewriting $\langle F, t\rangle \longrightarrow \langle F', t\rangle$, where the new fluent $enables(a, b)$ belongs to F'. Now, if a has also a different successor b' (that is, $seq(a,b')$ holds, for b' different from b), then by using rule S_3, we may also get the rewriting $\langle F, t\rangle \longrightarrow \langle F'', t\rangle$, where the new fluent $enables(a, b')$ belongs to F'' and $enables(a, b)$ does not belong to F'', and thus we can have two different extensions of γ via the same $\mathcal{R}$.

Note that for any given state and removed set R, there exists at most one instance of a rule in the set $\{S_1, \ldots, S_7\} \setminus \{S_3\}$ which, when applied to the given state, generates a rewriting whose removed set is R, and thus there exists at most one next state.

We have the following theorem, whose proof is given in Appendix A.

Theorem 3.7. Let $\overline{\mathcal{R}}$ be a fixed selection function. For every derivation δ from a state s via any selection function $\mathcal{R}$, there exists a derivation $\overline{\delta}$ from s via $\overline{\mathcal{R}}$ such that: (i) if a state $\langle F, t\rangle$ occurs in δ and $f \in F$, then there exists a state $\langle \overline{F}, t\rangle$ in $\overline{\delta}$ such that $f \in \overline{F}$, and (ii) if the rewriting $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$, with $t < t'$, occurs in δ, then the same rewriting occurs in $\overline{\delta}$.

4. Encoding Controllability Properties into CHCs

In this section we show how the operational semantics is translated into CHCs by defining a CHC *interpreter*, that is, a set of CHCs that encode the operational semantics of business processes. The CHC interpreter defines a predicate *reach*, encoding state reachability, which is then used for formalizing the weak and strong controllability properties.

4.1. Encoding the Operational Semantics in CHCs

A state of the operational semantics is represented 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. (Here we use the Prolog notation and we use capital letters as initials of variable names.) The rewriting relation '$\longrightarrow$' between states and its reflexive, transitive closure '$\longrightarrow^*$' are encoded by the predicates *tr* and *reach*, respectively. The definitions of these predicates are shown in Table 4.1, where clauses C_1–C_7 encode rules S_1–S_7, respectively.

C_1. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $select(\{begins(X)\}, F),\ task_duration(X, D, U, C),$
  $update(F, \{begins(X)\}, \{enacting(X, D)\}, FU)$

C_2. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $par_branch(X),\ select(\{completes(X)\}, F),\ findall(enables(X, S), seq(X, S), Enbls),$
  $update(F, \{completes(X)\}, Enbls, FU)$

C_3. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $not_par_branch(X),\ select(\{completes(X)\}, F),\ seq(X, S),$
  $update(F, \{completes(X)\}, \{enables(X, S)\}, FU)$

C_4. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $par_merge(X),\ findall(enables(P, X), seq(P, X), Enbls),\ select(Enbls, F),$
  $update(F, Enbls, \{begins(X)\}, FU)$

C_5. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $not_par_merge(X),\ select(\{enables(P, X)\}, F),$
  $update(F, \{enables(P, X)\}, \{begins(X)\}, FU)$

C_6. $tr(s(F,T), s(FU,T), U, C) \leftarrow$
  $select(\{enacting(X, 0)\}, F),\ update(F, \{enacting(X, 0)\}, \{completes(X)\}, FU)$

C_7. $tr(s(F,T), s(FU,TU), U, C) \leftarrow$
  $no_other_premises(F),\ member(enacting(X1, Res1), F),$
  $findall(Y, (Y = enacting(X, Res), member(Y, F)), Enacts),\ mintime(Enacts, M),\ M > 0,$
  $findall(Z, (Z = enables(P, S), member(Z, F)), Enbls),$
  $set_union(Enacts, Enbls, EnactsEnbls),\ decrease_residual_times(Enacts, M, EnactsU),$
  $update(F, EnactsEnbls, EnactsU, FU),\ TU = T + M$

R_1. $reach(S, S, U, C) \leftarrow$
R_2. $reach(s(F0, T0), s(F2, T2), U, C) \leftarrow T1 \leq T2,\ tr(s(F0, T0), s(F1, T1), U, C),$
  $reach(s(F1, T1), s(F2, T2), U, C)$

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

Let us briefly describe the various predicates used in clauses C_1–C_7. Given a state $s(F,T)$, the predicate $select(R,F)$, used in the bodies of clauses C_1–C_6, holds iff there exists at least

one rewriting from state $s(F,T)$ to a new state, generated by a rule in S_1–S_6, and $R \subseteq F$ is the removed set of that rewriting. This predicate encodes a selection function in the sense of Definition 3.4, that is, the set R is uniquely determined by F. In the body of clause C_7 we do not use the predicate $select(R,F)$, as the applicability of rule S_7 is determined by the predicate $no_other_premises(F)$, which holds iff there exists no R such that $select(R,F)$ holds. *EnactsEnbls* in clause C_7 is the removed set of the rewriting generated by rule S_7.

Note that, by Theorem 3.7, the reachability of a final state is independent of the specific selection function used by the operational semantics, and hence the actual implementation of the predicate $select(R,F)$ is immaterial.

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 (if X is uncontrollable) or the list C of durations of the controllable tasks (if X is controllable). The predicate $update(F,R,S,FU)$ holds iff FU is the set of fluents obtained from the set F by removing the fluents of the removed set R and adding the fluents of the added set S. The predicate $mintime(Enacts,M)$ holds iff *Enacts* is a set of fluents of the form $enacting(X,Res)$ and M is the minimum value of *Res* among the elements of *Enacts*. The predicate $decrease_residual_times(Enacts,M,EnactsU)$ holds iff *EnactsU* is the set of fluents obtained from the set *Enacts* by replacing every fluent of the form $enacting(X,Res)$ with the fluent $enacting(X,RU)$, where $RU = Res - M$. The predicates $member(El,Set)$ and $set_union(A,B,AB)$ are self-explanatory. The predicate $findall(X,G,L)$ 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.

Theorem 4.2 below shows the correctness of the encoding of both the process specifications and the operational semantics (that is, the interpreter) of business processes. First we need the following definition.

Definition 4.1. (Interpreter)
We denote by *Sem* the set $\{C_1,\ldots,C_7,R_1,R_2\}$ of clauses listed in Table 4.1, together with the clauses that define a business process specification (see, for instance, the clauses of Figure 2).

Theorem 4.2. (Correctness of encoding)
Let *init* be the term that encodes the initial state $\langle\{begins(start)\},0\rangle$ of the process, and let $fin(t)$, for any time instant t, be the term that encodes a final state $\langle\{completes(end)\},t\rangle$ of the process. Then, for every time instant t, $\langle\{begins(start)\},0\rangle \longrightarrow^* \langle\{completes(end)\},t\rangle$ iff there exist tuples u and c of integers such that $Sem \cup LIA \models reach(init, fin(t), u, c)$.

Proof:
Suppose that there exists a derivation $\langle\{begins(start)\},0\rangle \longrightarrow^* \langle\{completes(end)\},t\rangle$. Let u and c be the tuples of integers corresponding to the durations of the uncontrollable and controllable tasks, respectively, used in the derivation. As mentioned above, the predicate *tr* implements the relation '$\longrightarrow$' with a fixed selection function, say $\overline{\mathcal{R}}$. By Theorem 3.7 there exists a derivation via $\overline{\mathcal{R}}$ of the form $\langle F_0,t_0\rangle \longrightarrow \langle F_1,t_1\rangle \longrightarrow \ldots \longrightarrow \langle F_n,t_n\rangle$, where $\langle F_0,t_0\rangle = \langle\{begins(start)\},0\rangle$ and $\langle F_n,t_n\rangle = \langle\{completes(end)\},t\rangle$. Then, for $i=0,\ldots,n{-}1$, $Sem \cup LIA \models tr(s(F_i,t_i), s(F_{i+1},t_{i+1}), u, c)$, where $s(F_0,t_0) = init$ and $s(F_n,t_n) = fin(t)$, and hence, by the definition of the predicate *reach*, $Sem \cup LIA \models reach(init, fin(t), u, c)$.

Vice versa, if $Sem \cup LIA \models reach(init, fin(t), u, c)$, then, by the definition of the predicate *reach*, there exists a positive integer n such that, for $i = 0, \ldots, n-1$, $Sem \cup LIA \models tr(s(F_i, t_i), s(F_{i+1}, t_{i+1}), u, c)$, where $s(F_0, t_0) = init$ and $s(F_n, t_n) = fin(t)$. Thus, by the definition of tr, there exists a derivation $\langle F_0, t_0\rangle \longrightarrow \langle F_1, t_1\rangle \longrightarrow \ldots \longrightarrow \langle F_n, t_n\rangle$ via $\overline{\mathcal{R}}$, where $\langle F_0, t_0\rangle = \langle\{begins(start)\}, 0\rangle$ and $\langle F_n, t_n\rangle = \langle\{completes(end)\}, t\rangle$, and hence $\langle\{begins(start)\}, 0\rangle \longrightarrow^* \langle\{completes(end)\}, t\rangle$. □

4.2. Encoding Controllability Properties

A *reachability property* which specifies that a final state $\langle\{completes(end)\}, Tf\rangle$, also denoted $fin(Tf)$, can be reached from the initial state $\langle\{begins(start)\}, 0\rangle$, is introduced by a clause of the form:

$RP.\ reachProp(U,C) \leftarrow c(Tf,U,C),\ reach(init,\ fin(Tf),\ U,\ C)$

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

We say that the duration D of task X is *admissible* iff $duration(X,D)$ holds. The *weak controllability* problem for a business process specification consists in checking whether or not, for all admissible uncontrollable durations U, there exist controllable durations C such that $reachProp(U,C)$ holds. The *strong controllability* problem for a business process specification consists in checking whether or not there exist controllable durations C such that, for all admissible uncontrollable durations U, the property $reachProp(U,C)$ holds.

Now let us formally introduce the notions of weak controllability and strong controllability for a business process specification. First we need the following definition.

Definition 4.3. (Reachability property)
We denote by I the set $Sem \cup \{RP\}$ of clauses, and we say that I defines a reachability property.

Definition 4.4. (Weak and strong controllability)
Given a business process specification $\mathcal{B}$ and a reachability property defined by a set I of clauses, we say that:
(i) $\mathcal{B}$ is *weakly controllable* iff $I \cup LIA \models \forall U. adm(U) \rightarrow \exists C\ reachProp(U,C)$, and
(ii) $\mathcal{B}$ is *strongly controllable* iff $I \cup LIA \models \exists C\, \forall U.\ adm(U) \rightarrow reachProp(U,C)$,
where $adm(U)$ holds iff U is a tuple of admissible durations.

Note that, by definition, $adm(U)$ is satisfiable because we have assumed that no duration interval is empty. Moreover, if $reachProp(U,C)$ holds, then all durations that allow the process to reach a final state are admissible, and hence in the definition of the weak or strong controllability there is no need to require that the existentially quantified controllable durations C are admissible.

When a business process specification 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 be an unrealistic requirement in practice, as uncontrollable tasks may occur after controllable ones.

Note also that strong controllability implies weak controllability and guarantees that suitable durations of the controllable tasks can be computed in advance, before the enactment of the process, by using the constraints on the uncontrollable durations.

5. Specializing Reachability Properties

The clauses of the set I (see Definition 4.3) use complex terms (that is, terms with function symbols of positive arity), and in particular they represent a state by a pair of a set of fluents and a time instant (see clauses C_1–C_7). Now we present a transformation that *specializes* the set I to the particular business process specification under consideration, and derives an equivalent set I_{sp} of function-free CHCs, on which CHC solvers are very effective. The specialization algorithm is a variant of the one called *Removal of the Interpreter*, which has been proposed in the area of verification of imperative programs [15]. The specialization algorithm makes use of the following well-studied transformation rules: *unfolding*, *definition introduction*, and *folding* [17].

The specialization algorithm (see Figure 3) starts off by unfolding clause RP, and then it performs some more unfolding steps that realize a symbolic exploration of an initial portion 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.

Now, if we unfold every atom of every clause derived by unfolding, we may get into an infinite unfolding process. Thus, in order to ensure termination, in the Unfolding phase of the specialization algorithm we put a restriction on the unfolding rule. This restriction is based on the notion of an *unfoldable* atom. An atom is said to be *unfoldable* if: either (i) its predicate is different from *reach*, or (ii) it is of the form $reach(s(F1, T1), s(F2, T2), U, C)$ and $I \cup LIA \not\models \exists X.\, no_other_premises(F1)$, where X is the tuple of variables occurring in $F1$ (that is, rule S_7 is not applicable in state $s(F1, T1)$).

In the Unfolding phase, for every clause $C \in InCls$, we first unfold the only atom in the body of C, and then we unfold all *unfoldable* atoms in the body of the clauses obtained by previous applications of the unfolding rule. At the end of the Unfolding phase, we obtain a set of CHCs where each clause is either a constrained fact or a clause of the form:

$E.\;\; H \leftarrow e,\; reach(s(fl(Rs), T), fin(Tf), U, C)$

where e is a constraint, $fl(Rs)$ is a term representing a set of fluents, and Rs is the tuple of variables representing the residual times.

The goal of the Definition-Introduction & Folding phase is to derive a function-free clause from every clause E by applying the folding rule. If clause E can be folded by using a

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

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

Figure 3. The Specialization Algorithm.

clause already in *Defs*, then a function-free clause is obtained by folding (see the *then* branch of the *if-then-else* statement in the DEFINITION-INTRODUCTION & FOLDING phase). Otherwise, a new predicate definition is introduced by a clause of the form:

$$\mathit{newr}(Rs,T,\mathit{Tf},U,C) \leftarrow \widetilde{e}(Rs),\ \mathit{reach}(s(\mathit{fl}(Rs),T),\mathit{fin}(\mathit{Tf}),U,C)$$

where $\widetilde{e}(Rs)$ is obtained by projecting the constraint e onto the tuple Rs of residual time variables

(see the *else* branch of the *if-then-else* statement). We have that $e \sqsubseteq \widetilde{e}(Rs)$, and this guarantees the correctness of the subsequent folding step. At the end of the execution of either branch of the *if-then-else* statement, every *reach* atom with a complex argument representing a state, is replaced by a function-free call 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}. Moreover, since initially $I_{sp} = \emptyset$, we have that all clauses in I_{sp} are function-free. The specialization algorithm proceeds by adding the clause defining the new predicate *newr* to the set *InCls* of the clauses to be specialized and also to the set *Defs* of the clauses introduced by the definition introduction rule. The algorithm terminates when all clauses in *InCls* have been processed.

Note that, since the residual time variables range over finite integer intervals, there are finitely many non-equivalent constraints $\widetilde{e}(Rs)$. This property ensures that the set of clauses introduced in *Defs* is finite, and hence the specialization algorithm terminates for any set I of CHCs defining a reachability property (see Theorem B.5 in Appendix B for a detailed proof).

By known results, the unfolding, definition introduction, and folding rules preserve satisfiability [17], and hence for all tuples of integer values u and c, $reachProp(u, c)$ holds in $I \cup LIA$ if and only if it holds in $I_{sp} \cup LIA$.

Thus, we have the following result.

Theorem 5.1. (Total correctness of the specialization algorithm)
For any input set I of CHCs defining a reachability property, the specialization algorithm terminates. Suppose that the specialization algorithm returns the set I_{sp} of CHCs. Then we have that: (i) I_{sp} is a set of function-free CHCs, and (ii) for all tuples of integer values u and c,
$I \cup LIA \models reachProp(u, c)$ iff $I_{sp} \cup LIA \models reachProp(u, c)$.

Example 5.2. Let *Sem* be the set of clauses as in Definition 4.1, and let the reachability property for the process *CarDealer* of Figure 1 be denoted by the set $I = Sem \cup \{RP1\}$, where $RP1$ is the following clause:

$$reachProp(U0,U1,C0,C1,C2,C3) \leftarrow Tf \geq 15,\ Tf \leq 20,\ \ reach(init, fin(Tf), U0,U1,C0,C1,C2,C3)$$

In the above clause, the pair $(U0, U1)$ denotes the duration of the uncontrollable tasks $u0$ and $u1$, and the 4-tuple $(C0, C1, C2, C3)$ denotes the duration of the controllable tasks $c0$, $c1$, $c2$, and $c3$. Tf denotes a final time instant.

Now, let us run the specialization algorithm using the set I of clauses as input. Initially, $I_{sp} = \emptyset$, $InCls = \{RP1\}$, and $Defs = \emptyset$.

The UNFOLDING phase, starting from clause $RP1$, derives the following clause:

$$\begin{aligned} E.\ \ & reachProp(U0, U1, C0, C1, C2, C3) \leftarrow \\ & \quad Rc0 = C0,\ Ru1 = U1,\ Ti = 0,\ Tf \geq 15,\ Tf \leq 20,\ C0 \geq 1,\ C0 \leq 8,\ U1 \geq 4,\ U1 \leq 6, \\ & \quad reach(s(\{enacting(c0, Rc0), enacting(u1, Ru1)\}, Ti), s(\{completes(end)\}, Tf), \\ & \qquad U0, U1, C0, C1, C2, C3) \end{aligned}$$

Then, we perform the DEFINITION & FOLDING phase and, since *Defs* is the empty set of clauses, we introduce a new predicate $new1$ defined by the following clause:

$$\begin{aligned} F.\ \ & new1(Rc0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc0 \geq 1,\ Rc0 \leq 8,\ Ru1 \geq 4,\ Ru1 \leq 6, \\ & \quad reach(s(\{enacting(c0, Rc0), enacting(u1, Ru1)\}, Ti), s(\{completes(end)\}, Tf), \\ & \qquad U0, U1, C0, C1, C2, C3) \end{aligned}$$

whose constraint is the projection of the constraint in the body of clause E on the variables $Rc0$ and $Ru1$, denoting residual times, occurring in the set $\{enacting(c0, Rc0), enacting(u1, Ru1)\}$ of fluents. Then, by folding E using F, we get the following function-free clause:

$G.\ reachProp(U0, U1, C0, C1, C2, C3) \leftarrow Rc0 = C0,\ Ru1 = U1,\ Ti = 0,\ Tf \geq 15,\ Tf \leq 20,$
$\quad C0 \geq 1,\ C0 \leq 8,\ U1 \geq 4,\ U1 \leq 6,\ new1(Rc0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3)$

Now, the specialization algorithm proceeds by performing a new iteration of the body of the outer while-loop starting from $I_{sp} = \{G\}$, $InCls = \{F\}$, and $Defs = \{F\}$. Eventually, the algorithm terminates because the DEFINITION & FOLDING phase does not introduce any new predicate, and we derive the following function-free clauses as output:

$new1(Rc0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc0 \geq 1,\ Rc0 \leq Ru1,\ Ru1 \geq 4,\ Ru1 \leq 6,$ (‡)
$\quad Ru0 = U0,\ Ru1' = Ru1 - Rc0,\ Ti' = Ti + Rc0,\ Ti' \leq Tf,\ U0 \geq 2,\ U0 \leq 5,$
$\quad new2(Ru0, Ru1', Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new1(Rc0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc0 \geq Ru1,\ Rc0 \leq 8,\ Ru1 \geq 4,\ Ru1 \leq 6,$
$\quad Rc0' = Rc0 - Ru1,\ Rc1 = C1,\ Ti' = Ti + Ru1,\ Ti' \leq Tf,\ C1 \geq 2,\ C1 \leq 4,$
$\quad new3(Rc0', Rc1, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new2(Ru0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Ru0 \geq 2,\ Ru0 \leq 5,\ Ru1 = 0,\ Rc1 = C1,$
$\quad Ru0' = Ru0,\ C1 \geq 2,\ C1 \leq 4,\ new4(Rc1, Ru0', Ti, Tf, U0, U1, C0, C1, C2, C3)$

$new2(Ru0, Ru1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Ru0 \geq 2,\ Ru0 \leq 5,\ Ru1 \geq 1,\ Ru1 \leq Ru0,$
$\quad Rc1 = C1,\ Ru0' = Ru0 - Ru1,\ Ti' = Ti + Ru1,\ Ti' \leq Tf,\ C1 \geq 2,\ C1 \leq 4,$
$\quad new6(Rc1, Ru0', Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new3(Rc0, Rc1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc0 = 0,\ Rc1 \geq 2,\ Rc1 \leq 4,\ Rc1' = Rc1,$
$\quad Ru0 = U0,\ U0 \geq 2,\ U0 \leq 5,\ new4(Rc1', Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3)$

$new3(Rc0, Rc1, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc0 \geq 1,\ Rc0 \leq Rc1,\ Rc1 \geq 2,\ Rc1 \leq 4,$
$\quad Rc1' = Rc1 - Rc0,\ Ru0 = U0,\ U0 \geq 2,\ U0 \leq 5,\ Ti' = Ti + Rc0,\ Ti' \leq Tf,$
$\quad new5(Rc1', Ru0, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new4(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 4,\ Ru0 = Rc1,\ Rc2 = C2,$
$\quad Ti' = Ti + R1,\ Ti' \leq Tf,\ C2 \geq 1,\ C2 \leq 2,\ new7(Rc2, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new4(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 4,\ Ru0 = Rc1,\ Rc3 = C3,$
$\quad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ C3 \geq 4,\ C3 \leq 5,\ new8(Rc3, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new5(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 3,\ Ru0 = Rc1,\ Rc2 = C2,$
$\quad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ C2 \geq 1,\ C2 \leq 2,\ new7(Rc2, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new5(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 3,\ Ru0 = Rc1,\ Rc3 = C3,$
$\quad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ C3 \geq 4,\ C3 \leq 5,\ new8(Rc3, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new6(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 4,\ Ru0 = Rc1,\ Rc2 = C2,$
$\quad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ C2 \geq 1,\ C2 \leq 2,\ new7(Rc2, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new6(Rc1, Ru0, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc1 \geq 2,\ Rc1 \leq 4,\ Ru0 = Rc1,\ Rc3 = C3,$
$\quad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ C3 \geq 4,\ C3 \leq 5,\ new8(Rc3, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new7(Rc2, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc2 \geq 1,\ Rc2 \leq 2,\ Rc0 = C0,\ Ru1 = U1,$
$\quad Ti' = Ti + Rc2,\ Ti' \leq Tf,\ U1 \geq 4,\ U1 \leq 6,\ C0 \geq 1,\ C0 \leq 8,$
$\quad new1(Rc0, Ru1, Ti', Tf, U0, U1, C0, C1, C2, C3)$

$new8(Rc3, Ti, Tf, U0, U1, C0, C1, C2, C3) \leftarrow Rc3 \geq 4,\ Rc3 \leq 5,\ Tf = Ti + Rc3$

Clause G and the above clauses for $new1$–$new8$ form the set I_{sp} of clauses, which is the output of the specialization algorithm when given as input a set I of clauses defining a reachability property.

In the clauses for $new1$–$new8$ the variables RuN and RcN, possibly primed, denote the values of the residual durations of the uncontrollable and controllable tasks uN and cN, respectively. The variables Ti and Ti' denote time instants. The new predicates $newN$, which have been introduced by the specialization algorithm, represent *symbolic states*, denoting sets of states, by abstracting away the current time instants. For instance, predicate $new1$ corresponds to the following symbolic state (see clause F):

$s(\{enacting(c0, Rc0), enacting(u1, Ru1)\}, Ti)$

with $Rc0 \geq 1,\ Rc0 \leq 8,\ Ru1 \geq 4,\ Ru1 \leq 6$. Each clause describes a transition between symbolic states, where, by applying the unfolding rule, many transition steps may have been condensed into one only. For instance, the first clause for $new1$ (see clause (‡) above), when read 'from the head to the body', tells us that from the symbolic state represented by $new1$ the business process can go to the following symbolic state represented by $new2$:

$s(\{enacting(u0, Ru0), enacting(u1, Ru1')\}, Ti')$

with $Ru0 \geq 2,\ Ru0 \leq 5,\ Ru1' \geq 0,\ Ru1' \leq 5$. We leave to the reader to check that indeed these constraints on $Ru0$ and $Ru1'$ are the projections on the variables $Ru0$ and $Ru1'$ of the constraints in the body of clause (‡) .

6. Solving Controllability Problems

State-of-the-art CHC solvers are often not effective for solving controllability problems defined by a direct encoding of the formulas that occur in Definition 4.4, mainly because those formulas have nested existential and universal quantifiers and involve predicates that are defined by (possibly recursive) constrained Horn clauses. Thus, we propose an alternative method which essentially consists in solving a sequence of problems, each of which refers to simpler formulas and, in particular, to formulas whose quantifications are over *LIA* constraints only.

Our method is based on the fact that any reachability property defined by the set I of clauses (see Definition 4.3) is decidable. Indeed, for the set I_{sp} of CHCs obtained by the specialization algorithm starting from I, we have the following theorem (see Appendix C for a proof), where U and C are tuples of variables denoting the durations of the uncontrollable and controllable tasks, respectively.

Theorem 6.1. (Decidability of reachability properties)
There exists an algorithm, call it *solve*, such that, for any input query Q of the form $reachProp(U, C)$, (i) terminates, and (ii) returns an *answer constraint*, that is, a satisfiable *LIA* constraint $a(U, C)$ such that $I_{sp} \cup LIA \models \forall(a(U, C) \rightarrow Q)$, if such a constraint exists, and *false*, otherwise.

In what follows, for reasons of simplicity, an *answer constraint* will also be called an *answer*.

From Theorem 6.1 it follows that *solve* is a decision algorithm for the class of reachability properties considered in this paper. In the implementation of *solve*, unlike what we have done in the proof of Theorem 6.1, we do *not* perform an exhaustive instantiation of the variables occurring in U and C using the integer values of the finite domains over which those variables

range. Instead, we make use of the CLP(FD) library provided by SWI Prolog[1] which allows the manipulation of constraints over finite domains, and hence variables are instantiated on demand. Here we do not enter into the details of the implementation of *solve* via the CLP(FD) library.

The method we propose for solving the weak and strong controllability problems consists of two algorithms: (i) the Weak Controllability Algorithm (WCA, for short), and (ii) the Strong Controllability Algorithm (SCA, for short), respectively (see Figure 4). Each of these algorithms calls, once or more times, the algorithm *solve* and constructs a satisfiable constraint $\widetilde{a}(U,C)$, if any, where U and C are tuples of variables denoting the durations of the uncontrollable and controllable tasks, such that: (1) $I_{sp} \cup LIA \models \forall U\, \forall C.\, \widetilde{a}(U,C) \rightarrow reachProp(U,C)$ and either

(2.W) $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$ (for WCA) or

(2.S) $LIA \models \exists C\, \forall U.\, adm(U) \rightarrow \widetilde{a}(U,C)$ (for SCA).

Once WCA terminates, from (1) and (2.W) we get weak controllability (see Definition 4.4). Analogously, once SCA terminates, from (1) and (2.S) we get strong controllability.

Algorithm WCA constructs the constraint $\widetilde{a}(U,C)$ as a disjunction of answers computed by the algorithm *solve* until either Condition (2.W) holds or there are no more answers, in which case *solve* returns *false*. Algorithm SCA works similarly to algorithm WCA by considering Condition (2.S), instead of Condition (2.W).

In order to avoid redundant answers, at each iteration of the *do-while* body of the algorithms WCA and SCA, the algorithm $solve(I_{sp}, Q)$ is called with input queries Q of the form $reachProp(U,C) \wedge \varphi$, where φ is the (possibly quantified) negation of the disjunction of the answers computed so far.[2] Note that, since the *LIA* theory is decidable [5], Theorem 6.1 holds also for any query Q of this extended form (see Appendix C).

The following theorem, whose proof is in Appendix D, states that the algorithms WCA and SCA are totally correct.

Theorem 6.2. (Total correctness of the controllability algorithms WCA and SCA)
Let I_{sp} be the set of CHCs derived by the specialization algorithm of Section 5 starting from a set I of CHCs defining a reachability property for a business process specification $\mathcal{B}$.
(1) Given as input the set I_{sp}, the algorithm WCA: (i) terminates, and (ii) returns a satisfiable constraint $\widetilde{a}(U,C)$ such that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$, if $\mathcal{B}$ is weakly controllable, and returns *false*, otherwise.
(2) Given as input the set I_{sp}, the algorithm SCA: (i) terminates, and (ii) returns a satisfiable constraint $\widetilde{a}(U,C)$ such that $LIA \models \exists C.\, \forall U.\, adm(U) \rightarrow \widetilde{a}(U,C)$, if $\mathcal{B}$ is strongly controllable, and returns *false*, otherwise.

Now we illustrate how the algorithm WCA works by applying it to the clauses obtained by specialization at the end of the previous section (see Example 5.2).

During the first iteration of the *do-while* loop, the algorithm WCA evaluates the function call $solve(I_{sp}, reachProp(U0,U1,C0,C1,C2,C3) \wedge \neg\, \exists\, C0,C1,C2,C3.\ false)$.

We get the following answer constraint

$$a1(U0,U1,C0,C1,C2,C3):\ U0\!=\!4,\ U1\!=\!6,\ C0\!=\!U1,\ C1\!=\!U0,\ C3\!=\!5. \qquad (a1)$$

[1] http://www.pathwayslms.com/swipltuts/clpfd/clpfd.html

[2] In particular, in WCA the negated existential quantification of the controllable duration variables C avoids computing multiple answers with the same values of the durations U of the uncontrollable tasks.

Input: A set I_{sp} of CHCs defining a reachability property for a business process specification $\mathcal{B}$.
Output of WCA: A satisfiable constraint $\widetilde{a}(U,C)$ such that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$, if $\mathcal{B}$ is weakly controllable, and otherwise, *false*.
Output of SCA: A satisfiable constraint $\widetilde{a}(U,C)$ such that $LIA \models \exists C. \forall U.\, adm(U) \rightarrow \widetilde{a}(U,C)$, if $\mathcal{B}$ is strongly controllable, and otherwise, *false*.

WCA: *Weak Controllability Algorithm*	SCA: *Strong Controllability Algorithm*
$\widetilde{a}(U,C) := false$	$\widetilde{a}(U,C) := false$
do {	*do* {
$A := solve(I_{sp}, reachProp(U,C) \wedge \neg \exists C. \widetilde{a}(U,C))$;	$A := solve(I_{sp}, reachProp(U,C) \wedge \neg \widetilde{a}(U,C))$;
if $(A = false)$ *return false*	*if* $(A = false)$ *return false*
else $\widetilde{a}(U,C) := \widetilde{a}(U,C) \vee A$;	*else* $\widetilde{a}(U,C) := \widetilde{a}(U,C) \vee A$;
} *while* $(LIA \not\models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C))$;	} *while* $(LIA \not\models \exists C\, \forall U.\, adm(U) \rightarrow \widetilde{a}(U,C))$;
return $\widetilde{a}(U,C)$;	*return* $\widetilde{a}(U,C)$;

Figure 4. The algorithms WCA and SCA for verifying weak controllability and strong controllability, respectively.

In our example, the constraint $adm(U0,U1)$ is $U0 \geq 2,\ U0 \leq 5,\ U1 \geq 4,\ U1 \leq 6$.

Now we have that:

$LIA \not\models \forall U0, U1.\, adm(U0,U1) \rightarrow \exists\, C0, C1, C2, C3.\ a1(U0,U1,C0,C1,C2,C3)$

and hence the algorithm executes the second iteration of the *do-while* loop. Thus, the algorithm WCA evaluates the function call

$solve(I_{sp}, reachProp(U0,U1,C0,C1,C2,C3) \wedge \neg \exists\, C0, C1, C2, C3.\, a1(U0,U1,C0,C1,C2,C3))$.

We get the following answer constraint

$$a2(U0,U1,C0,C1,C2,C3)\!:\ U0 \geq 2,\ U0 \leq 3,\ U1 \geq 4,\ U1 \leq 6,\ C0 = U1,\ C1 = U0,\ C2 \geq 1,\ C2 \leq 2,\ C3 \geq 4,\ C3 \leq 5. \quad (a2)$$

Again, we have that:

$LIA \not\models \forall U0, U1.\, adm(U0,U1) \rightarrow \exists\, C0, C1, C2, C3.\ (a1(U0,U1,C0,C1,C2,C3) \vee a2(U0,U1,C0,C1,C2,C3))$.

Hence the algorithm performs further iterations of the *do-while* loop. At the first iteration the *solve* function returns the answer constraint

$$a3(U0,U1,C0,C1,C2,C3)\!:\ U0 = 5,\ U1 \geq 4,\ U1 \leq 6,\ U1 \geq C0,\ C0 \leq 3,\ C0 \geq 5,\ C1 \leq 2,\ C1 \geq 4,\ C3 \geq 4,\ C3 \leq 5, \quad (a3)$$

and, at the second iteration, *solve* returns the answer constraint

$$a4(U0,U1,C0,C1,C2,C3)\!:\ U0 = 4,\ U1 \geq 4,\ U1 \leq 5,\ C0 \leq 2,\ C0 \geq 4,\ C1 \leq 2,\ C1 \geq 3,\ C2 \leq 1,\ C2 \geq 2,\ C3 \geq 4,\ C3 \leq 5,\ U1 - C0 \geq 1,\ U1 - C0 \leq 2,\ U1 - C0 + C1 = 4. \quad (a4)$$

Now, the condition of the *do-while* loop is false, because $LIA \models \forall U0, U1.\, adm(U0,U1) \rightarrow \exists C0, C1, C2, C3.\ \bigvee_{i=1}^{4} a_i(U0,U1,C0,C1,C2,C3)$. Thus, the algorithm WCA terminates and we conclude that the process *CarDealer* is weakly controllable.

Now, as a further example of use of the controllability algorithms, we will see the strong controllability algorithm SCA in action for the business process, called *Proc*2, depicted in Figure 5.

The reachability property we now consider expresses the fact that the total duration time of *Proc*2, when expressed in time units, is in the interval $[6, 10]$. This property is defined by the following clause *RP*2, together with the set *Sem* of clauses listed in Table 4.1 defining the predicate *reach*:

$$reachProp(U0, U1, U2, C0, C1) \leftarrow Tf \geq 6,\ \ Tf \leq 10,\ \ reach(init, fin(Tf), U0, U1, U2, C0, C1)$$

where $U0, U1$, and $U2$ denotes the duration of the uncontrollable tasks $u0$, $u1$ and $u2$, respectively, and similarly, $C0$ and $C1$ denotes the durations of the controllable tasks $c0$ and $c1$.

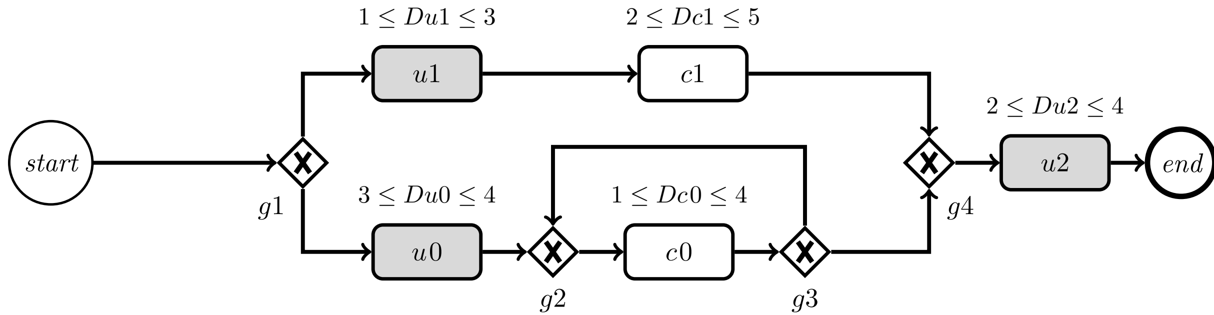

Figure 5. A business process *Proc*2. Tasks $u0$, $u1$, and $u2$ (with grey background) have uncontrollable durations. The other tasks $c0$ and $c1$ (with white background) have controllable durations.

By applying the specialization algorithm to the set I of input clauses $Sem \cup \{RP2\}$, we derive the following set I_{sp} of function-free constrained Horn clauses:

$$\begin{aligned}
&reachProp(U0, U1, U2, C0, C1) \leftarrow Ru1 \geq 1,\ Ru1 \leq 3,\ Ru1 = U1,\ Ti = 0,\ Tf \geq 6,\ Tf \leq 10,\\
&\qquad new1(Ru1, Ti, Tf, U0, U1, U2, C0, C1)\\
&reachProp(U0, U1, U2, C0, C1) \leftarrow Ru0 \geq 3,\ Ru0 \leq 4,\ Ru0 = U0,\ Ti = 0,\ Tf \geq 6,\ Tf \leq 10,\\
&\qquad new2(Ru0, Ti, Tf, U0, U1, U2, C0, C1)\\
&new1(Ru1, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Ru1 \geq 1,\ Ru1 \leq 3,\ Rc1 = C1,\ C1 \geq 2,\ C1 \leq 5,\\
&\qquad Ti' = Ti + Ru1,\ Ti' \leq Tf,\ new4(Rc1, Ti', Tf, U0, U1, U2, C0, C1)\\
&new2(Ru0, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Ru0 \geq 3,\ Ru0 \leq 4,\ Rc0 = C0,\ C0 \geq 1,\ C0 \leq 4,\\
&\qquad Ti' = Ti + Ru0,\ Ti' \leq Tf,\ new3(Rc0, Ti', Tf, U0, U1, U2, C0, C1)\\
&new3(Rc0, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Rc0 \geq 1,\ Rc0 \leq 4,\ Rc0' = C0,\ C0 \geq 1,\ C0 \leq 4,\\
&\qquad Ti' = Ti + Rc0,\ Ti' \leq Tf,\ new3(Rc0', Ti', Tf, U0, U1, U2, C0, C1)\\
&new3(Rc0, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Rc0 \geq 1,\ Rc0 \leq 4,\ Ru2 = U2,\ U2 \geq 2,\ U2 \leq 4,\\
&\qquad Ti' = Ti + Rc0,\ Ti' \leq Tf,\ new5(Ru2, Ti', Tf, U0, U1, U2, C0, C1)\\
&new4(Rc1, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Rc1 \geq 2,\ Rc1 \leq 5,\ Ru2 = U2,\ U2 \geq 2,\ U2 \leq 4,\\
&\qquad Ti' = Ti + Rc1,\ Ti' \leq Tf,\ new5(Ru2, Ti', Tf, U0, U1, U2, C0, C1)\\
&new5(Ru2, Ti, Tf, U0, U1, U2, C0, C1) \leftarrow Ru2 \geq 2,\ Ru2 \leq 4,\ Tf = Ti + Ru2
\end{aligned}$$

During the first iteration of the *do-while* loop of the algorithm SCA, the function call

$$\mathit{solve}(I_{sp},\ \mathit{reachProp}(U0,U1,U2,C0,C1) \wedge \neg \mathit{false}) \qquad (\dagger)$$

is evaluated and returns the following answer constraint:

$$a1(U0,U1,U2,C0,C1):\ U1\geq 1,\ U1\leq 3,\ U2\geq 2,\ U2\leq 4,\ C1\geq 2,\ C1\leq 5,\ U1+C1+U2\geq 6,\ U1+C1+U2\leq 10.$$

In our example we have that:

$\mathit{adm}(U0,U1,U2)$ is $U0\geq 3,\ U0\leq 4,\ U1\geq 1,\ U1\leq 3,\ U2\geq 2,\ U2\leq 4$.

Thus, we have that $\mathit{LIA} \models \exists\, C0,C1.\, \forall\, U0,U1,U2.\ \mathit{adm}(U0,U1,U2) \rightarrow a1(U0,U1,U2,C0,C1)$. As a consequence, the SCA algorithm terminates and we conclude that the strong controllability property holds for process *Proc*2, that is, there exist $C0$ and $C1$ such that for all admissible values of $U0$, $U1$, and $U2$, we have that the total duration Tf of *Proc*2 is in the interval $[6,10]$. Indeed, Tf is ensured to be in that interval if we take $C1=3$ and any $C0$ in the interval $[1,4]$.

A different answer constraint, besides $a1$, that can be obtained by evaluating the function call (†), is the following:

$$a2(U0,U1,U2,C0,C1):\ U0\geq 3,\ U0\leq 4,\ U2\geq 2,\ U2\leq 4,\ C0\geq 1,\ C0\leq 4,\ U0+C0+U2\geq 6,\ U0+C0+U2\leq 10.$$

In this case, if we take either $C0=1$ or $C0=2$, then Tf is ensured to be in the interval $[6,10]$ for all admissible values of $U0$, $U1$, and $U2$,

We have used the VeriMAP transformation and verification system for CHCs [14] to implement the specialization algorithm of Section 5, and SWI Prolog and the Z3 solver to implement the algorithms WCA and SCA. We have applied our method for the verification of the weak controllability of the process *CarDealer* and the strong controllability of the process *Proc*2.

The times taken in these verifications are as follows[3]. For the verification of the weak controllability of the process *CarDealer* the execution of the specialization algorithm requires 0.19 seconds and the execution of the algorithm WCA requires 0.67 seconds, whereas for the verification of the strong controllability of the process *Proc*2 the execution of the specialization algorithm requires 0.03 seconds and the execution of the algorithm SCA requires 0.38 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 *CarDealer*, contains cycles that may generate an unbounded proof search, and hence may cause non-termination if not handled in an appropriate way. In particular, in every example we have considered, the Z3 solver is not able to provide a proof of the desired controllability property within a time limit of one hour for a direct encoding of the controllability properties as they are formulated in Definition 4.4 above.

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 addressed this

[3]The experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under GNU/Linux OS, for the execution of the specialization algorithm, and on an Intel i5 2.3GHz with 8GB memory under macOS, for the execution of the algorithms WCA and SCA.

class of problems and we have presented a method for checking the so-called weak and strong 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 past. We refer to a paper by Cheikhrouhou et al. [6] for a survey which covers the recent years, while among the less recent techniques we limit ourselves to mention PERT [28] and GERT [32]. Although both techniques can be used for analyzing the time behaviour of business processes, and in particular for computing the minimal (or the expected) total duration of a process, neither of them addresses controllability problems as we do in this paper. PERT and GERT also have some other limitations: (i) PERT does not allow cycles in the activity diagrams and considers the duration of every activity to be specified by numbers, not by intervals, as done in our approach here, and (ii) GERT requires, in general, the Montecarlo simulation method to determine the time properties of the business processes, because the durations of the activities are specified in GERT via probabilistic distributions.

More recently, techniques for addressing violation of time constraints in business processes have been proposed in a paper by Combi et al. [10]. In that paper timer events, parallel gateways, and event-based gateways are used to represent and deal with time constraints. However, the issue of how to control the duration of the tasks so to avoid those violations has not been addressed.

The notion of controllability has been extensively studied in the context of scheduling and planning problems over temporal networks [7, 8, 9, 31, 36, 37], and it has been considered as a useful concept for supporting decisions in business process management and design [11, 24, 25].

Algorithms for checking weak and strong controllability properties were first introduced for Simple Temporal Networks with Uncertainty [37]. Later, sound and complete algorithms were developed for both weak [36] and strong [31] controllability of Disjunctive Temporal Problems with Uncertainty (DTPU). More recently, a general and effective method for checking weak [9] and strong [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 [11]. 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 [11] 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 [11]. 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 [11], 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 weak and strong 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] A. R. Bradley and Z. Manna. *The Calculus of Computation.* Springer, 2007.

[6] 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.

[7] A Cimatti, L. Hunsberger, A. Micheli, R. Posenato, and M. Roveri. Dynamic controllability via timed game automata. *Acta Informatica*, 53(6):681–722, 2016.

[8] A. Cimatti, A. Micheli, and M. Roveri. Solving strong controllability of temporal problems with uncertainty using SMT. *Constraints*, 20(1):1–29, 2015.

[9] 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.

[10] C. Combi, B. Oliboni, F. Zerbato. Modeling and handling duration constraints in BPMN 2.0. *Proc. SAC '17*, pp. 727–734, ACM, 2017.

[11] C. Combi and R. Posenato. Controllability in Temporal Conceptual Workflow Schemata. *Proc. BPM '09*, LNCS 5701, pp. 64–79. Springer, 2009.

[12] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Transactions on Database Systems*, 37(3):1–36, 2012.

[13] E. De Angelis, F. Fioravanti, M. C. Meo, A. Pettorossi, and M. Proietti. Verification of time-aware business processes using Constrained Horn Clauses. *Proc. LOPSTR '16*, LNCS 10184, pp. 38–55. Springer, 2017.

[14] 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.

[15] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming*, 147:78–108, 2017.

[16] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *Proc. TACAS '08*, LNCS 4963, pp. 337–340. Springer, 2008.

[17] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

[18] F. Fioravanti and A. Pettorossi and M. Proietti and V. Senni. Improving Reachability Analysis of Infinite State Systems by Specialization. *Fundamenta Informaticae*, 119(3–4): 281–300, 2012.

[19] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement*, FDR2 User Manual. `www.fsel.com`, 1998.

[20] 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.

[21] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[22] J. Jaffar and M. Maher and K. Marriott and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37:1–46, 1998.

[23] R.A. Kowalski and M.J. Sergot. A Logic-based Calculus of Events. *New Generation Comput.*, 4(1):67–95, 1986.

[24] 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.

[25] 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.

[26] 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, 1997.

[27] 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.

[28] D. G. Malcolm, J. H. Roseboom, C. E. Clark and W. Fazar. Application of a Technique for Research and Development Program Evaluation. *Operation Research*, 7(5), pp. 646–669, 1959.

[29] 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.

[30] OMG. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/`.

[31] 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.

[32] A. A. B. Pritsker. GERT: Graphical Evaluation and Review Technique. Memorandum RM-4973-NASA. National Aeronautics and Space Administration, 1966.

[33] M. Proietti and F. Smith. Reasoning on data-aware business processes with constraint logic. *Proc. SIMPDA '14*, *CEUR*, Vol. 1293, pp. 60–75, 2014.

[34] F. Smith and M. Proietti. Rule-based behavioral reasoning on semantic business processes. In: *Proc. ICAART '13, Vol. II*, pp. 130–143. SciTePress, 2013.

[35] 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.

[36] 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, Association for the Advancement of Artificial Intelligence, 2010.

[37] 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.

[38] 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.

[39] I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: On the verification of semantic business process models. *Distrib. Parallel Databases*, 27:271–343, 2010.

[40] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[41] P.Y.H. Wong and J. Gibbons. A relative timed semantics for BPMN. *Electronic Notes Theoretical Computer Science*, 229(2):59–75, 2009.

A. Proof of Theorem 3.7

The following fact holds for rules S_1–S_6.

Fact A.1. (i) Any two distinct rewritings, from the same state, generated by two instances of (not necessarily distinct) rules in S_1–S_6 have the removed sets which are either identical or disjoint. (ii) If an instance σ of a rule in S_1–S_6 is applied to a state $\langle F, t\rangle$, then for every fluent f of the removed set of the rewriting generated by σ, the premise of σ implies $f \in F$.

Proof:
Point (i) follows immediately from the definitions of the rules. Indeed, if we take two rewritings generated by two instances of rule S_3 with the same value of x and different values of s, then their removed sets will be identical. In all other cases the removed sets of two distinct rewritings are disjoint.

For Point (ii) only rule S_4 requires some explanation. By the premise of rule S_4, if we consider the instance σ of that rule which binds the variable x to the constant a, we have that: (1) $\forall p\ seq(p, a) \rightarrow (enables(p, a) \in F)$. Moreover, for any flow object p, when the element $enables(p, a)$, is added to a set of fluents (see rules S_2 and S_3), we have that: (2) $seq(p, a)$ holds. From (1) and (2), it follows that $enables(p, a) \in F$ is implied by the premise of the instance σ. □

The following lemma states that two rewritings using rules S_1–S_6 can be switched.

Lemma A.2. (Switching lemma for rules S_1–S_6)
Consider a state $s_0 = \langle F_0, t\rangle$ and two rewritings, say $s_0 \longrightarrow s_1$ and $s_0 \longrightarrow s'_1$, generated by two distinct instances σ and σ' of two (not necessarily distinct) rules in S_1–S_6. Let $R \subseteq F_0$ and $R' \subseteq F_0$, respectively, be the removed sets of those rewritings.

Suppose that $\delta = s_0 \longrightarrow s_1 \longrightarrow s_2$ is a derivation generated by applying the rule instance σ on the state s_0 and then the rule instance σ' on the state s_1. We have the following.
Point (i). There exists a derivation $\delta' = s_0 \longrightarrow s'_1 \longrightarrow s_2$, such that σ' whose removed set is R', generates $s_0 \longrightarrow s'_1$, and σ whose removed set is R, generates $s'_1 \longrightarrow s_2$.
Point (ii). A fluent f occurs in the set of fluents of a state in $\{s_0, s_1, s_2\}$ iff f occurs in the set of fluents of a state in $\{s_0, s'_1, s_2\}$.

Proof:
Proof of Point (i). First we observe that the time components of the states s_0, s_1, s'_1, and s_2 are all t, because when applying rules S_1–S_6, time does not pass.

Let F_1, F'_1, and F_2 be the sets of fluents of the states s_1, s'_1, and s_2, respectively. Let S and S' be the added set of the rewritings $s_0 \longrightarrow s_1$ and $s_0 \longrightarrow s'_1$, respectively. By construction of δ, we have that R' and S' are also the removed set and the added set, respectively, of the rewriting $s_1 \longrightarrow s_2$.

From $s_0 \longrightarrow s_1$, by safeness of the business process, we get $F_0 \cap S = \emptyset$, and thus $R \cap S = \emptyset$ and $R' \cap S = \emptyset$ (because R and R' are subsets of F_0). Symmetrically, from $s_0 \longrightarrow s'_1$, we get $F_0 \cap S' = \emptyset$, and thus $R' \cap S' = \emptyset$ and $R \cap S' = \emptyset$.

Since: (1) $F_1 = (F_0 \backslash R) \cup S$, (2) $F_0 \cap S = \emptyset$, and (3) R' is the removed set of the two rewritings $s_0 \longrightarrow s'_1$ and $s_1 \longrightarrow s_2$, by Fact A.1, we have that $R \cap R' = \emptyset$. From $S \subseteq F_1$ and $F_1 \cap S' = \emptyset$ (by safeness), we get $S \cap S' = \emptyset$.

Thus, we have that the sets R, R', S, and S' of fluents are pairwise disjoint. From the derivation $\delta = s_0 \longrightarrow s_1 \longrightarrow s_2$, we have that $F_2 = (((F_0 \backslash R) \cup S) \backslash R') \cup S'$.

Now the derivation δ' is obtained as follows. From state s_0 by applying the rule instance σ' we derive the state $s'_1 = \langle F'_1, t\rangle$ such that $F'_1 = (F_0 \backslash R') \cup S'$ and then, by applying the rule instance σ we derive a state, say $\langle F'_2, t\rangle$, with $F'_2 = (((F_0 \backslash R') \cup S') \backslash R) \cup S$. To complete the proof of Point (i) it remains to show that $F_2 = F'_2$. This is a consequence of the fact that the sets R, R', S, and S' are pairwise disjoint, and thus F_2 and F'_2 are both equal to $(F_0 \backslash (R \cup R')) \cup (S \cup S')$.
Proof of Point (ii). If the fluent f occurs in $F_0 \cup F_2$, then Point (ii) is immediate. Otherwise, if the fluent f occurs in $F_1 \cup F'_1$, Point (ii) follows from the fact that $F_1 \cup F'_1 = F_0 \cup F_2$. □

We also have the following result which is a consequence of Fact A.1.

Fact A.3. (Permanence for rules S_1–S_6)
If an instance σ of a rule in S_1–S_6 can be applied to a state s, then σ can also be applied to any state s' such that $s \longrightarrow^+ s'$, where $\longrightarrow^+$ denotes the rewriting relation generated by a non-empty sequence of rule instances in S_1–S_6, each of which has a removed set different from the one of σ.

Now we introduce a notion which we need in what follows.

Definition A.4. (Removed fluent and added fluent in a derivation)
A fluent is *removed* (or *added*) in a derivation δ if it belongs to the removed set (or the added set, respectively) of a rewriting $s \longrightarrow s'$ in δ.

Lemma A.5. Let δ be an infinite derivation using rules S_1–S_7 starting from the initial state. Then there exists in δ a flow object a such that *begins*(a) is removed infinitely many times.

Proof:
First, let us observe that: (i) δ is an infinite derivation, (ii) the set of flow objects and the set of sequence flows in a business process are finite, and (iii) the durations of the flow objects are non-negative integers. From (i), (ii), and (iii), it follows that there must be a flow object (not necessarily a task) b, whose duration we denote by d_b, such that a fluent of the form: either (i) *begins*(b), or (ii) *enacting*(b, d) for some duration d, with $0 \leq d \leq d_b$, or (iii) *completes*(b), or (iv) *enables*(b, c) for some flow object c successor of b, is removed infinitely many times in δ. Now, if *begins*(b) is removed infinitely many times, then we have the thesis by taking the flow object a to be b.

Otherwise, take any flow object x with duration d_x such that *begins*(x) in δ is removed finitely many times (possibly zero). In this case, we have that in the infinite derivation δ, both *enacting*(x, d), for some d such that $0 \leq d \leq d_x$, and *completes*(x) are removed finitely many times (possibly zero). This is a consequence of the following Points (α), (β), and (γ).
• *Point* (α). When *begins*(x) is removed by rule S_1, then *enacting*(x, d_x) is added to the set of fluents.
• *Point* (β). For any fluent *enacting*(x, d), with $0 \leq d \leq d_x$, which is removed by rule S_7, a fluent *enacting*(x, d'), with $0 \leq d' < d$ is added. Thus, the number of applications of rule S_7 to the flow object x during the whole derivation δ is finite.

• *Point* (γ). $completes(x)$ is added by rule S_6 at the expense of removing the fluent $enacting(x, 0)$, which in turn, by the previous Point (β), is added a finite number of times during the whole derivation δ.

It remains to show that during the infinite derivation δ it is not the case that $enables(x, c)$, for some flow object c, is removed infinitely many times. We will show this final point by contradiction. Thus, let us assume to the contrary that $enables(x, c)$, for some flow object c, is removed infinitely many times. Then, $enables(x, c)$ is added infinitely many times. Now, if x is a *par_branch*, then by rule S_2 we have that $completes(x)$ is added infinitely many times and by Points (γ) above, this is a contradiction. Similarly, we get a contradiction if x is not a *par_branch*, by considering rule S_3, instead of rule S_2. This concludes the proof of the lemma. □

Lemma A.6. (Finiteness lemma for rules S_1–S_6)
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 rules S_1–S_6 only. The following points hold.

• *Point* (i): *In δ every state refers to the same time instant.* The proof is obvious.

• *Point* (ii): *In δ there exists a flow object, call it a, such that $begins(a)$ is removed infinitely many times.* This point follows from Lemma A.5, because any derivation which uses rules S_1–S_6 only, is a particular derivation which uses rules S_1–S_7.

• *Point* (iii): *In δ there exists a gateway g such that $begins(g)$ is removed infinitely many times.* This point can be shown as follows. From Point (ii) we have that if the duration d_a of a is 0, then Point (iii) is shown by taking g to be a, because: (1) gateways have duration 0, (2) the event *start* has duration 0, but *begin*(*start*) is removed once only, because the event *start* has no predecessor, (3) the event *end* has duration 0, but *begin*(*end*) is never removed, because the event *end* has no successor, and (4) tasks have positive integer duration.

Otherwise, we have that $d_a > 0$, and thus a is a task. Let g be the predecessor of a (that is, $seq(g, a)$ holds). Since, by Point (ii), $begins(a)$ is removed infinitely many times in δ, it should also be added infinitely many times. Thus, by rules S_4 and S_5 we have that $enables(g, a)$ is removed (and added) infinitely many times in δ. Then, by rules S_2 and S_3, $completes(g)$ is removed (and added) infinitely many times in δ and, by Points (α), (β), and (γ) of Lemma A.5, we have that $begins(g)$ is removed (and added) infinitely many times in δ.

Moreover, since $completes(g)$ is added infinitely many times in δ, by rule S_6 we have that $enacting(g, 0)$ is removed infinitely many times in δ. Since δ uses rules S_1–S_6 only and these rules do not allow time to pass, as stated in Point (i), by rule S_1 we have that $duration(g, 0)$ holds. This implies that g is a gateway, because: (1) g cannot be the *start* event (indeed, $completes(g)$ is added infinitely many times), and (2) g cannot be the *end* event either (indeed, $completes(g)$ is removed infinitely many times and $completes(end)$ is never removed).

• *Point* (iv): *There exists a cycle made out of gateways only.* This point can be shown as follows. Since $begins(g)$ is removed infinitely many times in δ (see Point (iii)), it is also the case that $begins(g)$ is added infinitely many times in δ. Thus, by rules S_4 and S_5, there exists a flow object p predecessor of g such that: (1) $seq(p, g)$, and (2) $enables(p, g)$ is removed (and added) infinitely many times in δ.

Now by the same argument of Point (iii) we conclude that p is a gateway and *begins*(p) is removed (and added) infinitely many times in δ. By an inductive argument, we get that for all $n \geq 0$, there exists a sequence $p_0, \ldots, p_n$ of flow objects with duration 0, such that: (1) for all i, with $0 \leq i < n$, $seq(p_i, p_{i+1})$, (2) $seq(p_n, g)$, and (3) $enables(p_n, g)$. Since neither the *start* nor the *end* event have a predecessor and a successor flow object, and the number of gateways in every business process we consider is finite, we get Point (iv).

Now, from Point (iv), we derive a contradiction (recall that we have assumed that in every business process no cycle made out of gateways only exists), and the proof of the lemma is completed. □

In the following, given the two derivations $\delta_1 = s_0 \longrightarrow^* s_n$ and $\delta_2 = s_n \longrightarrow^* s_m$, the concatenated derivation $s_0 \longrightarrow^* s_n \longrightarrow^* s_m$ will be denoted by $\delta_1 \cdot \delta_2$.

We also denote by $States_{\#}$ the set of states such that none of the rules in $\{S_1, \ldots, S_6\}$ can be applied.

Theorem A.7. Let $\overline{\mathcal{R}}$ be a fixed selection function. For every derivation δ from a state s_0 ending in a state in $States_{\#}$ via a selection rule $\mathcal{R}$, there exists a derivation $\overline{\delta}$ from s_0 via $\overline{\mathcal{R}}$ such that: (i) δ and $\overline{\delta}$ end in the same state, and (ii) for each state $\langle F, t\rangle$ in δ and $f \in F$, there exists a state $\langle \overline{F}, t\rangle$ in $\overline{\delta}$ and $f \in \overline{F}$.

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

Basis. $l = 0$. In this case δ uses the rules S_1–S_6 only. Let us assume that $s_0 = \langle F_0, t\rangle$, for some set F_0 of fluents and time instant t. Thus, δ is of the form: $\langle F_0, t\rangle \longrightarrow \langle F_1, t\rangle \longrightarrow \ldots \longrightarrow \langle F_n, t\rangle$.

Consider the smallest i (≥ 0) such that in state s_i the set R of fluents returned by $\mathcal{R}$ in δ differs from the set $\overline{R}$ of fluents returned by $\overline{\mathcal{R}}$ in $\overline{\delta}$. Since $R \neq \overline{R}$, by Fact A.1 we also have that $R \cap \overline{R} = \emptyset$, and suppose that $\overline{\sigma}$ is the instance of a rule in S_1–S_6 with removed set $\overline{R}$. Then, by Fact A.3, $\overline{\sigma}$ can also be applied in any state which is derived from s_i by applying a sequence of rules in S_1–S_6, whose removed sets, selected by $\mathcal{R}$, are different from $\overline{R}$. Moreover, δ ends with a state in $States_{\#}$, where no rule in S_1–S_6 can be applied. Thus, we have that, for some $j > 0$, $\overline{R}$ is the removed set computed by $\mathcal{R}$ in the state s_{i+j} of δ (since we assume that there are no loops through gateways only, this integer j is unique).

If such an i does not exist, namely δ is via $\overline{\mathcal{R}}$, then we take $i = n$ and $j = 0$.

Informally, i is the first position where the selection function $\mathcal{R}$ of δ deviates from the selection function $\overline{\mathcal{R}}$, and j is the delay of the selection function $\mathcal{R}$ of δ with respect to $\overline{\mathcal{R}}$. We call $(n - i, j)$ the (deviate, delay) pair of δ with respect to $\overline{\mathcal{R}}$.

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, for all non-negative integers h, k, h', k', we have that $(h, k) \prec (h', k')$ iff ($h < h'$ or ($h = h'$ and $k < k'$)).

If the (deviate, delay) pair is $(0, 0)$, then δ is via $\overline{\mathcal{R}}$ and we get the thesis.

Otherwise, suppose that the (deviate, delay) pair is $(n - i, j)$, with $(0, 0) \prec (n - i, j)$. Thus, the derivation δ is of form:

$s_0 \longrightarrow^* s_i \longrightarrow^* s_{i+j-1} \longrightarrow s_{i+j} \longrightarrow s_{i+j+1} \longrightarrow^* s_n$

where the removed set of the rewriting $s_{i+j} \longrightarrow s_{i+j+1}$ is $\overline{R}$ (that is, the removed set computed by $\overline{\mathcal{R}}$ at state s_i) and, by construction, the removed sets computed by $\mathcal{R}$ at states $s_i, \ldots, s_{i+j-1}$ is different from $\overline{R}$. Then, the rule instance applied at state s_{i+j} is also applicable at state s_{i+j-1}, and hence by Lemma A.2, the rule instances applied at states s_{i+j-1} and s_{i+j} can be switched yielding a derivation δ' of the form:

$s_0 \longrightarrow^* s_i \longrightarrow^* s_{i+j-1} \longrightarrow s' \longrightarrow s_{i+j+1} \longrightarrow^* s_n$

such that a fluent f occurs in s' iff f occurs either in s_{i+j-1} or in s_{i+j+1}.

The (deviate, delay) pair of δ' with respect to $\overline{\mathcal{R}}$ is $(n-i, j-1)$ if $j>1$, and it is $(n-i-1, \ell)$, for some $\ell \geq 0$, if $j=1$. Thus, in both cases the (deviate, delay) pair of δ' is below $(n-i, j)$ and, by well-founded induction with respect to $\prec$, we get the thesis.

Step. Assume for l, show for $l+1$. Let δ be of the form $s_0 \longrightarrow^* s_n \longrightarrow s_{n+1} \longrightarrow^* s_m$, where the number of applications of S_7 in $\delta_1 = s_0 \longrightarrow^* s_n$ and in $\delta_2 = s_{n+1} \longrightarrow^* s_m$ is l and 0, respectively, and $s_n \longrightarrow s_{n+1}$ is obtained by applying S_7. By definition of rule S_7, it is the case that $s_n \in \mathit{States}_{\#}$.

By inductive hypothesis we have that there exists a derivation $\overline{\delta_1} = s_0 \longrightarrow^* s_n$ from s_0 via $\overline{\mathcal{R}}$ such that: (i) δ_1 and $\overline{\delta_1}$ end in the same state, and (ii) for each state $\langle F, t\rangle$ in δ_1 and $f \in F$, there exists a state $\langle \overline{F}, t\rangle$ in $\overline{\delta}_1$ and $f \in \overline{F}$. Moreover, since $s_n \in \mathit{States}_{\#}$, only rule S_7 can be applied thereby obtaining the state s_{n+1}. Thus, $\overline{\delta_1'} = \overline{\delta_1} \cdot (s_n \longrightarrow s_{n+1})$ is a derivation via $\overline{\mathcal{R}}$.

Now, let $\overline{\mathcal{R}'}$ be the selection rule such that, for any derivation γ, $\overline{\mathcal{R}'}$ returns the same set of fluents of $\overline{\mathcal{R}}$ applied to $\overline{\delta_1'} \cdot \gamma$. Since the number of application of rule S_7 in δ_2 is equal to 0, there exists a derivation $\overline{\delta_2} = s_{n+1} \longrightarrow^* s_m$ via $\overline{\mathcal{R}'}$, such that: (i) δ_2 and $\overline{\delta_2}$ end in the same state, and (ii) for each state $\langle F, t\rangle$ in δ_2 and $f \in F$, there exists a state $\langle \overline{F}, t\rangle$ in $\overline{\delta_2}$ and $f \in \overline{F}$.

Now we get the thesis, because by construction $\overline{\delta_1'} \cdot \overline{\delta_2}$ is a derivation via $\overline{\mathcal{R}}$. □

Proof of Theorem 3.7

Proof of Point (i). If δ ends in a state $\mathit{States}_{\#}$, then Point (i) follows from Theorem A.7. Otherwise, by Lemma A.6, derivation δ can be extended to a derivation δ^{ext} ending in a state in $\mathit{States}_{\#}$. Then by Theorem A.7, there exists a derivation $\overline{\delta^{ext}}$ from s via $\overline{\mathcal{R}}$ such that for each state $\langle F, t\rangle$ in δ^{ext} and $f \in F$, there exists a state $\langle \overline{F}, t\rangle$ in $\overline{\delta^{ext}}$, and $f \in \overline{F}$. Since δ is a proper prefix of δ^{ext}, the proof of Point (i) is completed.

Proof of Point (ii). If the rewriting $\langle F, t\rangle \longrightarrow \langle F', t'\rangle$, with $t < t'$, occurs in δ, it is generated by rule S_7, and thus the state $\langle F, t\rangle$ belongs to $\mathit{States}_{\#}$. Let us consider the prefix γ of δ that ends in the state $\langle F, t\rangle$. By Theorem A.7 there exists a derivation $\overline{\gamma}$ from s via $\overline{\mathcal{R}}$ such that γ and $\overline{\gamma}$ end in the same state $\langle F, t\rangle$. Now, derivation $\overline{\gamma}$ can be extended by the selection rule $\overline{\mathcal{R}}$ and $\mathcal{R}$ in the same way, because they both apply rule S_7. This completes the proof of Point (ii). □

B. Proofs of Termination of the Specialization Algorithm

In order to show the termination of the specialization algorithm we prove some preliminary lemmas.

Lemma B.1. The UNFOLDING phase of the specialization algorithm terminates.

Proof:
Let C be a clause in *InCls*. Let $C_0, \ldots, C_n, \ldots$, where $C_0 = C$, be any sequence of clauses generated by the UNFOLDING phase. We have that, for $i = 0, \ldots, n, \ldots$, $C_{i+1} \in \mathit{Unf}(C_i, A, I)$, where A is either the single *reach* atom in the body of C_0 or the leftmost unfoldable atom in the body of C_i, if $i \geq 1$. We need to show that this sequence is finite. Assume, by contradiction, that $C_0, \ldots, C_n, \ldots$ is an infinite sequence. Since every atom with predicate different from *reach* is unfoldable, and the unfolding of such atoms generates finite sequences of clauses, from $C_0, \ldots, C_n, \ldots$ we can extract an infinite subsequence of clauses of the form:

E_0: $H \leftarrow e_0,\ \mathit{reach}(s(\mathit{fl}_0(\mathit{Rs}_0), T_0), \mathit{fin}(\mathit{Tf}), U, C)$
$\ldots$
E_k: $H \leftarrow e_k,\ \mathit{reach}(s(\mathit{fl}_k(\mathit{Rs}_k), T_k), \mathit{fin}(\mathit{Tf}), U, C)$
$\ldots$

where, by the correctness of the encoding (see Theorem 4.2), for $i = 0, \ldots, k, \ldots$, there exists a state $\langle F_i, t_i \rangle$ which is represented by a ground instance $s(\mathit{fl}_i(rs_i), t_i)$ of the term $s(\mathit{fl}_i(\mathit{Rs}_i), T_i)$, such that the residual times Rs_i and the time instant T_i satisfy the constraint e_i. Moreover, $\langle F_{i+1}, t_{i+1} \rangle$ is obtained from $\langle F_i, t_i \rangle$ by applying a rule in S_1–S_7. By Lemma A.6, there exists $m > 0$ such that $\langle F_{m+1}, t_{m+1} \rangle$ is obtained from $\langle F_m, t_m \rangle$ by applying rule S_7. Thus, $\mathit{no_other_premises}(F_m)$ holds, and hence

$I \cup \mathit{LIA} \models \exists \mathit{Rs}_m.\ \mathit{no_other_premises}(\mathit{fl}_m(\mathit{Rs}_m))$

which contradicts the fact that the atom $\mathit{reach}(s(\mathit{fl}_m(\mathit{Rs}_m), T_m), \mathit{fin}(\mathit{Tf}), U, C)$ is unfoldable. □

Let BPS be an abbreviation for Business Process Specification.

Lemma B.2. For every BPS the set $\mathcal{F} = \{F \mid \text{there exists } t \geq 0 \text{ such that } \mathit{init} \longrightarrow^* \langle F, t \rangle\}$ is finite.

Proof:
Since F is a set, for every flow object x, for every residual time r, for every successor y of x, there exists in F at most one fluent $\mathit{begins}(x)$, at most one fluent $\mathit{completes}(x)$, at most one fluent $\mathit{enacting}(x, r)$, and at most one fluent $\mathit{enables}(x, y)$.

The thesis follows from the facts that: (i) in any BPS there are finitely many flow objects, and (ii) for every flow object x, the set of fluents $\mathit{enacting}(x, r)$ that occur in $\mathcal{F}$ is finite, because the residual time r is a non-negative integer initialized to a duration belonging to a finite interval, and it can only decrease. □

Definition B.3. 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 B.4. All new definitions introduced by the specialization algorithm are clauses of the form :

$\mathit{newr}(\mathit{Rs}, T, \mathit{Tf}, U, C) \leftarrow \widetilde{e}(\mathit{Rs}),\ \mathit{reach}(s(\mathit{fl}(\mathit{Rs}), T), \mathit{fin}(\mathit{Tf}), U, C)$

where $\widetilde{e}(\mathit{Rs})$ is bounded by the largest value among the maximal task durations specified by the given BPS.

Proof:
Let m be the largest value among the maximal task durations specified by the BPS. Recall that the variables in the tuple Rs are 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 C_1, C_6, and C_7, and then unfolding the atoms occurring in the bodies of these clauses. If the *tr* atom is unfolded using C_1, then the subsequent unfolding of the atom $task_duration(X, D, U, C)$ 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 because, by definition of m, we have that $d_{max} \leq m$.

If the *tr* atom is unfolded using C_6, then the constraint $R = 0$ is added to a residual time variable R. Clearly, this constraint is bounded by m.

The case where the *tr* atom is unfolded using C_7 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 C_7, and subsequently with respect to the atoms in the body of C_7, we derive (zero or more) clauses whose body has a constraint of the following form, for some $j \in \{1, \ldots, n\}$:

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

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 $j = 1, \ldots, n$, 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 fact that the constraint $\widetilde{e}(Rs)$ is derived by projection onto the residual time variables Rs and this projection operation preserves boundedness. □

Now we are ready to show the termination of the specialization algorithm.

Theorem B.5. (Termination of the specialization algorithm)
For any input set of CHCs, the specialization algorithm terminates.

Proof:
By Lemma B.1, every execution of the UNFOLDING phase terminates. Also, every execution of the DEFINITION-INTRODUCTION & FOLDING phase terminates, because for each clause in *SpC*, at most one definition introduction and folding step is performed.

Thus, it remains to show that the outer loop of the specialization algorithm terminates, that is, the new predicate definitions that are introduced in *Defs* by the specialization algorithm constitute a finite set. Every new definition is a clause of the form

$newr(Rs, T, Tf, U, C) \leftarrow \widetilde{e}(Rs),\ reach(s(fl(Rs), T), fin(Tf), U, C)$

where: (i) Rs, U, and C are tuples of variables, (ii) T and Tf are variables, and (iii) $\widetilde{e}(Rs)$ is a constraint.

Now, by Lemma B.4, $\widetilde{e}(Rs)$ is bounded by the largest integer m which is an upper bound of a task duration interval specified by I, and the set of pairwise non-equivalent constraints bounded by m is finite. Moreover, consider the set $\mathcal{A}$ of all atoms $reach(s(fl(Rs), T), fin(Tf), U, C)$,

where $s(\mathit{fl}(Rs), T)$ is a term that can be obtained from a state $\langle F, t\rangle$ by replacing the residual time values in F by the variables Rs, and the time value t by the variable T. By Lemma B.2, the set $\mathcal{A}$ is finite, modulo variable names.

Since the specialization algorithm does not introduce two new definitions with equivalent constraints and equal *reach* atoms, modulo variable names, the set of new predicates introduced in *Defs* is finite. □

C. Proof of decidability of reachability properties

Proof:
Any set I_{sp} which is the output of the specialization algorithm is of the form:

$$\begin{array}{l} \mathit{reachProp}(U,C) \leftarrow c(\mathit{Tf},U,C),\ d_0(Rs,U,C),\ \mathit{new}1(Rs,0,\mathit{Tf},U,C) \\ \quad \ldots \\ \mathit{newK}(Rs1,T1,\mathit{Tf},U,C) \leftarrow d_K(Rs1,Rs2,U,C,M),\ T2{=}T1{+}M,\ T2{\leq}\mathit{Tf}, \\ \qquad \mathit{newN}(Rs2,T2,\mathit{Tf},U,C) \\ \quad \ldots \\ \mathit{newZ}(Rs1,T1,\mathit{Tf},U,C) \leftarrow d_Z(Rs1,U,C,M),\ T2{=}T1{+}M,\ T2{=}\mathit{Tf} \end{array}$$

where: (i) U and C denote tuples of uncontrollable and controllable durations, respectively, (ii) Tf is the time instant when the final state is reached, (iii) $T1$ and $T2$ are time instants at non-final states, (iv) $Rs, Rs1$, and $Rs2$ are tuples of residual times, (v) $M \geq 0$ is a time elapse, (vi) $c(\mathit{Tf},U,C)$, $d_0(Rs,U,C)$, ..., $d_K(Rs1,Rs2,U,C,M)$, ..., $d_Z(Rs1,U,C,M)$ are constraints.

The variables $U, C, Rs, Rs1, Rs2$, and M range over finite intervals of non-negative integers (they cannot take values higher than the upper bounds of the durations of the uncontrollable or controllable tasks). Thus, we can instantiate the above clauses using values in these finite intervals, and then replace instantiated atoms by atoms with new predicate names. Hence we get the following set I_{inst} of clauses:

$$\begin{array}{l} \mathit{reachProp}_1 \leftarrow c_1(\mathit{Tf}),\ \mathit{new}1_1(0,\mathit{Tf}) \\ \mathit{reachProp}_2 \leftarrow c_2(\mathit{Tf}),\ \mathit{new}1_2(0,\mathit{Tf}) \\ \qquad \ldots \\ \mathit{newK}_1(T1,\mathit{Tf}) \leftarrow T2{=}T1{+}m_1,\ T2{\leq}\mathit{Tf},\ \mathit{newN}_1(T2,\mathit{Tf}) \\ \mathit{newK}_2(T1,\mathit{Tf}) \leftarrow T2{=}T1{+}m_2,\ T2{\leq}\mathit{Tf},\ \mathit{newN}_2(T2,\mathit{Tf}) \\ \qquad \ldots \\ \mathit{newZ}_1(T1,\mathit{Tf}) \leftarrow T2{=}T1{+}r_1,\ T2{=}\mathit{Tf} \\ \mathit{newZ}_2(T1,\mathit{Tf}) \leftarrow T2{=}T1{+}r_2,\ T2{=}\mathit{Tf} \\ \qquad \ldots \end{array}$$

where, for $i = 1, \ldots$, we have the following: (i) $\mathit{reachProp}_i$ is an instance of $\mathit{reachProp}(U,C)$, (ii) $c_i(\mathit{Tf})$ is an instance of $c(\mathit{Tf},U,C)$, (iii) $\mathit{newX}_i(T1,\mathit{Tf})$ and $\mathit{newX}_i(T2,\mathit{Tf})$ are instances of $\mathit{newX}(Rs1,T1,\mathit{Tf},U,C)$ and $\mathit{newX}(Rs2,T2,\mathit{Tf},U,C)$, respectively, (iv) $m_1, m_2, \ldots, r_1, r_2, \ldots$ are non-negative integer values. Without loss of generality, we assume that $c_i(\mathit{Tf})$ is of the form either $\tau_1 \leq \mathit{Tf} \leq \tau_2$ or $\mathit{Tf} \geq \tau_2$, for some non-negative integers τ_1, τ_2 (indeed any constraint $c_i(\mathit{Tf})$ can be rewritten as a disjunction of constraints of the form $\tau_1 \leq \mathit{Tf} \leq \tau_2$ or $\mathit{Tf} \geq \tau_2$, and we can split the clause for $\mathit{reachProp}_i$ into a new clause for each disjunct).

A *constrained goal* is a conjunction (d, G), where d is a constraint and G is a (possibly empty) conjunction of atoms. (Recall that the empty conjunction is *true*.) Now let us recall the definition of a *derivation* (called *c-derivation* here, to avoid confusion with Definition 3.5), which is the basis of the operational semantics of CLP [22]. In this definition, for reasons of simplicity, we assume that all clauses have distinct variables in their head.

Definition C.1. (c-Derivation)
A *c-derivation* from a constrained goal (c_0, G_0) in a set P of clauses is a (finite or infinite) sequence of constrained goals $(c_0, G_0) \Rightarrow (c_1, G_1) \Rightarrow \ldots \Rightarrow (c_n, G_n) \Rightarrow \ldots$ where, for $i \geq 1$, (c_i, G_i) is obtained from (c_{i-1}, G_{i-1}) as follows:
if G_{i-1} is of the form (A, G) and in P there exists a clause $H \leftarrow c, B$ such that A and H are unifiable via an mgu ϑ and $(c_{i-1}, c)\vartheta$ is satisfiable in LIA,
then $c_i = (c_{i-1}, c)\vartheta$ and $G_i = (B, G)\vartheta$.
A finite c-derivation of the form $(c_0, G_0) \Rightarrow (c_1, G_1) \Rightarrow \ldots \Rightarrow (c_n, \mathit{true})$, where c_n is a satisfiable constraint, is said to be *successful*.

Suppose that a c-derivation is of the form $(c_0, G_0) \Rightarrow (c_1, G_1) \Rightarrow \ldots \Rightarrow (c_{i-1}, G_{i-1})$, where, for all clauses $H \leftarrow c, B$ in P, *either* A and H are not unifiable *or* they are unifiable via an mgu ϑ and $(c_{i-1}, c)\vartheta$ is unsatisfiable in LIA. Then the c-derivation cannot be extended, and it is said to be a *failed* c-derivation.

From the soundness and completeness of the operational semantics of CLP [22], we have that, for any tuples u, c of non-negative integers, $I_{sp} \cup LIA \models \forall(U = u \wedge C = c \rightarrow \mathit{reachProp}(U, C))$ iff there exists a successful c-derivation from $(\mathit{true}, \mathit{reachProp}(u, c))$ in I_{inst}. Thus, there exists an answer constraint for the query $\mathit{reachProp}(U, C)$ iff there exists a successful c-derivation from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst}, where $\mathit{reachProp}_i$ is one of the atoms introduced as explained above.

A c-derivation from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst} is said to be *up-cycling* if it is of the form:

$$\begin{aligned}(\mathit{true}, \mathit{reachProp}_i) &\Rightarrow (c_i(\mathit{Tf}),\ \mathit{new}1_i(0, \mathit{Tf})) \Rightarrow \ldots\\ &\Rightarrow (c_i(\mathit{Tf}),\ t1 \leq \mathit{Tf},\ \mathit{newX}_j(t1, \mathit{Tf})) \Rightarrow \ldots\\ &\Rightarrow (c_i(\mathit{Tf}),\ t2 \leq \mathit{Tf},\ \mathit{newX}_j(t2, \mathit{Tf})) \Rightarrow \ldots\end{aligned}$$

where $t1$ and $t2$ are non-negative integers such that $\tau_2 < t1 < t2$ (see above for the definition of τ_2). Since for every clause in I_{inst} of the form $\mathit{newX}_i(T1, \mathit{Tf}) \leftarrow T2 = T1 + x_i,\ T2 \leq \mathit{Tf},\ \mathit{newY}_i(T2, \mathit{Tf})$, the integer x_i is non-negative, we have $t1 \leq t2$. Moreover, by the assumption that there are no cycles whose flow objects are gateways only (see Section 2), and time elapses by at least one unit, when a task is enacted (that is, rule S_7 is applied), we easily get that $t1 < t2$. Thus, there is a threshold length l such that every c-derivation from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst} which is longer than l is up-cycling, and hence the set of c-derivations from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst} which are not up-cycling is finite.

Now, we show that if there exists an up-cycling successful c-derivation from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst}, then there exists also a successful c-derivation from $(\mathit{true}, \mathit{reachProp}_i)$ in I_{inst} which is *not up-cycling*.

Consider an up-cycling successful c-derivation of the form:

$$\begin{aligned}(\mathit{true}, \mathit{reachProp}_i) &\Rightarrow (c_i(\mathit{Tf}),\ \mathit{new}1_i(0, \mathit{Tf})) \Rightarrow \ldots\\ &\Rightarrow (c_i(\mathit{Tf}),\ t1 \leq \mathit{Tf},\ \mathit{newX}_j(t1, \mathit{Tf}))\ (\dagger) \Rightarrow \ldots\\ &\Rightarrow (c_i(\mathit{Tf}),\ t2 \leq \mathit{Tf},\ \mathit{newX}_j(t2, \mathit{Tf})) \overset{C_1}{\Rightarrow} \ldots \overset{C_n}{\Rightarrow} (c_i(\mathit{tf}), \mathit{true}) \qquad (\delta)\end{aligned}$$

where $C_1, \ldots, C_n$, for $n \geq 1$, are the clauses used in the construction of the final part of the c-derivation, and tf is the final time instant. Then, we can construct a shorter successful c-derivation of the form:

$$(true, reachProp_i) \Rightarrow (c_i(Tf),\ new1_i(0, Tf)) \Rightarrow \ldots \Rightarrow (c_i(Tf),\ t1 \leq Tf,\ newX_j(t1, Tf)) \stackrel{C_1}{\Rightarrow} \ldots \stackrel{C_2}{\Rightarrow} \ldots \stackrel{C_n}{\Rightarrow} (c_i(tf'), true) \qquad (\widetilde{\delta})$$

To see this, let us consider the following two cases.

Case (1). Suppose that $c_i(Tf)$ is of the form $\tau_1 \leq Tf \leq \tau_2$. Then, the constraint in the constrained goal $(c_i(Tf),\ t1 \leq Tf,\ newX_j(t1, Tf))$ is $(\tau_1 \leq Tf \leq \tau_2,\ t1 \leq Tf)$, which is unsatisfiable because $\tau_2 < t1$. Thus, this case is impossible.

Case (2). Suppose that $c_i(Tf)$ is of the form $Tf \geq \tau_2$. Then, the constraint in the constrained goal $(c_i(Tf),\ t1 \leq Tf,\ newX_j(t1, Tf))$ (†) is $(Tf \geq \tau_2,\ t1 \leq Tf)$, which is satisfiable. Clause C_1 can be used to derive a new constrained goal from the goal (†). Indeed, clause C_1 is of the form either (if $n=1$)

Case (2A). $newX_j(T1, Tf) \leftarrow T2 = T1 + r_j,\ T2 = Tf$

or (if $n > 1$)

Case (2B). $newX_j(T1, Tf) \leftarrow T2 = T1 + m_j,\ T2 \leq Tf,\ newY_j(T2, Tf)$

In Case (2A) from the constrained goal (†), that is, $(Tf \geq \tau_2,\ t1 \leq Tf,\ newX_j(t1, Tf))$ (††), we derive $(Tf \geq \tau_2,\ t1 \leq Tf,\ Tf = t1 + r_j,\ true)$, where the constraint $Tf \geq \tau_2,\ t1 \leq Tf,\ Tf = t1 + r_j$ is satisfiable, because $t1 \geq \tau_2$ and $r_j \geq 0$. Thus, in this case, we have obtained a c-derivation of the form $(\widetilde{\delta})$, because $(Tf \geq \tau_2,\ t1 \leq Tf,\ Tf = t1 + r_j,\ true)$ is equivalent to $(c_i(tf'), true)$, with $tf' = t1 + r_j$.

In Case (2B) from the constrained goal (††) we derive $(Tf \geq \tau_2,\ t1' \leq Tf,\ newY_j(t1', Tf))$, where $t1' = t1 + m_j$, with $t1' \geq t1$ (recall that $m_j \geq 0$), and the constraint $Tf \geq \tau_2,\ t1' \leq Tf$ is satisfiable. Now, in this case, we can continue the c-derivation by applying clause C_2 to the newly derived constrained goal $(Tf \geq \tau_2,\ t1' \leq Tf,\ newY_j(t1', Tf))$. Then, after k steps, with $k < n$, we derive a constrained goal $(Tf \geq \tau_2,\ t1^{(k)} \leq Tf,\ newZ_j(t1^{(k)}, Tf))$, with $t1^{(k)} \geq t1 > \tau_2$, to which C_{k+1} is applicable. Thus, for $k = n-1$, we derive a constrained goal to which the constrained fact C_n is applicable, like in Case (2A), and we have constructed a c-derivation of the form $(\widetilde{\delta})$ by using $C_1, \ldots, C_n$. Since $(\widetilde{\delta})$ is *strictly shorter* than (δ), by iterating the process which led us from the c-derivation (δ) to the c-derivation $(\widetilde{\delta})$, we conclude that there exists a successful c-derivation from $(true, reachProp_i)$ in I_{inst} which is not up-cycling.

Thus, in order to decide whether or not $solve(I_{sp}, reachProp(U, C)$ returns a satisfiable answer constraint, we can explore, for all tuples u, c of integers in the given finite intervals, all (finitely many) c-derivations from $(true, reachProp_i)$ in I_{inst} which are not up-cycling, where $reachProp_i$ is the predicate corresponding to $reachProp(u, c)$. If the constrained goal $(true, reachProp_i)$, corresponding to $reachProp(u, c)$, has a successful c-derivation in I_{inst}, then the decision algorithm returns the answer constraint $U = u,\ C = c$. Otherwise, the decision algorithm returns *false*. In particular, the search for a successful (not up-cycling) c-derivation from $(true, reachProp_i)$ in I_{inst} can be done by arranging the c-derivations in a *c-derivation tree* [22] and traversing that tree in a depth-first manner, as done by most CLP systems. □

Finally, we note that the decision algorithm for $\mathit{reachProp}(U, C)$ queries can be extended to a decision algorithm for queries Q of the form $\mathit{reachProp}(U, C) \wedge \varphi(U, C)$, where $\varphi(U, C)$ is any (possibly quantified) *LIA* formula whose free variables are among the ones in $\{U, C\}$. Indeed, $\mathit{solve}(I_{sp}, Q)$ returns an answer constraint iff there exist two tuples u and c of integers belonging to the finite intervals over which U and C range, such that: (i) $I_{sp} \cup \mathit{LIA} \models \mathit{reachProp}(u, c)$, and (ii) $\mathit{LIA} \models \varphi(u, c)$. Point (i) can be checked by the decision algorithm shown in the above proof, and the Point (ii) is decidable, because *LIA* is a decidable theory.

D. Proof of total correctness of the controllability algorithms WCA and SCA

We present the proof for WCA. The proof for SCA is similar, and we omit it.

In order to show the total correctness of WCA we first prove some lemmas.

Lemma D.1. (Property of *solve*)
Let us consider n (≥ 0) constraints $a_1(U, C), \ldots, a_n(U, C)$. Let $\widetilde{a}(U, C)$ be an abbreviation for $a_1(U, C) \vee \ldots \vee a_n(U, C)$. For $n = 0$, $\widetilde{a}(U, C)$ is *false*. Let $\mathit{lm}(I_{sp}, \mathit{LIA})$ denote the least *LIA*-model of I_{sp}, where the function and predicate symbols of *LIA* are interpreted as expected.
We have that: if $\mathit{solve}(I_{sp}, \mathit{reachProp}(U, C) \wedge \neg\exists C'. \widetilde{a}(U, C')) = \mathit{false}$, then $\mathit{lm}(I_{sp}, \mathit{LIA}) \models \forall U, C. (\mathit{reachProp}(U, C) \rightarrow \exists C'. \widetilde{a}(U, C'))$.
In particular, for $n = 0$, we have that if $\mathit{solve}(I_{sp}, \mathit{reachProp}(U, C)) = \mathit{false}$, then $\mathit{lm}(I_{sp}, \mathit{LIA}) \models \forall U, C. (\mathit{reachProp}(U, C) \rightarrow \mathit{false})$.

Proof:
We make the proof by contradiction. We assume that $\mathit{lm}(I_{sp}, \mathit{LIA}) \not\models \forall U, C. (\mathit{reachProp}(U, C) \rightarrow \exists C'. \widetilde{a}(U, C'))$. Thus, by definition of a model, we have that there exist u and c such that $\mathit{lm}(I_{sp}, \mathit{LIA}) \models \mathit{reachProp}(u, c) \wedge \neg\exists C'. \widetilde{a}(u, C')$, and hence $\mathit{lm}(I_{sp}, \mathit{LIA}) \models \mathit{reachProp}(u, c)$ (†1) and $\mathit{LIA} \models \neg\exists C'. \widetilde{a}(u, C')$ (†2).

From (†1), by the model intersection property, we get: $I_{sp} \cup \mathit{LIA} \models \mathit{reachProp}(u, c)$ (†3).
From (†2) and (†3) we get: $I_{sp} \cup \mathit{LIA} \models \mathit{reachProp}(u, c) \wedge \neg\exists C'. \widetilde{a}(u, C')$ (†4).

From (†4), since $\varphi(v) \leftrightarrow \forall V. V = v \rightarrow \varphi(V)$, we get:
$I_{sp} \cup \mathit{LIA} \models \forall U, C. (U = u \wedge C = c \rightarrow (\mathit{reachProp}(U, C) \wedge \neg\exists C'. \widetilde{a}(U, C')))$ (†5).
By Theorem 6.1, the existence of the constraint $U = u \wedge C = c$ that satisfies (†5), implies that $\mathit{solve}(I_{sp}, \mathit{reachProp}(U, C) \wedge \neg\exists C'. \widetilde{a}(U, C')) \neq \mathit{false}$. This contradicts the hypothesis of the lemma which states that *solve* returns *false*. □

Lemma D.2. If WCA returns *false*, then $\mathcal{B}$ is not weakly controllable.

Proof:
If WCA returns *false*, then during the execution of WCA there exists a call to *solve* of the form $\mathit{solve}(I_{sp}, \mathit{reachProp}(U, C) \wedge \varphi)$, for some *LIA* formula φ, which returns *false*.

We will make the proof by contradiction. Thus, let us assume that $\mathcal{B}$ is weakly controllable. Hence, by definition, we have that $I_{sp} \cup \mathit{LIA} \models \forall U. \mathit{adm}(U) \rightarrow \exists C. \mathit{reachProp}(U, C)$ holds, and in particular $\mathit{lm}(I_{sp}, \mathit{LIA}) \models \forall U. \mathit{adm}(U) \rightarrow \exists C. \mathit{reachProp}(U, C)$ (†6) holds.

Now there are two cases.

Case (i) During WCA we have that *solve* is called once only.

In this case, by definition of WCA, φ is *true*. Thus, by hypothesis, we have that; $solve(I_{sp}, reachProp(U,C)) = false$. Moreover, by Lemma D.1 (for $n = 0$) $lm(I_{sp}, LIA) \models \forall U\, \forall C.\, (reachProp(U,C) \rightarrow false)$ holds, and this contradicts (†6), because $adm(U)$ is satisfiable.

Case (ii) During WCA we have that *solve* is called more than once.

In this case, before the last call to *solve* which, by hypothesis, returns *false*, we have that $LIA \not\models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$ (†7), where $\widetilde{a}(U,C)$ is the disjunction of all the answer constraints returned by *solve* during the execution of WCA. Since the last call to *solve* returns *false*, by Theorem 6.1 there is no satisfiable *LIA* constraint $w(U,C)$ such that $I_{sp} \cup LIA \models \forall U, C.\, w(U,C) \rightarrow (reachProp(U,C) \wedge \neg \exists C'.\, \widetilde{a}(U,C'))$. By Lemma D.1 (for $n > 0$), $lm(I_{sp}, LIA) \models \forall U, C.\, (reachProp(U,C) \rightarrow \exists C'.\, \widetilde{a}(U,C'))$ (†8) holds. Then, by (†6) and (†8), we have that $LIA \models \forall U.\, adm(U) \rightarrow \exists C'.\, \widetilde{a}(U,C')$, which contradicts (†7). □

Lemma D.3. If WCA returns a satisfiable constraint $\widetilde{a}(U,C)$ such that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$, then $\mathcal{B}$ is weakly controllable.

Proof:

If WCA returns a satisfiable constraint $\widetilde{a}(U,C)$, then the exit condition of the *do-while* loop implies that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$ (†9). By construction, the answer constraint $\widetilde{a}(U,C)$ is of the form $a_1(U,C) \vee a_2(U,C) \vee \ldots \vee a_n(U,C)$, where $a_1(U,C), a_2(U,C), \ldots,$ and $a_n(U,C)$ are the n (≥ 1) answer constraints returned by *solve* during the execution of WCA. By definition of *solve* and WCA we have:

(1) $I_{sp} \cup LIA \models \forall (a_1(U,C) \rightarrow reachProp(U,C))$

(2) $I_{sp} \cup LIA \models \forall (a_2(U,C) \rightarrow reachProp(U,C) \wedge \neg \exists C.\, a_1(U,C))$

...

(n) $I_{sp} \cup LIA \models \forall (a_n(U,C) \rightarrow reachProp(U,C)) \wedge \neg \exists C.\, (a_1(U,C) \vee a_2(U,C) \vee \ldots \vee a_{n-1}(U,C)))$

Thus,

$I_{sp} \cup LIA \models \forall ((a_1(U,C) \vee a_2(U,C) \vee \ldots \vee a_n(U,C)) \rightarrow reachProp(U,C))$, that is,

$I_{sp} \cup LIA \models \forall U, C.\ \widetilde{a}(U,C) \rightarrow reachProp(U,C)$. (†10)

Now from (†9) and (†10) it follows that $I_{sp} \cup LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, reachProp(U,C)$, and hence $\mathcal{B}$ is weakly controllable. □

Now we are ready to show the total correctness of WCA.

Proof:

(i) *Proof of termination of* WCA.

The termination of WCA follows from the termination of *solve* (see Theorem 6.1) and the fact that the duration of each task of any given business process belongs to a finite integer interval, and thus the set of all possible answers that can be returned by *solve*, is finite.

(ii) *Proof of correctness of* WCA.

(ii.1) First let us show that, if $\mathcal{B}$ is weakly controllable, then WCA returns a satisfiable *LIA* constraint $\widetilde{a}(U,C)$ such that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$.

Suppose that $\mathcal{B}$ is weakly controllable. Let us assume, by contradiction, that WCA returns *false* (recall that WCA always terminates). Then, by Lemma D.2, we have that $\mathcal{B}$ is not weakly controllable, which contradicts the hypothesis.

(ii.2) It remains to show that, if $\mathcal{B}$ is not weakly controllable, then WCA returns *false*.

Suppose that $\mathcal{B}$ is not weakly controllable. Let us assume, by contradiction, that WCA returns a satisfiable *LIA* constraint $\widetilde{a}(U,C)$ such that $LIA \models \forall U.\, adm(U) \rightarrow \exists C.\, \widetilde{a}(U,C)$ (recall that WCA always terminates). Then, by Lemma D.3, we have that $\mathcal{B}$ is weakly controllable, which contradicts the hypothesis. □