Specification and Validation of gPROMS Semantics

Damian Nadales Agut Martin Hüfner Michel Reniers

gPROMS is an equation-based modeling language and tool for complex, controlled, hybrid systems, which focuses on the process industry. Semantic-preserving algorithmic transformations of models written in gPROMS are necessary to enable model checking of system specifications, as well as the use of features and libraries not present in gPROMS. The above task requires a formal semantic specification of this language, which was absent until now. We present structured operational semantics (SOS) for gPROMS. To validate it, we give a procedure, obtained from the SOS rules, which given a gPROMS model produces a hybrid automaton that is proved to conform to our SOS specification. To check the validity of the SOS rules, we compare the runs of the original gPROMS models with the runs of their corresponding automata. As a result we obtained a validated formal semantics for gPROMS, and a semantic-preserving transformation to hybrid automata.

1 Introduction

Model-based methods for the design and analysis of control systems are becoming increasingly important in industry since they offer many advantages over traditional design methodologies, such as increased safety, a more economical operation of the controlled system, and a significant reduction of the design duration, which translates to significant financial savings. In recent years, sophisticated methods, formalisms, and tools have been developed for all stages of control systems design, ranging from requirements analysis and simulation to automatic synthesis of optimal and robust controllers and implementation verification [19].

One of these tools is gPROMS [27], an equation-based modeling language for describing and simulating complex, controlled, hybrid systems, which focuses on the process industry. The modeling paradigm adopted by this language is similar to the view of a process engineer, and therefore separates plant (MODEL in gPROMS) and controller (TASK or PROCESS). Besides modeling, simulation, and plotting, gPROMS provides powerful optimization and parameter estimation capabilities and can also work with integral and partial differential equations (IPDAEs). This means that gPROMS can be used to optimize models, which were created with other modeling tools in a model-based design flow. Conversely, in the development of embedded and hybrid-systems, the use of gPROMS can be complemented with the use of other formalisms (such as hybrid automata) and their associated tools (such as model-checkers), which can model other aspects of the system under consideration.

A major obstacle in the application of model-based techniques is the incompatibility of the tools that are employed at different stages of the development phase. Most of the existing software tools are based on different model formalisms, and existing export and import functions are mainly limited to the exchange of results, see e.g. [29]. The translation of models must therefore be carried out manually, which is time-consuming, error-prone, and expensive.

Automatic transformations between models arise as a promising approach to overcome this obstacle. If multiple formalisms are involved in these automatic transformations, a common interchange format is required to avoid the need for many bi-lateral translators [7].

Currently, the Compositional Interchange Format (CIF) [4] is being developed in the European project MULTIFORM [22], which is intended to serve as an interchange format between different formalisms for the specification, simulation, and validation of hybrid and timed systems. In particular, it is

Submitted to:

possible to translate CIF models to timed and hybrid automata [24, 15], and in this manner CIF models can be verified using tools such as Uppaal [8] or SpaceEx [14].

It is our goal to develop gPROMS to CIF model transformations, since this enables the verification of gPROMS specifications, or the use of features or equipment libraries (commercial or user built) present in other simulation languages and tools, which are not present in gPROMS. For instance, Modelica is a language for which a vast range of libraries exist and are continuously created, such as the Modelica Standard Library and specialized libraries, like those for the modeling of thermo-hydraulic power generation processes [9].

To accomplish this, we can resort to *model transformations* that enable gPROMS specifications to be translated to different formalisms such as hybrid process algebra or hybrid automata. These transformations can be done in an informal way, based on an intuitive understanding of the source and target language. However transformations tend to be complex, and there is no guarantee that the transformations are *semantic-preserving*. For the development of mission critical systems this is a necessary requirement, since properties of gPROMS specifications validated using other formalisms can give false positives. To enable mathematically proved semantic-preserving transformations we need a formal semantics for the languages which are going to be linked. However, till now gPROMS lacked a formal semantics.

This work constitutes an effort for providing a formal semantics for a subset of gPROMS. The semantics is described using *structured operational semantics* (SOS) [26]. In a nutshell, SOS semantics is defined by giving inference rules. This makes it possible to associate a *hybrid transition system*(HTS) [11] with each gPROMS specification. Hybrid transition systems are widely used for formalizing the behavior of hybrid languages [10, 6]. We have chosen SOS for specifying the semantics of gPROMS since it is a suitable framework to characterize the behavior of structured languages [28], as well as interacting processes [3].

Obtaining a set of SOS rules for gPROMS is only half of the work. These rules are basically definitions, and as such, it is difficult to determine how well they describe the actual implementation of the existing language. Even though the SOS description originates from the observed behavior of the simulator, when considering a complete gPROMS specification, it is hard to keep track of what the results of a simulation should be according to our semantics.

A possible way to validate semantics is to take the same approach as in process algebra, by stating expected properties of the language constructs, such as “parallel composition is commutative”, and then check that these hold [6]. However, this does not work for our case. The actual semantics of the language is only available by means of its simulator, and we may be proving properties for our semantics that are not valid in the implementation. Secondly, typical properties found in process algebraic languages either cannot be stated in gPROMS, or they do not hold. For instance, parallel composition does not seem to be commutative, according to the experiments we performed in the simulator.

Another possibility for validating the semantics is comparing the actual runs of the existing interpreter or simulator, with the expected behavior, as defined by our semantics. For a complex language such as gPROMS, this task cannot be made manually, and construction of an *interpreter* for the SOS rules is required. This was the approach taken for validating the semantics of the discrete subset of Stateflow [16]. The main obstacles for this are, on the one hand, the effort required to build such an interpreter (specially when dealing with hybrid languages); and, on the other hand, the challenge of ensuring that the implementation of the interpreter is correct with regard to the SOS specification.

In addition to the formal semantics of gPROMS, we propose a method for obtaining a hybrid automaton from any given gPROMS specification, in such a way that they are equivalent modulo our semantics. Then, by using the CIF simulator, whose formal semantics is well known [4], we obtain an interpreter for

our semantics, together with a mathematical proof that this interpreter conforms to the SOS rules. The procedure for obtaining hybrid automata out of gPROMS specifications is *derived* from the SOS rules, in a formal and stepwise manner. As a byproduct we obtained a semantic-preserving transformation of gPROMS to hybrid automata.

This report is divided into two parts. The fist part is devoted to the semantic description of gPROMS. Section 2 presents the syntax of the language, Section 3 introduces our semantic framework, and the formal semantics of gPROMS is explained in Section 4. The second part gives an account of the validation process. In Section 5 we describe the procedure for obtaining hybrid automata form gPROMS specifications, and in Section 6 we compare the runs of gPROMS specifications, obtained by means of the gPROMS simulator, with the runs of their corresponding hybrid automata, obtained by means of the CIF simulator.

2 Specifications and system descriptions

In this work we formalize the semantics of a subset of gPROMS. This subset was chosen in such a way that it contains a minimal set of basic concepts, in the sense that every other construct can be expressed in terms of basic ones, and are therefore regarded as *modeling extensions*. This section provides an informal description of the subset of gPROMS we consider in this work, as well as the formal syntax necessary for the mathematical modeling of its semantics.

A gPROMS specification models a hybrid system, and it is divided into two main parts: a model part, which describes the *continuous behavior* of the system; and a process part, which describes the *discrete behavior* of the system.

To illustrate gPROMS syntax and informal semantics, we model a system consisting of a ball, which is dropped from a certain height and bounces repeatedly against the floor. We want to measure the evolution in time of the following variables: the height of the ball, its velocity, the number of times it hits the ground, and a timer that keeps track of the simulation time. In gPROMS each variable has an initial value, a minimum value, and a maximum value. These constraints can be specified using a *type declaration*. For the model variables considered here, these declarations are shown in Listing 1, where a type declaration of the form

```
DECLARE TYPE
    name=i:mn:mx
END
```

states that all variables of type `name` have `i`, `mn`, and `mx` as their initial, minimum, and maximum value, respectively.

Listing 1: Type declarations for the model of a bouncing ball.

```
DECLARE TYPE
    height=0.0:-500.0:500.0
END
DECLARE TYPE
    number=0.0:0.0:4.2949673e9
END
DECLARE TYPE
    timer=0.0:0.0:4.2949673e9
END
DECLARE TYPE
```

```
    velo=0.0:-500.0:4.2949673e9
END
```

The continuous dynamics of the system can be described as shown in Listing 2. This model consists of the following elements:

- Two parameters. The constant dampening factor `e` and the gravity constant `g`. The parameter of a model are variables whose values remain constant during the execution (simulation) of the system.
- Four variables. The height of the ball `h`, its velocity `v`, the number of times the ball hits the ground `n`, and a clock `t` that keeps track of the simulation time. The value of variables can change during simulation.
- A selector declaration, which introduces the *selector variables* of the model. Selector variables are used in `CASE` statements to identify the different cases (see below).
- Three equations. The first (`$h=v`) and last (`$t=1.0`) equations are simple differential equations, where `$x` denotes the derivative of variable `x` and the second equation is a case equation, which states that equation `$v=-(g)` must hold if `ballPosition` equals `air` and for as long as `h<=0.0 AND v<=0.0` is not satisfied, and `$v=0.0` must hold if `ballPosition` equals `ground` and for as long as `h <=0.0 AND v <=0.0` is satisfied. At every point in time during the simulation these three equations have to be satisfied.

Listing 2: Continuous behavior of a bouncing ball.

```
MODEL Ball
    PARAMETER
        e AS REAL
        g AS REAL
    VARIABLE
        h AS height
        v AS velo
        n AS number
        t AS timer
    SELECTOR
        ballPossition AS (air, ground)
    EQUATION
        $h=v;
        CASE ballPosition OF
            WHEN air:
                $v=-(g);
                SWITCH TO ground IF (h<=0.0 AND v<=0.0);
            WHEN ground:
                $v=0.0;
                SWITCH TO air IF NOT ((h<=0.0 AND v<=0.0));
        END
        $t=1.0;
END
```

Listing 5 shows the model of the discrete behavior of the system, which consists of the following elements:

- A model instantiation, of the form `b AS Ball`. The effect of this sentence is to introduce in the system all the equations appearing in `Ball`, where each variable `x` is replaced by `b.x`.

- A parameter instantiation, specified using the SET keyword, which assigns initial values to the model parameters.
- An initial assignment of values to the model variables. For discrete variables, this assignment is carried out using the ASSIGN keyword. For continuous variables the INITIAL keyword is used instead.
- An initial assignment of values to selector variables, specified by the keyword INITIALSELECTOR.
- An algorithm, which describes the discrete evolution of the values of variables, and is specified using the SCHEDULE keyword. The PARALLEL construct is used to execute sentences in parallel, the semantics of the WHILE command is standard, REINITIAL and RESET statements can be interpreted as assignments, and the CONTINUE UNTIL command delays until its accompanying condition is true. The formal semantics of the sentences and control structures used in this example is explained in Section 4.

Listing 3: Discrete behavior of a bouncing ball.

```
PROCESS Bouncing
    UNIT
        b AS Ball
    SET
        b.e:=0.8;
        b.g:=9.81;
    ASSIGN
        b.n:=0.0;
    INITIALSELECTOR
        b.ballPosition:=b.air;
    INITIAL
        b.h=10.0;
        b.v=0.0;
        b.t=0.0;
    SCHEDULE
    PARALLEL
        WHILE TRUE DO
            SEQUENCE
                CONTINUE UNTIL b.h<0.0
                REINITIAL
                    b.v
                WITH
                    b.v=(-(b.e)*OLD(b.v));
                END
                REINITIAL
                    b.h
                WITH
                    b.h=0.0;
                END
                RESET
                    b.n:=(OLD(b.n)+1.0);
                END
            END
        END
        SEQUENCE
```

```
            CONTINUE UNTIL 12.0<=b.t
            STOP
        END
    END
END
```

We have seen, the three main components of gPROMS specifications: type declarations, models, and processes. Thus, we can define the set of all gPROMS specifications, denoted as gPROMS_{core}, using the Extended Backus Naur Form (EBNF) meta-syntax as follows.

Definition 1 (gPROMS specifications)**.** *The set of all gPROMS specifications, is defined by the following grammar:*

$$\langle gPROMS_{core} \rangle \rightarrow \langle DECLARE_{core} \rangle^* \ \langle MODEL_{core} \rangle \ \langle PROCESS_{core} \rangle$$

Here $DECLARE_{core}$ denotes the set of all type declarations, $MODEL_{core}$ is the set of all gPROMS models, and $PROCESS_{core}$ is the set of all gPROMS processes.

The syntax of the non-terminals $DECLARE_{core}$, $MODEL_{core}$, and $PROCESS_{core}$ is omitted here, since it is not relevant for the current work. We present the details in Appendix B.

2.1 The behavioral subset of gPROMS

In the previous example, it can be observed that gPROMS specifications mix models of the continuous and discrete behavior of a system (equations and schedules, respectively), with *modeling extensions* such as model instantiations, and variable aliasing, among others.

The semantics we want to define is only concerned with the behavioral aspects of a system. By defining SOS rules only for the equations and schedule specifications we can obtain a simple and lean semantics. Naturally, the question remains about what is the meaning of a gPROMS specification (containing modeling extensions). To this end, we implemented a *transformation function*, named gs2gsd (see Appendix C), that takes a gPROMS specification and returns a tuple, which contains the equations and schedule defining the behavior of the system.

The transformation function gs2gsd extracts the equations that are instantiated in the unit section of a gPROMS process, and the schedule part of a process. For the model of the bouncing ball described before, function gs2gsd returns the equations depicted in Listing 4, and the schedule part of Listing 2.

Listing 4: Instantiated equations of the bouncing ball model

```
$b.h=b.v;
CASE b.ballPosition OF
    WHEN b.air:
        $b.v=-(b.g);
        SWITCH TO b.ground IF (b.h<=0.0 AND b.v<=0.0);
    WHEN b.ground:
        $b.v=0.0;
        SWITCH TO b.air IF NOT ((b.h<=0.0 AND b.v<=0.0));
END
$b.t=1.0;
```

Let $\mathbb{E}$ and $\mathbb{S}$ denote the set of gPROMS equations and schedules respectively. Their combination is called a *system description*, and is formally defined as follows.

Definition 2 (System descriptions)**.** *A gPROMS system description is a tuple of the form:*

$$\langle es, S \rangle$$

where $es \in \mathbb{E}^*$ *is a list of equations, and* $S \in \mathbb{S}$ *is a schedule.*

Throughout this work, we make use of lists and operations on lists:

- $[\,]$ denotes the empty sequence.
- Given an element x and a list xs, $x : xs$ is the list obtained from appending x at the beginning of xs.
- Given a list xs, $\#xs$ denotes the number of elements of xs.
- Given a list xs and a natural number i, $xs.i$ denotes the element of xs at position i. The elements of a list are numbered from 0.
- Given a function $f : A \to B$, and elements $a \in A$ and $b \in B$, $f[a : b] : A \to B$ and for all $x \in A$ we have:

$$f[a : b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

  As a particular case of the operator above, given a sequence xs, an element x, and a natural number i, $xs[i : x]$ denotes the list resulting from replacing in xs the element at position i by x, and $xs \setminus i$ is the list resulting after removing the element at position i from xs.

The formal definitions of the set of equations and schedules are given next, where $\mathscr{V}$ is the set of all gPROMS variables, Λ is the set of all possible values variables can assume, $\mathscr{P}$ is the set of all gPROMS predicates, and $\mathscr{E}$ denotes the set of all gPROMS expressions. We do not discuss the exact syntax of predicates and expressions in the present work. We assume that it is possible to write boolean combinations of equalities and inequalities, this is, expressions like $x + y \leq 17 \wedge y \leq 7$ or $v = O \vee 10 \leq h$.

Definition 3 (gPROMS equations)**.** *The set of all gPROMS equations, denoted as* $\mathbb{E}$*, is defined by the grammar of Table 1.*

$$\begin{aligned} \mathbb{E} &\to \mathscr{E} = \mathscr{E} \\ &\quad \mid \textbf{CASE } \mathscr{V} \textbf{ OF } W^+ \textbf{ END} \\ W &\to \textbf{WHEN } \Lambda \ : \ \mathbb{E}^+ \ SC^+ \\ SC &\to \textbf{SWITCH TO } \Lambda \textbf{ IF } \mathscr{P}^+ ; SC^+ \end{aligned}$$

Table 1: Gramar of gPROMS equations

Definition 4 (gPROMS schedules)**.** *The set of all gPROMS schedules, denoted as* $\mathbb{S}$*, is defined by the grammar of Table 2.*

Henceforth, *gPROMS compositions* is used to refer to the set denoted as $\mathscr{G}$, which contains all gPROMS specifications, system descriptions, equations, list of equations, and schedules.

Conditions that follow `SWITCH TO` and `CONTINUE UNTIL` statements, are called *switching conditions*. These define *urgency constraints* on the passage of time. For instance, the statement:

```
CONTINUE UNTIL volume <= 2
```

$$
\begin{aligned}
\mathbb{S} \rightarrow\ & \textbf{SEQUENCE}\ \mathbb{S}^{+}\ \textbf{END} \\
|\ & \textbf{PARALLEL}\ \mathbb{S}^{+}\ \textbf{END} \\
|\ & \textbf{WHILE}\ \mathscr{P}\ \textbf{DO}\ \mathbb{S}\ \textbf{END} \\
|\ & \textbf{IF}\ \mathscr{P}\ \textbf{THEN}\ \mathbb{S}\ \textbf{ELSE}\ \mathbb{S}\ \textbf{END} \\
|\ & \textbf{CONTINUE UNTIL}\ \mathscr{P}\ ; \\
|\ & \textbf{RESET}\ \mathscr{V} := \mathscr{E}\ \textbf{END} \\
|\ & \textbf{REINITIAL}\ \mathscr{V}\ \textbf{WITH}\ \mathscr{V} = \mathbb{E}\ \textbf{END} \\
|\ & \textbf{SWITCH}\ \mathscr{V} := \Lambda\ \textbf{END} \\
|\ & \textbf{STOP}
\end{aligned}
$$

Table 2: Gramar of gPROMS schedules

allows time delays as long as `2 < volume`. If this condition does not hold, then time cannot longer progress, and the control is passed to the statement that follows (if any).

There are switching conditions that are troublesome from the semantic modeling point of view. Consider the statement

```
CONTINUE UNTIL x < 2
```

where `x` is a continuous variable whose value changes at rate 1. The intuition behind such a statement is that it must terminate as soon as `x` is greater than 2. If initially `x` has value 0, then it is not possible to find the smallest time point at which this condition becomes enabled. The gPROMS simulator has no problem with this due to the discretization of the integration process, but from a pure mathematical point of view this forces us to adapt the semantics to consider these cases. In addition, conditions such as `v = 1`, where `v` is a continuous variable, pose additional problems, since the simulator performs numerical approximations and as a consequence, a condition that is mathematically valid, may not hold due to the presence of round off errors.

To avoid making compromises in the semantics that can obscure it, in this work we restrict the set of allowed predicates in switching conditions. These restrictions on the switching conditions do not limit the interesting gPROMS models, since they are coherent with good modeling guidelines. Most of the models written in practice satisfy these constraints.

Definition 5 (Continuous-closed constraints and predicates)**.** *A continuous-closed constraint is a predicate of the form:*

$$f \bowtie 0$$

where either f is a continuous function of time, and $\bowtie \in \{\leq\}$; or f is a constant function, and $\bowtie \in \{=, \neq\}$.

A continuous closed predicate is a boolean predicate whose atomic propositions are semantically equivalent to a continuous-closed constraint.

Continuous-closed predicates are important for the translation from gPROMS specifications to hybrid automata since they allow us to implement urgency conditions using invariants.

3 Semantic framework

The SOS rules associate a hybrid transition system (HTS) [11] with each gPROMS composition. The states of the HTS are of the form: (p, σ) where p is a gPROMS composition, $\sigma \in \mathscr{V} \rightarrow \Lambda$ is a valuation,

and Λ is the set of all possible values. Henceforth, we use $\Sigma \triangleq \mathscr{V} \to \Lambda$ to refer to the set of all valuations, and $\$\Sigma \triangleq (\mathscr{V} \cup \{\$x \mid x \in \mathscr{V}\}) \to \Lambda$ to refer to the set of all valuations which include the dotted version of variables. There are six kinds of transitions:

Action transitions of the form $(p,\sigma) \to (p',\sigma')$, model the execution of an action by the composition p in initial valuation σ, which transforms p into p' and results in a new valuation σ'.

Terminating transitions of the form $(p,\sigma) \to (\checkmark,\sigma')$, model the execution of an action by the composition p in initial valuation σ, which terminates with final valuation σ'.

Halting transitions of the form $(p,\sigma) \to (\bot,\sigma')$, model the execution of an action by the composition p in initial valuation σ, which stops the simulation with final valuation σ'.

Time transitions of the form $(p,\sigma) \overset{\rho}{\longmapsto} (p',\sigma')$, model the passage of time in p that results in a new composition p' and a final valuation σ'. Here $\rho \in [0,t] \to \$\Sigma$ is a variable trajectory that returns a valuation for each point in $[0,t]$, where $t \in \mathbb{T}$, and $\mathbb{T}$ is the set of all time points.

No-switch transitions (predicates) of the form $(e,\sigma) \nrightarrow$, model the fact that the equation (or list of equations) e cannot switch its active equations in valuation σ (a composition can change its active equations if a switch condition of a case statement becomes true).

Initialization transitions of the form $(e,\sigma) \overset{\text{init}}{\rightsquigarrow} \ell s$, allow to choose new active equations in e according to the conditions that are valid in σ. Valuation ℓs contains the new values of selector variables after the initialization. Section 4.3 provides more details.

Even though in the present work predicates are treated as abstract entities, we assume that a satisfaction relation $\sigma \models u$ is defined, which expresses that predicate $u \in \mathscr{P}$ is satisfied (*i.e.* it is true) in valuation σ. For an expression $e \in \mathscr{E}$ and a valuation σ, $\sigma(e)$ denotes the value of expression e in valuation σ. Additionally, making some notational abuse, we define a satisfaction relation involving variable trajectories, this is: functions whose domain is a dense interval, and whose range is a valuation.

$$(\rho \models u) \triangleq \langle \forall s : s \in \text{dom}(\rho) : \rho(s) \models u \rangle$$

For modeling differential equations, the above satisfaction relation does not suffice: not only these kinds of equations have to be satisfied throughout the delay, but also dotted variables must be the time derivatives of continuous variables appearing in the equation. For this reason in this work we rely upon the definition of a satisfaction relation of the form:

$$\rho \models_{\text{flow}} u$$

which expresses the fact that boolean expression u is true in all the points of the trajectory ρ, and the dotted variables are the time derivatives of continuous variables in the interval $\text{dom}(\rho)$. In this way, relation $\models_{\text{flow}}$ allows us to model constraints in the joint evolution of a variable and its dotted version. For differential equations of the form:

$$F(\mathbf{x},\$\mathbf{x}) = 0$$

the satisfaction relation $\rho \models_{\text{flow}} F(\mathbf{x},\$\mathbf{x}) = 0$ could be defined as: *"*$\mathbf{x}$ *is the derivative of* $\mathbf{x}$ *in the interval* $\text{dom}(\rho)$*, and for each* $s \in \text{dom}(\rho)$ *we have:* $\rho(s) \models F(\mathbf{x},\$\mathbf{x}) = 0$*"*.

This notion of satisfaction for differential equation substitutes, in the context of this work, the concept of *dynamic types* [20, 5]. Note also that $\rho \models_{\text{flow}} e$ implies $\rho \models e$ but the converse does not necessarily hold.

4 Semantics

In this section we describe the semantics of the gPROMS language constructs. We begin by giving semantics to gPROMS specifications using functions that extract the relevant components out of them. Then we turn our attention to the interplay between equations and schedules, as described in Section 4.2. Finally, in the last two subsections, we specify the semantics of list of equations, equations, and schedules.

4.1 gPROMS specifications

In this section we consider the semantics of gPROMS specifications, in terms of hybrid transitions systems. To this end, we make use of a function, named 'gs2gsd', which returns the system description (equations, and schedule term) associated to a given gPROMS specification G. The details of such a definition are presented in Appendix C. Also, we have defined a function 'iconds', which extracts the initial conditions of gPROMS specifications. The use of these functions allows us to give semantics to concepts such as model and parameter instantiations, or variable aliasing without obscuring our SOS rules.

Rules 1 and 2 model the behavior of gPROMS specifications. A specification G can perform an action or time transition if its associated system description ($\text{gs2gsd}(G)$) can, and the initial conditions of the specification $\text{iconds}(G)$ are satisfied in the initial valuation.

$$\frac{\sigma \models \text{iconds}(G) \quad (\text{gs2gsd}(G), \sigma) \to (p', \sigma')}{(G, \sigma) \to (p', \sigma')}\ 1 \qquad \frac{\sigma \models \text{iconds}(G) \quad (\text{gs2gsd}(G), \sigma) \xmapsto{\rho} (p', \sigma')}{(G, \sigma) \xmapsto{\rho} (p', \sigma')}\ 2$$

Table 3: Explicit rules for gPROMS specifications

In the semantic rules for gPROMS specifications we do not assign a meaning to those constructs that have no behavior but introduce type constraints, such as upper and lower bounds, variable and parameter declarations. Since we are only concerned with the behavioral subset of gPROMS we assume the model is well typed.

4.2 gPROMS system descriptions

In this section we formalize the semantics of system descriptions. A system description of the form

$$\langle es, S \rangle$$

consists of a list es of equations that determines the continuous evolution of variables, and a schedule term S that describes the discrete changes in the values of those variables (changes in the system).

To model the fact that equation switches, via case statements, take precedence over schedule actions we use a *no-switching predicate*, which holds only for equations or lists of equations in which no switch is possible. Table 4 presents the formal details. This predicate can be defined in terms of action transitions, using negative premises. However, our symbolic semantics approach (see Section 5.1) forces us to give a constructive definition of this predicate.

The semantic rules for system descriptions are presented in Table 5. The first two rules (Rules 6 and 7) describe the action behavior of system descriptions. These rules state that if a sub-process performs an

$$\frac{}{(e_0 = e_1, \sigma) \not\rightarrow} \; 3$$

$$\frac{\begin{array}{c}(\textbf{WHEN}\ x\ :\ es_x\ ts_x\ ) \in ws \\ \sigma \models (v = x \wedge \langle \bigwedge y, c : (\textbf{SWITCH TO}\ y\ \textbf{IF}\ c;) \in ts_x : \neg c \rangle) \\ (es_x, \sigma) \not\rightarrow\end{array}}{(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END}, \sigma) \not\rightarrow} \; 4$$

$$\frac{(e, \sigma) \not\rightarrow, \qquad (es, \sigma) \not\rightarrow}{(e : es, \sigma) \not\rightarrow} \; 5$$

Table 4: Explicit rules for no-switch transitions

action, then the whole system performs an action as well. Note that lists of equations can perform actions, which is required to model the change of an equation caused by a case statement. Rule 8, expresses that if the schedule stops, then the whole system also halts. A time transition is possible in a gPROMS system description, if this time transition can be generated by the list of equations, and by the schedule.

$$\frac{(es, \sigma) \rightarrow (es, \sigma')}{(\langle es, S \rangle, \sigma) \rightarrow (\langle es, S \rangle, \sigma')} \; 6 \qquad \frac{(es, \sigma) \not\rightarrow, \qquad (S, \sigma) \rightarrow (S', \sigma')}{(\langle es, S \rangle, \sigma) \rightarrow (\langle es, S' \rangle, \sigma')} \; 7$$

$$\frac{(es, \sigma) \not\rightarrow, \qquad (S, \sigma) \rightarrow (\bot, \sigma')}{(\langle es, S \rangle, \sigma) \rightarrow (\bot, \sigma')} \; 8 \qquad \frac{\begin{array}{c}(es, \sigma) \overset{\rho}{\longmapsto} (es, \sigma'), \\ (S, \sigma) \overset{\rho}{\longmapsto} (S', \sigma')\end{array}}{(\langle es, S \rangle, \sigma) \overset{\rho}{\longmapsto} (\langle es, S' \rangle, \sigma')} \; 9$$

Table 5: Explicit rules for gPROMS system descriptions

4.3 Equations

The explicit rules for equations are shown in Table 6. Rule 10 models the time behavior of equalities, and it states that time delays are possible only if the equality is satisfied throughout the time delay, and for each continuous variable `x`, `$x` must behave as its derivative. The latter requirement is formalized by using the $\models_{\text{flow}}$ predicate. Equalities do not have action behavior, and therefore there is no corresponding action rule. Here given a function f and a set A, $f \restriction$ is the domain restriction of f to A.

When a `CASE` statement performs a switch in its active equations, it may be the case that the new equations contain case statements as well, in which case active equations must be also chosen according to the switching conditions of the inner `CASE` statements. To model nested switches in case statements we need an *initialization transition for case statements*. This transition picks the appropriate values of selector variables according to the conditions that hold in the current valuation. The rules of Table 7 model this. Note that, unlike action transitions, initialization transitions do not take into account the current value of the selector variables. This is why we need initialization transitions when switching to a new `CASE` branch. In the aforementioned rules, we are assuming all variables used in case statements are unique.

Unlike equations, case statements can perform actions. A discrete event takes place whenever the switching condition of the active branch is satisfied. The change of the active branch is done by a

$$\frac{0<t, \qquad \mathrm{dom}(\rho)=[0,t], \qquad \sigma=\rho(0)\restriction_{\mathscr{V}}, \qquad \rho \models_{\mathrm{flow}} e_0=e_1}{(e_0=e_1,\sigma)\overset{\rho}{\longmapsto}(e_0=e_1,\rho(t)\restriction_{\mathscr{V}})}\ 10$$

$$\frac{\begin{array}{c}(\textbf{WHEN}\ x\ :\ es_x\ ts_x\ )\in ws,(\textbf{SWITCH TO}\ y\ \textbf{IF}\ c;)\in ts_x,\\ \sigma\models v=x\wedge c, \qquad (\textbf{WHEN}\ y\ :\ es_y\ ts_y\ )\in ws,(es_y,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\ell s'\end{array}}{\begin{array}{c}(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma)\rightarrow\\ (\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},(\{(v,y)\}\cup\ell s')\succ\sigma)\end{array}}\ 11$$

$$\frac{\begin{array}{c}(\textbf{WHEN}\ x\ :\ es_x\ ts_x\ )\in ws,\sigma\models v=x,\\ \sigma\models\langle\bigwedge y,c:(\textbf{SWITCH TO}\ y\ \textbf{IF}\ c;)\in ts:\neg c\rangle,\\ (es_x,\sigma)\rightarrow(es_x,\sigma')\end{array}}{(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma)\rightarrow(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma')}\ 12$$

$$\frac{\begin{array}{c}(\textbf{WHEN}\ x\ :\ es_x\ ts_x\ )\in ws,\sigma\models v=x\\ \rho\models\langle\bigwedge y,c:(\textbf{SWITCH TO}\ y\ \textbf{IF}\ c;)\in ts:\mathrm{closeNeg}(\neg c)\rangle,\\ (es,\sigma)\overset{\rho}{\longmapsto}(es,\sigma')\end{array}}{(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma)\overset{\rho}{\longmapsto}(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma')}\ 13$$

Table 6: Explicit rules for equations

$$\frac{}{(e_0=e_1,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\emptyset}\ 14$$

$$\frac{\begin{array}{c}(\textbf{WHEN}\ x\ :\ es_x\ ts_x\ )\in ws, \qquad (\textbf{SWITCH TO}\ y\ \textbf{IF}\ c;)\in ts_x,\\ (\textbf{WHEN}\ y\ :\ es_y\ ts_y\ )\in ws,\\ \sigma\models c, \qquad (es_y,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\ell s\end{array}}{(\textbf{CASE}\ v\ \textbf{OF}\ ws\ \textbf{END},\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\{(v,y)\}\cup\ell s}\ 15$$

$$\frac{(e,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\ell s, \qquad (es,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\ell ss}{(e:es,\sigma)\overset{\mathrm{init}}{\rightsquigarrow}\ell s\cup\ell ss}\ 16$$

Table 7: Explicit rules for initialization transitions

change in the value of the selector variable. Rule 11 for case constructs formalizes this, where given two functions $f:A\rightharpoonup C$ and $g:A\rightharpoonup C$, function $f\succ g:A\cup B\rightharpoonup C$ is defined as follows:

$$f\succ g(x)=\begin{cases}f(x) & \text{if } x\in\mathrm{dom}(f)\\ g(x) & \text{if } x\in\mathrm{dom}(g)\setminus\mathrm{dom}(f)\end{cases}$$

The equations inside a case construct can also contain other case constructs, which may cause switches in their active branches. Rule 12 for case constructs covers this case. Here, a switch caused by the inner equations is allowed only if no switching is possible in the enclosing case construct. This behavior was confirmed by the output of the gPROMS simulator.

The case construct allows time delays as long as no switch is possible during the time delay, as expressed in Rule 13. Here we use a *syntactic* closeNeg function to obtain predicates that express the closed negation of predicates, and it is defined below.

Definition 6 (Closed negation of predicates)**.** *The function* closeNeg $\in \mathscr{P} \to \mathscr{P}$ *is defined recursively as follows:*

$$\begin{aligned}
\text{closeNeg}(f < 0) &= 0 \leq f \\
\text{closeNeg}(f \leq 0) &= 0 \leq f \\
\text{closeNeg}(f = 0) &= f \neq 0 \\
\text{closeNeg}(f \neq 0) &= f = 0 \\
\text{closeNeg}(\neg p) &= p \\
\text{closeNeg}(p \wedge q) &= \text{closeNeg}(p) \vee \text{closeNeg}(q) \\
\text{closeNeg}(p \vee q) &= \text{closeNeg}(p) \wedge \text{closeNeg}(q)
\end{aligned}$$

For example, we have:

$$\text{closeNeg}(2 \leq x \wedge x \leq 5) \equiv x \leq 2 \vee 5 \leq x$$

In Rule 13, we use the closed negation of a predicate instead of its negation. This is because time must be able to pass up to a point in which the guard becomes true. For instance, consider a gPROMS `CASE` statement, whose active case branch contains a switching condition of the from `1 <= x`. Assume that the evolution of `x` is constrained by the differential equation `$x = 1`, and initially `x = 0`. Then, time should be able to pass up to a point in which the guard becomes true. If the condition for time to pass is `x < 1`, then the switch will never occur (according to the formal semantics). By negating and closing the switching condition, we obtain that the condition for time to pass in this case is `x <= 1`, and as a result the system can switch the active equation when `x` equals `1`.

So far we have been using *sequences of equation specifications*. Now we need to assign semantics to them. This is done as shown in Table 8. Rule 17 for sequences of equations states that the empty list of equations can perform any arbitrary time delay. There are no action rules for empty sequences. The rule for the inductive cases state that time delays must synchronize, whereas action transitions can alternate.

$$\frac{0 < t, \text{dom}(\rho) = [0,t], \sigma = \rho(0) \restriction_{\mathscr{V}}}{([\,], \sigma) \overset{\rho}{\longmapsto} ([\,], \rho(t) \restriction_{\mathscr{V}})} \; 17$$

$$\frac{(e, \sigma) \to (e', \sigma')}{(e : es, \sigma) \to (e' : es', \sigma')} \; 18 \qquad \frac{(es, \sigma) \to (es', \sigma')}{(e : es, \sigma) \to (e' : es', \sigma')} \; 19$$

$$\frac{(e, \sigma) \overset{\rho}{\longmapsto} (e', \sigma'), (es, \sigma) \overset{\rho}{\to} (es', \sigma')}{(e : es, \sigma) \overset{\rho}{\to} (e' : es', \sigma')} \; 20$$

Table 8: Explicit rules for lists of equations

4.4 Schedules

In this section we present the SOS rules for gPROMS schedules. Rule 21 for sequential composition states that a sequence of statements can perform an action if the first statement can. Rule 22 expresses that control is passed to the second statement in the list if the first one terminates. Rule 23 passes

the control to the last statement when all the previous ones terminate[1], and is standard for sequential composition [28]. The halting process ends the sequential and parallel composition. The time behavior of a sequential composition is determined only by the first statement of the list.

$$\frac{(s,\sigma) \to (s',\sigma')}{(\textbf{SEQUENCE}\ s : zs\ \textbf{END},\sigma) \to (\textbf{SEQUENCE}\ s' : zs\ \textbf{END},\sigma')}\ 21$$

$$\frac{(s,\sigma) \to (\checkmark,\sigma'), \qquad 2 \leq \#zs}{(\textbf{SEQUENCE}\ s : zs\ \textbf{END},\sigma) \to (\textbf{SEQUENCE}\ zs\ \textbf{END},\sigma')}\ 22$$

$$\frac{(s,\sigma) \to (\checkmark,\sigma'), \qquad \#zs = 1}{(\textbf{SEQUENCE}\ s : zs\ \textbf{END},\sigma) \to (zs.0,\sigma')}\ 23$$

$$\frac{(s,\sigma) \to (\bot,\sigma')}{(\textbf{SEQUENCE}\ s : zs\ \textbf{END},\sigma) \to (\bot,\sigma')}\ 24$$

$$\frac{(s,\sigma) \overset{\rho}{\longmapsto} (s',\sigma')}{(\textbf{SEQUENCE}\ s : zs\ \textbf{END},\sigma) \overset{\rho}{\longmapsto} (\textbf{SEQUENCE}\ s' : zs\ \textbf{END},\sigma')}\ 25$$

Table 9: Explicit rules for sequential composition

We present the rules for parallel composition in Table 10. Rules 26 and 27 expresses that the execution of any of the sentences that are composed in parallel is interleaved. This differs from actual gPROMS semantics, where the parallel components are executed in the order they appear. We believe this is a shortcoming in the way the simulator is implemented, since in reality, parallel components exhibit true concurrency. By modeling this in our semantics, errors that do not appear in the gPROMS simulation of a given gPROMS model, can be uncovered when analyzing it with other tools, once the original model gets translated to other formalisms, such as hybrid automata, by means of semantic-preserving transformations.

The terminating rule for parallel composition expresses that, if one of the components terminates, then it is removed from the parallel sentences. When two components remain, and one of them terminates, then the control is passed to the remaining one (Rule 28). If one of the components of the parallel composition stops the simulation, then the whole schedule stops as well. The rules for time say that all the components of the parallel composition must synchronize in the time delay.

We present next the rules for control structures: `WHILE` loops, and `CONTINUE UNTIL` statements. Note that:

- The evaluation of the guards generates events, as can be observed when running the gPROMS simulator.
- The evaluation of guards takes precedence over time delays, which corresponds with the behavior we saw in the simulation runs. As a result, in the rules for while loops there are no time transitions specified, which means all guards must be evaluated before a time delay can take place. The conditional `IF` construct follows the same principle.

[1] Alternatively, we could have defined a terminating rule for sequential composition, but this complicates the symbolic semantics when dealing with while loops since we get an infinite symbolic transition system.

$$\frac{(zs.i,\sigma) \to (s',\sigma'), \qquad 0 \le i < \#zs}{(\textbf{PARALLEL } zs \textbf{ END},\sigma) \to (\textbf{PARALLEL } zs[i:s'] \textbf{ END},\sigma')} 26$$

$$\frac{(zs.i,\sigma) \to (\checkmark,\sigma'), \qquad 0 \le i < \#zs, \qquad 3 \le \#zs}{(\textbf{PARALLEL } zs \textbf{ END},\sigma) \to (\textbf{PARALLEL } zs \setminus i \textbf{ END},\sigma')} 27$$

$$\frac{(zs.i,\sigma) \to (\checkmark,\sigma'), \qquad 0 \le i < \#zs, \qquad \#zs = 2}{(\textbf{PARALLEL } zs \textbf{ END},\sigma) \to ((zs \setminus i).0,\sigma')} 28$$

$$\frac{(zs.i,\sigma) \to (\bot,\sigma'), \qquad 0 \le i < \#zs}{(\textbf{PARALLEL } zs \textbf{ END},\sigma) \to (\bot,\sigma')} 29$$

$$\frac{(s,\sigma) \xmapsto{\rho} (s',\sigma'), \qquad (t,\sigma) \xmapsto{\rho} (t',\sigma')}{(\textbf{PARALLEL } [s,t] \textbf{ END},\sigma) \xmapsto{\rho} (\textbf{PARALLEL } [s,t] \textbf{ END},\sigma')} 30$$

$$\frac{\begin{array}{c}2 \le \#zs, \qquad (s,\sigma) \xmapsto{\rho} (s',\sigma'),\\ (\textbf{PARALLEL } zs \textbf{ END},\sigma) \xmapsto{\rho} (\textbf{PARALLEL } zs \textbf{ END},\sigma')\end{array}}{(\textbf{PARALLEL } s : zs \textbf{ END},\sigma) \xmapsto{\rho} (\textbf{PARALLEL } s : zs \textbf{ END},\sigma')} 31$$

Table 10: Explicit rules for parallel composition

$$\frac{\sigma \models b}{\begin{array}{c}(\textbf{WHILE } b \textbf{ DO } s \textbf{ END},\sigma) \to\\ (\textbf{SEQUENCE } s : [\textbf{WHILE } b \textbf{ DO } s \textbf{ END}] \textbf{ END},\sigma)\end{array}} 32$$

$$\frac{\sigma \models \neg b}{(\textbf{WHILE } b \textbf{ DO } s \textbf{ END},\sigma) \to (\checkmark,\sigma)} 33$$

$$\frac{\sigma \models b}{(\textbf{CONTINUE UNTIL } b \;;,\sigma) \to (\checkmark,\sigma)} 34$$

$$\frac{\mathrm{dom}(\rho) = [0,t], \qquad \rho \models \mathrm{closeNeg}(\neg b), \qquad \sigma = \rho \restriction_{\mathcal{V}}}{(\textbf{CONTINUE UNTIL } b \;;,\sigma) \xmapsto{\rho} (\textbf{CONTINUE UNTIL } b \;;,\sigma)} 35$$

Table 11: Explicit rules for control structures

The `CONTINUE UNTIL` command can perform time delays only when its associated switching condition is false. Otherwise it executes a transition into termination. The action rule (Rule 34) is straightforward: only perform the action if the switching condition is true. However the rule for time (Rule 35) requires closing the negation of the predicate to allow time delays up to the point in which the value of the switching condition changes from false to true, as explained in Section 4.3, when dealing with the time behavior of `CASE` statements.

Table 12 presents the explicit rules for atomic statements. Reset, switch, and reinitial actions are urgent. Time delays are not possible. A **STOP** term can only do a transition into halt.

$$\frac{}{(\textbf{RESET}\ x := e\ \textbf{END}, \sigma) \rightarrow (\checkmark, \sigma[x : \sigma(e)])}\ 36$$

$$\frac{}{(\textbf{SWITCH}\ x :=\!y\ \textbf{END}, \sigma) \rightarrow (\checkmark, \sigma[x : y])}\ 37$$

$$\frac{}{(\textbf{REINITIAL}\ y\ \textbf{WITH}\ y = e\ \textbf{END}, \sigma) \rightarrow (\checkmark, \sigma[y : \sigma(e)])}\ 38$$

$$\frac{}{(\textbf{STOP}, \sigma) \rightarrow (\perp, \sigma)}\ 39$$

Table 12: Explicit rules for atomic statements

5 Validation through an interpreter

In this section we turn our attention to the problem of validating the semantics we have defined for gPROMS. A possible way of validating gPROMS semantics is to build an *interpreter* that implements the semantics described in Section 4. This was the approach taken in [16] for a discrete subset of Stateflow. Then by comparing the runs of the gPROMS simulator and our interpreter, we can perform automated tests to determine if we have modeled the right behavior.

The first question is how to obtain an *actual* run of a gPROMS model. The output of the gPROMS simulator, contains information about evolution of the values of variables during the simulation. In particular, it is possible to observe the discrete changes in the values caused by the execution of actions, as well as the evolution of the values of variables as a function of time. Figure 1 shows the simulation results for the bouncing ball example.

Using this information it is possible to generate a trace of a given model:

$$(\sigma_0), (\sigma_1), (\sigma_2), \ldots$$

where σ_i are valuations, σ_{i+1} is generated form σ_i by performing an action or by letting time pass. In this way, we can obtain *actual* traces by means of the gPROMS simulator.

The next problem we must address, is how to build an interpreter. This can be done following the specification in terms of SOS rules, as given in Section 4. The first problem is to *ensure the correctness* of the implementation of the interpreter, with regards to the specification. Another problem is the effort that must be invested since we are dealing with a hybrid language, unlike [16] where a discrete subset is considered, which makes the construction of the interpreter a more difficult task. The construction of the interpreter can be avoided by translating gPROMS specifications intro hybrid automata, using *symbolic semantics*. We explain this idea next.

The hybrid transition system of a gPROMS model, as generated by the SOS rules (explicit rules), is in general infinite. This is due to the presence of data in the states of the transition system (and therefore in the semantic rules). However, the constraints in the initial states of a transition, as well as the changes in the values of variables, caused by continuous or discrete changes, can be represented symbolically by predicates. Using this observation, already present in symbolic model checking of hybrid systems [2] and in [17], it is possible to construct a *new set of semantic rules*, called symbolic semantics, which produce a *finite graph* representation, named *symbolic transition system* (STS). The symbolic rules are obtained from the explicit versions of these, and related by means of *soundness and completeness* results, which express that every explicit transition can be obtained from a symbolic one, and vice-versa. This idea is presented in detail in [23].

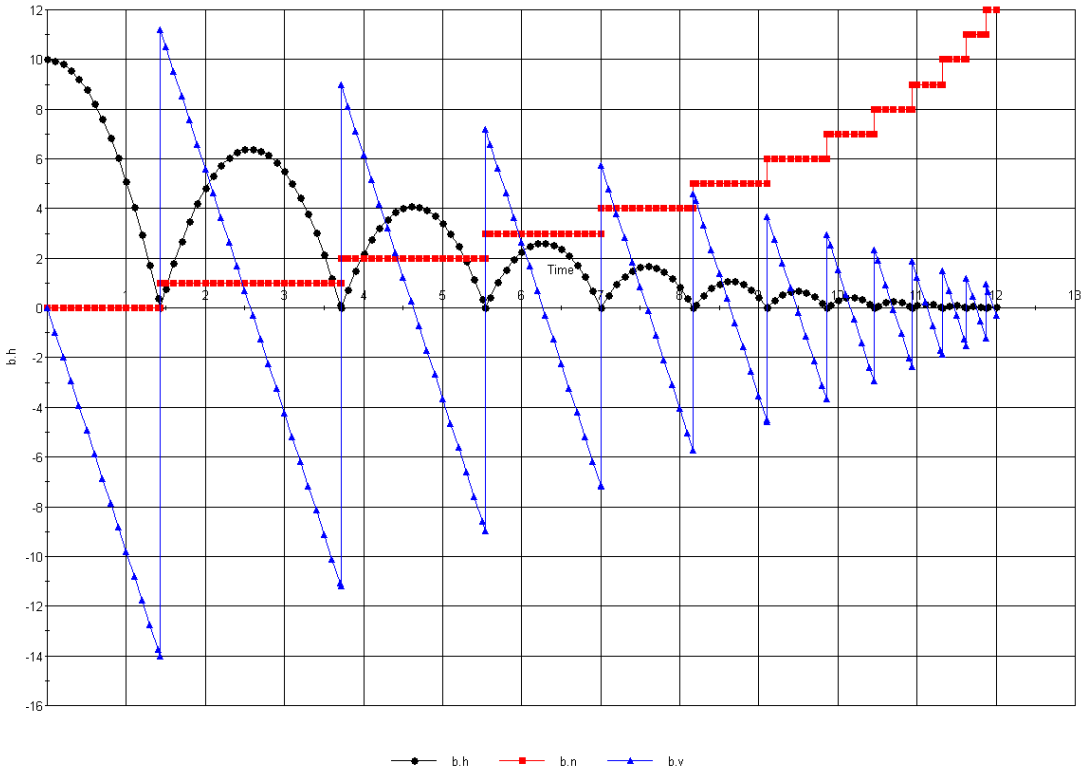

Figure 1: Bouncing ball simulation results.

A symbolic transition system *resembles* the shape of a hybrid automaton (HA) [18]. Then, it is straightforward to define a transformation from an STS to a hybrid automaton, in such a way that the gPROMS model that induced the STS and the resulting automaton are *equivalent* (state-based-bisimilar [21]). In this way, we can use *simulation* and *model checking* tools for hybrid automata to obtain a run (in the case of simulation) or a set of runs (in the case of model checking), which behaves according to the semantics we defined for gPROMS. Then it is possible to compare these runs, with the runs of the gPROMS simulator to validate the semantics.

Figure 2 illustrates the process described above, where the double headed arrow labeled (S + C) represents the soundness and completeness results existing between the two set of rules, $\underline{\leftrightarrow}$ is the bisimulation symbol, $\approx$ denotes that two runs are equal modulo decimal differences in the values of continuous variables, and $\mathscr{T}$ represents the transformation function.

The type of comparison between the generated traces depends on the tools (algorithms) we use for analyzing the behavior of the hybrid automata, which we obtain through the symbolic semantics. If we use a simulator, then we want the original trace and the trace of the simulated automaton to be *equivalent* (modulo round up errors introduced by the solvers of both tools). This requires the input model to be *deterministic*. If, on the other hand, we use a model checking tool, then we want the trace produced by the gPROMS simulator to be included in the set of reachable regions produced by the model checker.

In the next sections we present the theoretical results that are necessary for this approach. In Sec-

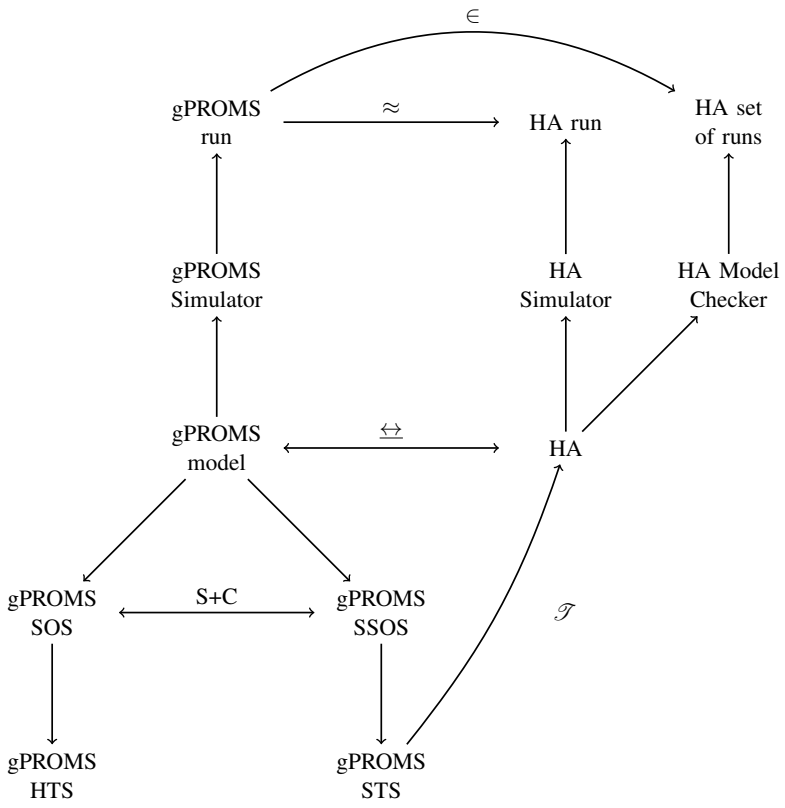

Figure 2: Iconic representation of the validation approach.

tion 5.1, we present the symbolic semantics of gPROMS. In Section 5.2, we present the CIF hybrid automata formalism, and we provide a semantic-preserving transformation (modulo our defined semantics) from gPROMS compositions to these automata.

5.1 Symbolic semantics

The states of a symbolic transition system are gPROMS compositions. The valuations are removed from it, and the state changes are represented through predicates. As in the case of the hybrid transition systems of Section 3, there are six kind of transitions:

Action transitions of the form $\langle p \rangle \xrightarrow{b,r} \langle p' \rangle$, where $b \in \mathscr{P}$ is a guard, and $r \in \mathscr{P}$ is the *reset predicate*, which constrains the possible changes in the values of variables.

Terminating transitions of the form $\langle p \rangle \xrightarrow{b,r} \langle \checkmark \rangle$, where all the components are as described before.

Halting transitions of the form $\langle p \rangle \xrightarrow{b,r} \langle \bot \rangle$.

Time transitions of the form $\langle p \rangle \xmapsto{f,g}$, where $f \in \mathscr{P}$ is a flow predicate, and $g \in \mathscr{P}$ is a time-can-progress (tcp) predicate.

No-switch transitions of the form $es \nrightarrow c$, where es is an equation or list of equations, and c is the "no-switching" condition: if c holds then the active equations in es cannot change.

(Re)initialization transitions of the form $es \overset{\text{init}}{\rightsquigarrow} c, n$, where es is a list of equations, n is a predicate that encodes the new values of the selector variables, and c is the condition that has to hold for the switch to these new values to take place. The new values of selector variables are encoded in predicates of the form:

$$x_0 = v_0 \wedge \ldots \wedge x_{n-1} = v_{n-1}$$

Table 13 presents the symbolic rules for no-switch transitions. Intuitively, these transitions extract the conditions under which no switch is possible.

$$\frac{}{\langle e_0 = e_1 \rangle \nrightarrow true} \; 40$$

$$\frac{\begin{array}{c}(\textbf{WHEN } x \ : \ es_x \ ts_x \ ) \in ws \\ b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x : \neg c \rangle, \\ \langle es_x \rangle \nrightarrow c_x \end{array}}{\langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle \nrightarrow v = x \wedge b \wedge c_x} \; 41$$

$$\frac{\langle e \rangle \nrightarrow c \qquad \langle es \rangle \nrightarrow c_s}{\langle e : es \rangle \nrightarrow c \wedge c_s} \; 42$$

Table 13: Symbolic rules for no-switch transitions

A symbolic initialization transition extracts the conditions under which a certain case branch and subbranches can be chosen, and the new values of the selector variables. Table 14 shows the symbolic rules for these kind of transitions. For proving soundness and completeness of these rules with regard to their explicit counterparts (see Section 5.1.2) we use the fact that there is an isomorphism between valuations and predicates of the form:

$$x_0 = v_0 \wedge \ldots \wedge x_{n-1} = v_{n-1}$$

such that for all i, $0 \leq i < n-1$, we have $x_i < x_{i+1}$, where $<$ is a total order among variables. Given a predicate n, of the form above, we denote as $\overline{n}$ the corresponding valuation. For instance $\overline{x = 1 \wedge y = 2} \equiv \{(x,1),(y,2)\}$, and $\overline{true} = \emptyset$.

$$\frac{}{\langle e_0 = e_1 \rangle \overset{\text{init}}{\rightsquigarrow} \text{true}, \text{true}} \; 43$$

$$\frac{\begin{array}{c}(\textbf{WHEN } x \ : \ es_x \ ts_x \ ) \in ws, \qquad (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x, \\ (\textbf{WHEN } y \ : \ es_y \ ts_y \ ) \in ws \\ \langle es_y \rangle \overset{\text{init}}{\rightsquigarrow} c_y, n \end{array}}{\langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle \overset{\text{init}}{\rightsquigarrow} c \wedge c_y, v = y \wedge n} \; 44$$

$$\frac{\langle e \rangle \overset{\text{init}}{\rightsquigarrow} c, n, \qquad \langle es \rangle \overset{\text{init}}{\rightsquigarrow} c_s, n_s}{\langle e : es \rangle \overset{\text{init}}{\rightsquigarrow} c \wedge c_s, n \wedge n_s} \; 45$$

Table 14: Symbolic rules for initialization transitions

The symbolic rules for gPROMS specifications, presented in Table 15, do not take into account the initial conditions. This is done in the soundness and completeness results, which are presented in Section 5.1.2.

$$\frac{\langle \text{gs2gsd}(G)\rangle \xrightarrow{b,r} \langle p'\rangle}{\langle G\rangle \xrightarrow{b,r} \langle p'\rangle}\ 46 \qquad \frac{\langle \text{gs2gsd}(G)\rangle \overset{f,g}{\longmapsto}}{\langle G\rangle \overset{f,g}{\longmapsto}}\ 47$$

Table 15: Symbolic rules for specifications

Table 16 presents the symbolic rules for gPROMS system descriptions. In Rules 49 and 50, the no-switching condition is added to the guard of the action so that it is only enabled when the equation part cannot perform a switch.

$$\frac{\langle es\rangle \xrightarrow{b,r} \langle es'\rangle}{\langle\langle es,S\rangle\rangle \xrightarrow{b,r} \langle\langle es',S\rangle\rangle}\ 48 \qquad \frac{es \nrightarrow c, \qquad \langle S\rangle \xrightarrow{b,r} \langle S'\rangle}{\langle\langle es,S\rangle\rangle \xrightarrow{c\wedge b,r} \langle\langle es,S'\rangle\rangle}\ 49$$

$$\frac{es \nrightarrow c, \qquad \langle S\rangle \xrightarrow{b,r} \langle \bot\rangle}{\langle\langle es,S\rangle\rangle \xrightarrow{c\wedge b,r} \langle \bot\rangle}\ 50 \qquad \frac{\langle es\rangle \overset{f_e,g_e}{\longmapsto}, \quad \langle S\rangle \overset{f_s,g_s}{\longmapsto}}{\langle\langle es,S\rangle\rangle \overset{f_e\wedge f_s,g_e\wedge g_s}{\longmapsto}}\ 51$$

Table 16: Symbolic rules for gPROMS system descriptions

Table 17 presents the symbolic rules for gPROMS equations. Rule 52 has no switching condition, which expresses that an equation can delay for as long as it holds. In Rule 53 the conditions for a switch to occur are that the selector variable has the value of the corresponding branch, and that the switch condition holds. The update condition is $v = y \wedge n$, and represents symbolically the change in the values of the selector variables after the switch takes place. Rule 54 is similar to the previous one, except that no update is made to the selector variable of the enclosing `CASE` statement. The symbolic time rule adds $v = x$ as a required flow condition. Note that several outgoing time transitions are generated by this rule: one per branch of the case statement.

Table 18 presents the symbolic rules for lists of equations. They are similar to their explicit counterparts, and therefore no further explanation is given.

The symbolic rules for sequential composition are presented in Table 19. Since the explicit rules for sequential composition pose no semantic constraints on the valuations, the symbolic information is transferred unaltered from the premises to the conclusion.

The symbolic rules for parallel composition are exhibited in Table 20. As with sequential composition, the symbolic rules for this language construct do not alter the information on the arrows, except for the time rule. In this case, a symbolic time transition of a parallel composition has as flow (switching, respectively) condition the conjunction of the flow (switching) conditions of its components. It is of decisive importance for the proof of completeness that, in the time rule, the flow and switching conditions of the conclusion are the conjunction of the corresponding conditions of the individual components. If we require these conditions to be the same then completeness does not hold.

The symbolic rules for control structures are shown in Table 21. The guards of these language constructs determine the guards and switching conditions of the corresponding symbolic transitions.

$$\frac{}{\langle e_0 = e_1 \rangle \overset{e_0=e_1,\text{true}}{\longmapsto}} 52$$

$$\frac{\begin{array}{c}(\textbf{WHEN } x \ : \ es_x \ ts_x \ ) \in ws, \qquad (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x \\ (\textbf{WHEN } y \ : \ es_y \ ts_y \ ) \in ws, \qquad \langle es_y \rangle \overset{\text{init}}{\rightsquigarrow} c_y, n\end{array}}{\begin{array}{c}\langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle \xrightarrow{v=x \wedge c \wedge c_y, v=y \wedge n} \\ \langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle\end{array}} 53$$

$$\frac{\begin{array}{c}(\textbf{WHEN } x \ : \ es \ ts \ ) \in ws, \\ b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts : \neg c \rangle, \\ \langle es \rangle \xrightarrow{b_e, r_e} \langle es \rangle\end{array}}{\langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle \xrightarrow{v=x \wedge b \wedge b_e, r_e} \langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle} 54$$

$$\frac{\begin{array}{c}(\textbf{WHEN } x \ : \ es_x \ ts_x \ ) \in ws, \\ b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x : \text{closeNeg}(\neg c) \rangle, \\ \langle es_x \rangle \overset{f_e, g_e}{\longmapsto}\end{array}}{\langle \textbf{CASE } v \textbf{ OF } ws \textbf{ END} \rangle \overset{v=x \wedge f_e, v=x \wedge b \wedge g_e}{\longmapsto}} 55$$

Table 17: Symbolic rules for equations

$$\frac{}{\langle [\,] \rangle \overset{\text{true,true}}{\longmapsto}} 56$$

$$\frac{\langle e \rangle \xrightarrow{b,r} \langle e' \rangle}{\langle e : es \rangle \xrightarrow{b,r} \langle e' : es \rangle} 57 \qquad \frac{\langle es \rangle \xrightarrow{b,r} \langle es' \rangle}{\langle e : es \rangle \xrightarrow{b,r} \langle e : es' \rangle} 58$$

$$\frac{\langle e \rangle \overset{g_e, g_e}{\longmapsto}, \qquad \langle es \rangle \overset{f_s, g_s}{\longmapsto}}{\langle e : es \rangle \xrightarrow{f_e \wedge f_s, g_e \wedge g_s}} 59$$

Table 18: Symbolic rules for lists of equations

In Table 22, the symbolic rules for atomic statements are displayed. Here, given a gPROMS expression e, $\textbf{old}(e)$ is the expression obtained after replacing all free variables x appearing in e by $\textbf{old}(x)$.

5.1.1 Example

We present the symbolic transition systems induced by the parts of the bouncing ball model. In Listing 5 we consider a part of the schedule of the bouncing ball specification.

Listing 5: Part of the schedule of the bouncing ball specification.

```
1 WHILE TRUE DO
2     SEQUENCE
3         CONTINUE UNTIL b.h<=0.0
```

$$\frac{\langle s\rangle \xrightarrow{b,r} \langle s'\rangle}{\langle \textbf{SEQUENCE } s : zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \textbf{SEQUENCE } s' : zs \textbf{ END}\rangle} 60$$

$$\frac{\langle s\rangle \xrightarrow{b,r} \langle \checkmark\rangle, \qquad 2 \leq \#zs}{\langle \textbf{SEQUENCE } s : zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \textbf{SEQUENCE } zs \textbf{ END}\rangle} 61$$

$$\frac{\langle s\rangle \xrightarrow{b,r} \langle \checkmark\rangle, \qquad \#zs = 1}{\langle \textbf{SEQUENCE } s : zs \textbf{ END}\rangle \xrightarrow{b,r} \langle zs.0\rangle} 62$$

$$\frac{\langle s\rangle \xrightarrow{b,r} \langle \bot\rangle}{\langle \textbf{SEQUENCE } s : zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \bot\rangle} 63$$

$$\frac{\langle s\rangle \overset{f,g}{\longmapsto}}{\langle \textbf{SEQUENCE } s : zs \textbf{ END}\rangle \overset{f,g}{\longmapsto}} 64$$

Table 19: Symbolic rules for sequential composition

$$\frac{\langle zs.i\rangle \xrightarrow{b,r} \langle s'\rangle, \qquad 0 \leq i < \#zs}{\langle \textbf{PARALLEL } zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \textbf{PARALLEL } zs[i : s'] \textbf{ END}\rangle} 65$$

$$\frac{\langle zs.i\rangle \xrightarrow{b,r} \langle \checkmark\rangle, \qquad 0 \leq i < \#zs, \qquad 3 \leq \#zs}{\langle \textbf{PARALLEL } zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \textbf{PARALLEL } zs \setminus i \textbf{ END}\rangle} 66$$

$$\frac{\langle zs.i\rangle \xrightarrow{b,r} \langle \checkmark\rangle, \qquad 0 \leq i < \#zs, \qquad \#zs = 2}{\langle \textbf{PARALLEL } zs \textbf{ END}\rangle \xrightarrow{b,r} \langle (zs \setminus i).0\rangle} 67$$

$$\frac{\langle zs.i\rangle \xrightarrow{b,r} \langle \bot\rangle, \qquad 0 \leq i < \#zs}{\langle \textbf{PARALLEL } zs \textbf{ END}\rangle \xrightarrow{b,r} \langle \bot\rangle} 68$$

$$\frac{\langle s\rangle \overset{f_s,g_s}{\longmapsto}, \qquad \langle t\rangle \overset{f_t,g_t}{\longmapsto}}{\langle \textbf{PARALLEL } [s,t] \textbf{ END}\rangle \overset{f_s \wedge f_t, g_s \wedge g_t}{\longmapsto}} 69$$

$$\frac{2 \leq \#zs, \qquad \langle s\rangle \overset{f_s,g_s}{\longmapsto} \quad \langle \textbf{PARALLEL } zs \textbf{ END}\rangle \overset{f_z,g_z}{\longmapsto}}{\langle \textbf{PARALLEL } s : zs \textbf{ END}\rangle \overset{f_s \wedge f_z, g_s \wedge g_z}{\longmapsto}} 70$$

Table 20: Symbolic rules for parallel composition

```
4           REINITIAL b.v WITH b.v=(-(b.e)*OLD(b.v)); END
5           REINITIAL b.h WITH b.h=0.0; END
6           RESET b.n:=(OLD(b.n)+1.0); END
7       END
8   END
```

$$\frac{}{\langle \textbf{WHILE } b \textbf{ DO } s \textbf{ END}\rangle \xrightarrow{b,\text{true}} \langle \textbf{SEQUENCE } s : [\textbf{WHILE } b \textbf{ DO } s \textbf{ END}] \textbf{ END}\rangle}\ 71$$

$$\frac{}{\langle \textbf{WHILE } b \textbf{ DO } s \textbf{ END}\rangle \xrightarrow{\neg b,\text{true}} \langle \checkmark \rangle}\ 72$$

$$\frac{}{\langle \textbf{CONTINUE UNTIL } b \ ;\rangle \xrightarrow{b,\text{true}} \langle \checkmark \rangle}\ 73$$

$$\frac{}{\langle \textbf{CONTINUE UNTIL } b \ ;\rangle \overset{\text{true},\text{closeNeg}(\neg b)}{\longmapsto}}\ 74$$

Table 21: Symbolic rules for control structures

$$\frac{}{\langle \textbf{RESET } x := e \textbf{ END}\rangle \xrightarrow{\text{true},x=\textbf{old}(e)} \langle \checkmark \rangle}\ 75$$

$$\frac{}{\langle \textbf{SWITCH } x :=y \textbf{ END}\rangle \xrightarrow{\text{true},x=y} \langle \checkmark \rangle}\ 76$$

$$\frac{}{\langle \textbf{REINITIAL } y \textbf{ WITH } y = e \textbf{ END}\rangle \xrightarrow{\text{true},y=\textbf{old}(e)} \langle \checkmark \rangle}\ 77$$

$$\frac{}{\langle \textbf{STOP}\rangle \xrightarrow{\text{true},\text{true}} \langle \bot \rangle}\ 78$$

Table 22: Symbolic rule for atomic statements

Figure 3 shows the symbolic transition system induced by the gPROMS composition of Listing 5. We use the letter W to refer to the gPROMS composition that spans lines 1 to 8. The numbers in the states refer to the composition in these line numbers. We write **SEQ** p as an abbreviation of **SEQUENCE** p **END**. The time transition is depicted as an outgoing arrow without target location. In the symbolic transition system of Figure 3 only one state has a time transition (which corresponds to the `CONTINUE UNTIL` statement).

Next we present the STS for the equations part of the bouncing ball. We use e to denote the gPROMS list of equations shown in Listing 4. The resulting symbolic transition system is presented in Figure 4, where, for the sake of succinctness, variable `b.ballPosition` is denoted as $b.p$.

Combining these two transition systems according to the rules of gPROMS system descriptions we get the STS shown in Figure 5. Due to space constrains, we use the following conventions:

- The self loops labeled a represent two action transitions of the form:

$$\langle p\rangle \xrightarrow{b.p=b.air\wedge b.h\leq 0\wedge b.v\leq 0,b.p=b.ground} \langle p\rangle$$
$$\langle p\rangle \xrightarrow{b.p=b.ground\wedge\neg(b.h\leq 0\wedge b.v\leq 0),b.p=b.air} \langle p\rangle$$

  where p is the source and target state of the loop.

- The edge labeled a_0 represents the action transitions:

$$\langle W\rangle \xrightarrow{b.p=b.air\wedge\neg(b.h\leq 0\wedge b.v\leq 0),\text{true}} \langle \textbf{SEQ } [\textbf{SEQ } [3,4,5,6],W]\rangle$$
$$\langle W\rangle \xrightarrow{b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0,\text{true}} \langle \textbf{SEQ } [\textbf{SEQ } [3,4,5,6],W]\rangle$$

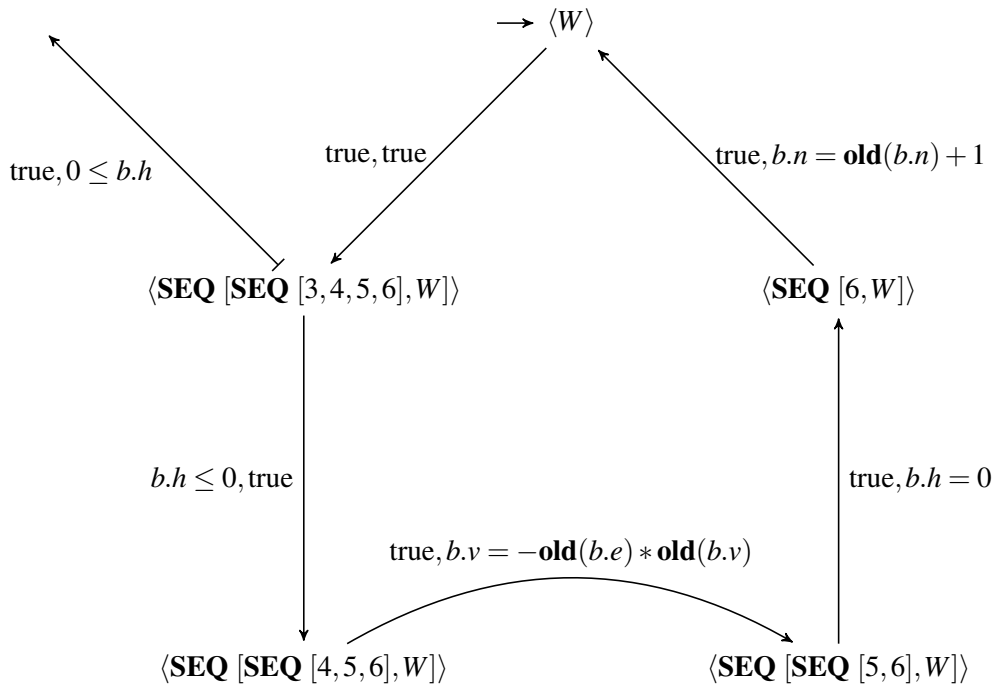

Figure 3: Symbolic transition system induced by while loop.

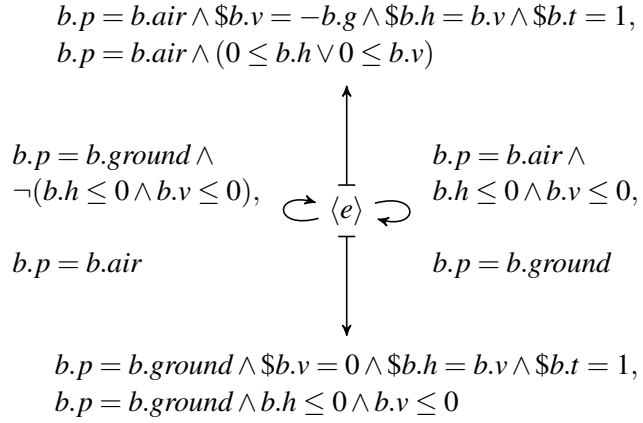

Figure 4: Symbolic transition system induced by the equations of the bouncing ball specification.

- The edge labeled a_1 represents the action transitions:

$$\langle \textbf{SEQ}\ [\textbf{SEQ}\ [3,4,5,6],W]\rangle \xrightarrow{b.p=b.air\wedge\neg(b.h\leq 0\wedge b.v\leq 0)\wedge b.h\leq 0,\text{true}} \langle \textbf{SEQ}\ [\textbf{SEQ}\ [4,5,6],W]\rangle$$

$$\langle \textbf{SEQ}\ [\textbf{SEQ}\ [3,4,5,6],W]\rangle \xrightarrow{b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0\wedge b.h\leq 0,\text{true}} \langle \textbf{SEQ}\ [\textbf{SEQ}\ [4,5,6],W]\rangle$$

- The edge labeled a_2 represents the action transitions:

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [4,5,6],W]\rangle \xrightarrow{b.p=b.air\wedge\neg(b.h\leq 0\wedge b.v\leq 0),b.v=-\mathbf{old}(b.e)*\mathbf{old}(b.v)} \langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [5,6],W]\rangle$$

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [4,5,6],W]\rangle \xrightarrow{b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0,b.v=-\mathbf{old}(b.e)*\mathbf{old}(b.v)} \langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [5,6],W]\rangle$$

- The edge labeled a_3 represents the action transitions:

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [5,6],W]\rangle \xrightarrow{b.p=b.air\wedge\neg(b.h\leq 0\wedge b.v\leq 0),b.h=0} \langle \mathbf{SEQ}\ [6,W]\rangle$$

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [5,6],W]\rangle \xrightarrow{b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0,b.h=0} \langle \mathbf{SEQ}\ [6,W]\rangle$$

- The edge labeled a_4 represents the action transitions:

$$\langle \mathbf{SEQ}\ [6,W]\rangle \xrightarrow{b.p=b.air\wedge\neg(b.h\leq 0\wedge b.v\leq 0),b.n=\mathbf{old}(b.n)+1} \langle W\rangle$$

$$\langle \mathbf{SEQ}\ [6,W]\rangle \xrightarrow{b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0,b.n=\mathbf{old}(b.n)+1} \langle W\rangle$$

- The edge labeled f represents the time transitions:

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [3,4,5,6],W]\rangle \xrightarrow{b.p=b.air\wedge\$b.v=-b.g\wedge\$b.h=b.v\wedge\$b.t=1,b.p=b.air\wedge 0\leq b.h\wedge 0\leq b.v\wedge 0\leq b.h}$$

$$\langle \mathbf{SEQ}\ [\mathbf{SEQ}\ [3,4,5,6],W]\rangle \xrightarrow{b.p=b.ground\wedge\$b.v=0\wedge\$b.h=b.v\wedge\$b.t=1,b.p=b.ground\wedge b.h\leq 0\wedge b.v\leq 0\wedge 0\leq b.h}$$

5.1.2 Soundness and completeness results

The explicit and symbolic rules are related by the following soundness and completeness properties. These properties state how an explicit transition system can be reconstructed from its symbolic version, and vice-versa.

Property 1 (Soundness and completeness of no-switch transitions)**.** *For all es, σ:*

1. *for all c, if $\langle es\rangle \nrightarrow c$ and $\sigma \models c$ then $(es,\sigma)\nrightarrow$.*
2. *if $(es,\sigma)\nrightarrow$ then there is a c such that $\langle es\rangle \nrightarrow c$, and $\sigma \models c$.*

Proof. Straightforward using structural induction. □

Property 2 (Soundness and completeness of initialization transitions)**.** *For all es and σ:*

1. *for all c, n, if $\langle es\rangle \overset{\text{init}}{\rightsquigarrow} c,n$, and $\sigma \models c$, then $(es,\sigma)\overset{\text{init}}{\rightsquigarrow}\overline{n}$.*
2. *for all ℓs, if $(es,\sigma)\overset{\text{init}}{\rightsquigarrow}\ell s$, then there are c and n such that $\langle es\rangle \overset{\text{init}}{\rightsquigarrow} c,n$, and $\sigma \models c\wedge \ell s=\overline{n}$.*

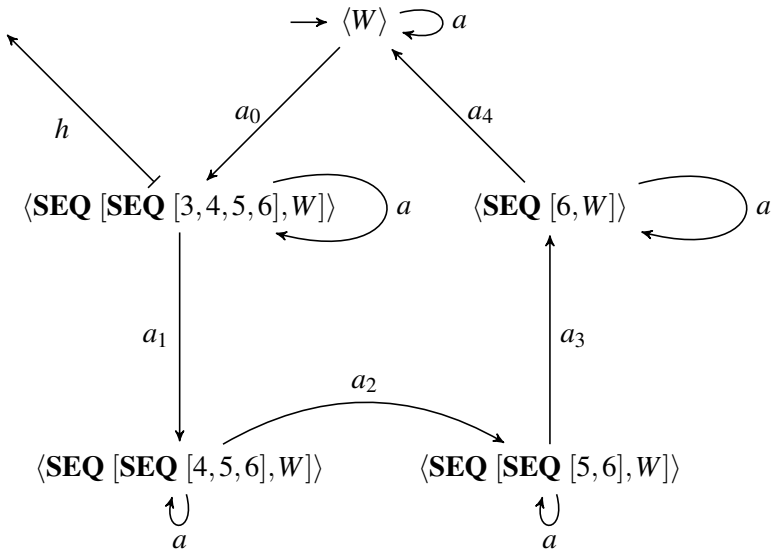

Figure 5: Symbolic transition system for a simplified version of the bouncing ball model.

Proof. Straightforward using structural induction, and the fact that all selector variables are unique. □

Property 3 (Soundness and completeness time and action)**.** *For all symbolic states p, p' and for any valuation σ, we have that the following properties hold:*

1. *for all σ' if $(p,\sigma) \to (p',\sigma')$ then there are b, and r such that $\langle p\rangle \xrightarrow{b,r} \langle p'\rangle$, $\sigma \models b$, $\mathbf{old}(\sigma)\cup\sigma' \models r$, $\sigma \overset{\overline{\mathrm{NFV}(r)}}{=} \sigma'$, and $\sigma \models \mathrm{iconds}(p)$, where:*
   (a) *given a predicate e, $\mathrm{NFV}(e)$ returns the set of free variables in expression e such that they are not of the form $\mathbf{old}(x)$.*
   (b) *given a set X, $\overline{X}$ denotes the complement of X.*
   (c) *given a valuation σ, we define $\mathbf{old}(\sigma) = \{(\mathbf{old}(x),v) \mid (x,v)\in\sigma\}$.*
   (d) *given two valuations σ, σ', and a set of variables X, we define:*

$$(\sigma \overset{X}{=} \sigma') \triangleq \langle \forall x : x \in X : \sigma(x) = \sigma(x)\rangle$$

2. *for all b, r, and σ', if $\langle p\rangle \xrightarrow{b,r} \langle p'\rangle$, $\sigma \models b$, $\mathbf{old}(\sigma)\cup\sigma' \models r$, $\sigma \overset{\overline{\mathrm{NFV}(r)}}{=} \sigma'$, and $\sigma \models \mathrm{iconds}(p)$ then $(p,\sigma) \to (p',\sigma')$.*
3. *for all ρ and σ', if $(p,\sigma) \overset{\rho}{\longmapsto} (\mathrm{gs2gsd}(p),\sigma')$ then there are f, g, and t such that $\langle p\rangle \overset{f,g}{\longmapsto}$, $0<t$, $\mathrm{dom}(\rho)=[0,t]$, $\sigma = \rho(0)\restriction_{\mathscr{V}}$, $\sigma' = \rho(t)\restriction_{\mathscr{V}}$, $\sigma \models \mathrm{iconds}(p)$, $\rho \models_{\mathrm{flow}} f$, $\rho \models g$.*
4. *for all f, g, ρ, and σ', if $\langle p\rangle \overset{f,g}{\longmapsto}$, $0<t$, $\mathrm{dom}(\rho)=[0,t]$, $\sigma \models \mathrm{iconds}(p)$, $\rho \models_{\mathrm{flow}} f$, $\rho \models g$, $\sigma = \rho(0)\restriction_{\mathscr{V}}$, $\sigma' = \rho(t)\restriction_{\mathscr{V}}$, then $(p,\sigma) \overset{\rho}{\longmapsto} (\mathrm{gs2gsd}(p),\sigma')$.*

In the previous property, note that we need to write $\mathrm{gs2gsd}(p)$ here since there is only one case in which the time transition is not a self loop (when $p \in \mathscr{G}_{\mathrm{specs}}$).

Proof. The proof is relatively simple given the similarity that exists between the two set of rules. The proof goes by structural induction on the gPROMS composition terms. Some considerations:

gPROMS Specifications case: To prove this case we use the hypothesis that $\sigma \models \text{iconds}(G)$.

System descriptions case: Here we need to use the soundness and completeness results for no-switch transitions. In this case it is also important that the iconds function is defined such that:

$$\text{iconds}(\langle es, S\rangle) = \text{iconds}(es) \wedge \text{iconds}(S)$$

which is the case in our current setting, since $\text{iconds}(p) \triangleq$ true for all gPROMS compositions p such that p is not a gPROMS specification.

Equations case: For proving soundness and completeness for `CASE` statements we use the soundness and completeness of initialization transitions (Prop. 2), and the fact that all selector variables are different. Also condition

$$\sigma \overset{\overline{\text{NFV}(r)}}{=} \sigma'$$

is decisive for the proof. For time transitions we use the fact that discrete variables, in particular selector variables, do not change over time delays.

Atomic statements: For proving this case we use the fact that

$$\mathbf{old}(\sigma) \cup \sigma[x : \sigma(x)] \models x = \mathbf{old}(e)$$

and

$$\sigma \overset{\overline{\text{NFV}(r)}}{=} \sigma'$$

□

5.2 Transforming transition systems to CIF hybrid automata

In this section we describe the syntax and semantics of the hybrid automata that can be simulated by CIF. We start by defining hybrid CIF automata, based on the characterization of [4] [2].

Definition 7 (CIF hybrid automaton)**.** *A hybrid CIF automaton H is a tuple*

$$(V, \text{init}, \text{inv}, \text{tcp}, E)$$

where:

- $V \subseteq \mathscr{L}$ *is a set of locations, $\mathscr{L}$ is the set of all locations.*
- $\text{init} \in (V \to \mathscr{P})$ *is the initial predicate function, which associates initial conditions to each location.*
- $\text{inv} \in (V \to \mathscr{P})$ *is the invariant function.*
- $\text{tcp} \in (V \to \mathscr{P})$ *is the time-can-progress function. A time can progress condition states the conditions under which time can pass.*
- $E \subseteq (V \times \mathscr{A}_\tau \times \mathscr{P} \times V)$ *is a set of edges, where $\mathscr{A}$ is the set of all actions, $\tau \notin \mathscr{A}$ is the silent action, and $\mathscr{A}_\tau \triangleq \mathscr{A} \cup \{\tau\}$.*

[2] The correspondence between this simplified version, and the original version should be obvious.

The semantics of a CIF hybrid automata is defined in terms of *hybrid transition systems*, that is, every automaton induces a *hybrid transition system* (HTS), which constitutes its semantic domain. The transition system space of CIF hybrid automata is generated according to the rules of Table 23, where we use the following conventions:

$$\alpha \triangleq (V, \text{init}, \text{inv}, \text{tcp}, E)$$
$$\alpha[v] \triangleq (V, \text{id}_v, \text{inv}, \text{tcp}, E)$$

and id_v denotes the function that returns true only if the location equals v. More specifically, $\text{id}_v \in Loc \to \mathbb{B}$, and $\text{id}_v(w) \triangleq\ v \equiv w$, where $\equiv$ denotes syntactic equivalence. Note that in the CIF hybrid transition systems, the dotted variables are part of the valuations.

$$\frac{\sigma \models \text{init}(v), \qquad (v,a,r,v') \in E, \qquad \textbf{old}(\sigma) \cup \sigma' \models r, \qquad \sigma \restriction_{\mathscr{V}} \overset{\overline{\text{NFV}(r)}}{=} \sigma' \restriction_{\mathscr{V}}, \qquad \sigma \models \text{inv}(v), \qquad \sigma' \models \text{inv}(v')}{(\alpha, \sigma) \xrightarrow{a} (\alpha[v'], \sigma')} \ 79$$

$$\frac{\sigma \models \text{init}(v), \qquad \text{dom}(\rho) = [0,t], \qquad 0 < t, \qquad \rho \models_{\text{flow}} \text{inv}(v), \qquad \rho \restriction_{[0,t)} \models \text{tcp}(v), \qquad \sigma = \rho(0)}{(\alpha, \sigma) \xrightarrow{t} (\alpha[v], \rho(t))} \ 80$$

Table 23: Semantic rules of CIF hybrid automata

Given a gPROMS composition, and the symbolic transition system space of gPROMS transitions (which is obtained from the symbolic SOS rules), we can construct a CIF *hybrid automaton*. The idea is to translate every symbolic action transition into an edge, and to use the symbolic time transitions to create invariants and tcp functions.

There are two main difficulties in this transformation. First, differential equations are expressed in CIF in the form of invariants. Thus we must use these to transform the flow conditions of the gPROMS STS's. The problem is that the invariants are checked every time a new location is entered, though this is not the case for gPROMS equations. Second, time-can-progress conditions in CIF consider right open intervals, whereas switching conditions in gPROMS must hold in a closed interval.

To obtain a CIF automaton from a gPROMS composition such that they have the same behavior, we require that all flow conditions in the induced STS of the composition are *solvable*.

Definition 8 (Solvable predicate)**.** *A predicate g is solvable if for all valuations $\sigma \in \Sigma$, there is a valuation $\$\sigma \in \{\$x \mid x \in \mathscr{V}\} \to \Lambda$ such that:*

$$\sigma \cup \$\sigma \models g$$

Most of the systems of differential equations found in practice satisfy this requirement.

To overcome the second problem (the difference between open and closed intervals), we require that all the switching conditions in the gPROMS composition are continuous-closed (Definition 5). This enables us to use the following property.

Property 4. *Let e be a **continuous-closed** predicate, and $\rho \in [0,t] \to \$\Sigma$, $0 < t$, then the following holds:*

$$(\rho \restriction_{[0,t)} \models_{\text{flow}} e) \Rightarrow (\rho(t) \models_{\text{flow}} e)$$

Note that the closed negation of continuous-closed switching conditions (Definition 5) generate continuous closed predicates when all the variables involved are continuous or discrete.

The following definition describes how to obtain a CIF hybrid automaton out of a gPROMS symbolic transition system.

Definition 9 (Hybrid automaton induced by a gPROMS composition)**.** *Given a gPROMS composition p, the CIF automaton induced by p is a tuple*

$$(V, \text{init}, \text{inv}, \text{tcp}, E)$$

where:

- *The initial predicate function is given by* [3]*:*

$$\text{init}(q) = \begin{cases} \text{iconds}(p) & \text{if } q \equiv p \\ \text{false} & \text{otherwise} \end{cases}$$

- *The set of locations is defined as the least set that satisfies:*
  1. $\text{gs2gsd}(p), \bot, \checkmark \in V$ *where* $\checkmark$*, and* $\bot$ *are special symbols that do not belong to* $\mathscr{G}$*.*
  2. *If* $q \in V$ *and* $\langle q \rangle \xrightarrow{b,r} \langle x \rangle$ *then* $x \in V$*, where* $x \in \mathscr{G} \cup \{\checkmark, \bot\}$*.*
- *The set of edges is defined as the least set that satisfies:*
  - *If* $q \in V$ *and* $\langle q \rangle \xrightarrow{b,r} \langle x \rangle$ *then* $(q, \tau, \mathbf{old}(b) \wedge r, x) \in E$*, where* $x \in V$*.*
- *The invariant function is defined as:*

$$\text{inv}(q) = \begin{cases} \langle \bigvee f_i, g_i : \langle q \rangle \xmapsto{f_i, g_i} : f_i \rangle & \text{if } q \text{ has at least a time transition} \\ true & \text{otherwise} \end{cases}$$

  where $\text{dom}(\text{inv}) = V$*.*
- *The tcp function is defined as:*

$$\text{tcp} = \{(q, g) \mid q \in Loc \wedge g \equiv \langle \bigvee f_i, g_i : \langle q \rangle \xmapsto{f_i, g_i} : g_i \rangle\}$$

Let α *be the automaton induced by p, then we denote this as*

$$\alpha = \mathscr{T}(p)$$

The set of all automata obtained from gPROMS compositions, using this definition, is denoted as $\mathscr{H}_g$*.*

Note that if a location has no time transitions, then the resulting invariant will be true, and the time can progress condition false, which means that no time can pass.

To be able to compute a CIF hybrid automaton from a gPROMS composition, it is necessary that the resulting transition system is finite.

Property 5 (Finiteness of the induced STS)**.** *The symbolic transition system induced by a gPROMS composition is finite.*

[3] Note that for a composition other than a gPROMS specification, $\text{iconds}(p) \equiv \text{true}$.

Proof. The proof goes via structural induction on the gPROMS process terms.

For the atomic gPROMS compositions (equalities, assignments, continue, and stop statements) the property holds trivially.

The inductive case can be proven easily. The only non-trivial case is the `WHILE` statements. Assume:

$$p \equiv \textbf{WHILE } b \textbf{ DO } s \textbf{ END}$$

By induction hypothesis, we know that the symbolic transition system induced by s is finite. Then, the only reachable states from p, are those of the form:

- $\langle \textbf{SEQUENCE } s : [p] \textbf{ END} \rangle$.
- $\langle \textbf{SEQUENCE } s' : [p] \textbf{ END} \rangle$, where s' is reachable from s.
- $\langle p \rangle$ itself.

Since the number of reachable states from s is finite, the number of states reachable from p is also finite. And since s is finitely branching, and the symbolic rules for while statements only add two edges, the symbolic transition system induced by p is finitely branching, and therefore finite. This concludes the proof for this case.

It is interesting to observe that the above argument does not work if the rule for sequential composition allows transitions of the form:

$$\langle \textbf{SEQUENCE } p : [q] \textbf{ END} \rangle \xrightarrow{b,r} \langle \textbf{SEQUENCE } [q] \textbf{ END} \rangle$$

since from p we would reach the state **SEQUENCE** $[p]$ **END**. In fact, if such transitions were allowed, the induced symbolic transition system would be infinite. These kind of transitions are prevented by our set of rules. □

The following corollary follows from the previous property.

Corollary 1 (Finiteness of the induced HA)**.** *The hybrid automaton induced by a gPROMS composition is finite.*

Note that even though the resulting transition systems are finite, their size can grow exponentially with the number of nested case statements, and parallel components. For the scope of our current work, validation of the SOS rules, this is not a serious problem. The size of the transition system could be reduced using *linearization* techniques [1].

It is our goal to show that for every gPROMS specification there is a hybrid automaton such that they are equivalent. It does not seem possible to prove stateless bisimulation, since invariants in CIF are checked at the beginning of every action transition, whereas in gPROMS there is no corresponding concept. Thus the notion of equivalence we use is state-based bisimulation. To this end, we define the notion of state-based bisimilarity between gPROMS compositions and CIF hybrid automata.

Definition 10 (gPROMS $\underline{\leftrightarrow}$ CIF HA)**.** *A relation $R \subseteq (\mathscr{G} \times \Sigma) \times (\mathscr{H}_g \times \$\Sigma)$ is a* state-based bisimulation *relation if and only if for all p, α, σ_0, σ_1 such that $((p,\sigma_0),(\alpha,\sigma_1)) \in R$ we have $\sigma_0 \restriction_{\mathscr{V}} = \sigma_1$ and:*

1. $\langle \forall p', \sigma_0' :: (p,\sigma_0) \to (p',\sigma_0') \Rightarrow$
   $\langle \exists \alpha', \sigma_1' :: (\alpha,\sigma_1) \xrightarrow{\tau} (\alpha', \sigma_0' \cup \sigma_1') \wedge ((p',\sigma_0'),(\alpha',\sigma_0' \cup \sigma_1')) \in R \rangle \rangle$
2. $\langle \forall \alpha', \sigma_1' :: (\alpha,\sigma_1) \xrightarrow{\tau} (\alpha',\sigma_1') \Rightarrow$
   $\langle \exists p', :: (p,\sigma_0) \to (p', \sigma_1' \restriction_{\mathscr{V}}) \wedge ((p', \sigma_1' \restriction_{\mathscr{V}}),(\alpha',\sigma_1')) \in R \rangle \rangle$

3. $\langle \forall p', \sigma', \rho :: (p, \sigma_0) \xmapsto{\rho} (p', \sigma_0') \Rightarrow$
   $\langle \exists \alpha', \sigma_1' :: (\alpha, \sigma_1) \xmapsto{\rho} (\alpha', \sigma_0' \cup \sigma_1') \wedge ((p', \sigma_0'), (\alpha', \sigma_0' \cup \sigma_1')) \in R \rangle \rangle$
4. $\langle \forall \alpha', \sigma', t :: (\alpha, \sigma_1) \xmapsto{t} (\alpha', \sigma_1') \Rightarrow$
   $\langle \exists p' :: (p, \sigma_0) \xmapsto{\rho} (p', \sigma_1' \restriction_{\mathscr{V}}) \wedge ((p', \sigma_1' \restriction_{\mathscr{V}}), (\alpha', \sigma_1')) \in R \rangle \rangle$

A gPROMS composition p and a CIF hybrid automaton α are state-based bisimilar*, denoted as $p \leftrightarrows \alpha$ if and only if there exists a state-based bisimulation relation R such that $(p, \alpha) \in R$.*

Using the notion of state-based bisimilarity, we state the main theorem of this section.

Theorem 1 (Relation between gPROMS and hybrid automata)**.** *For each gPROMS composition, $p \in \mathscr{G}$, its induced hybrid automaton is state-based bisimilar. This is:*

$$p \leftrightarrows \mathscr{T}(p)$$

Proof. Henceforth we assume:

$$\mathscr{T}(p) = (V, \text{init}, \text{inv}, \text{tcp}, E)$$

Then, it can be proven that relation R defined as:

$$\begin{aligned} R \triangleq & \{((q, \sigma \restriction_{\mathscr{V}}), (\mathscr{T}(p)[q], \sigma)) \mid q \in Loc \wedge q \notin \mathscr{G}_{\text{specs}} \wedge \sigma \models Inv(q)\} \cup \\ & \{((p, \sigma \restriction_{\mathscr{V}}), (\mathscr{T}(p), \sigma)) \mid \sigma \models Inv(\text{gs2gsd}(p))\} \end{aligned}$$

is a state-based bisimulation relation, where $\mathscr{G}_{\text{specs}}$ is the set of all gPROMS specifications.

□

6 Examples

This section shows several examples, which are used to evaluate the semantic. For this purpose gPROMS specifications are transformed into hybrid automata (HA), as described in Section 5. The HA are simulated with the CIF simulator [30]. The simulation results are employed to compare the behavior of the gPROMS specifications with their corresponding automaton models.

6.1 Bouncing ball

The bouncing ball specification, presented in Section 2, was transformed into a CIF model, using the procedure described in Section 5.2. The simulations results are shown in Figure 6, where the similarities with the simulation results of the gPROMS simulator (Figure 1) are evident. The bounces occur at approximately the same time (1.42, 3.70, 5.54, etc.), where variable `b.n` increases by one after each bounce, and the velocity is multiplied by the additive inverse of the damping factor. The differences observed in the plots are due to different methods used for solving the differential equations, numerics, and event detection. The differences are also in the same numerical range as the absolute tolerance of the simulations.

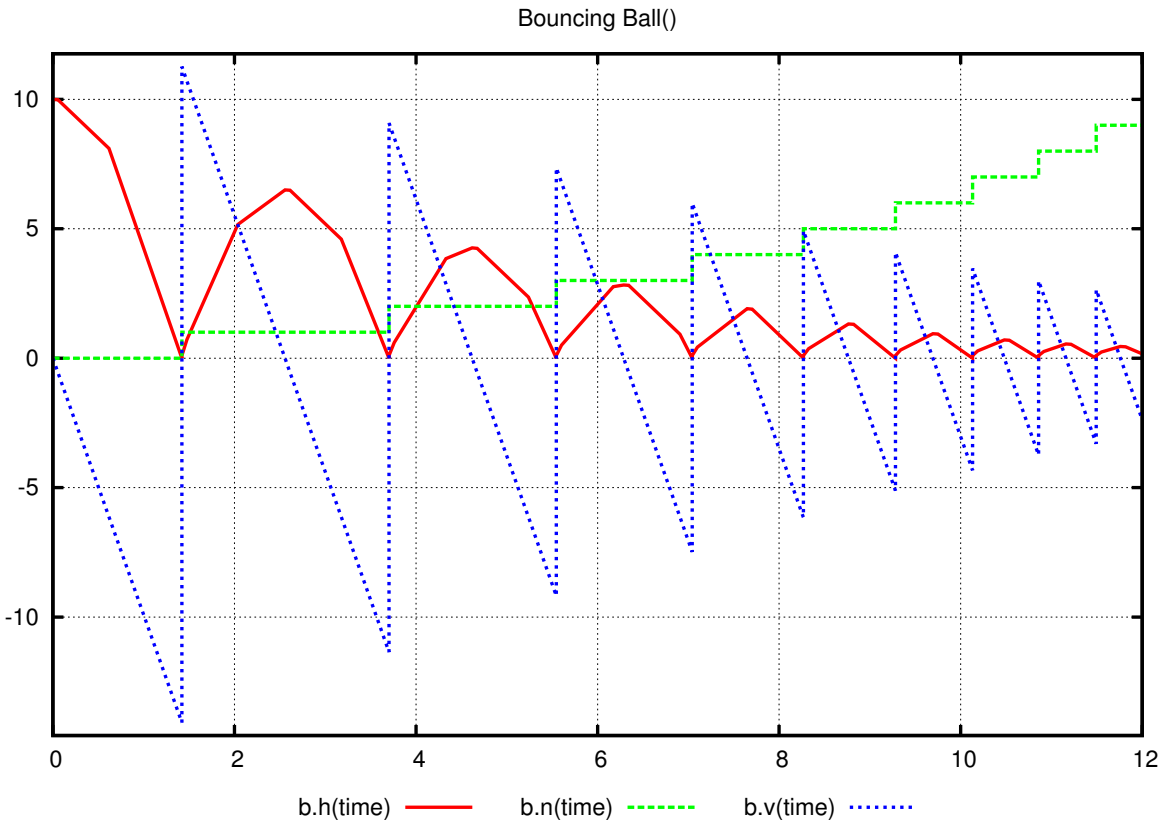

Figure 6: Simulation results of the automaton representation of the bouncing ball.

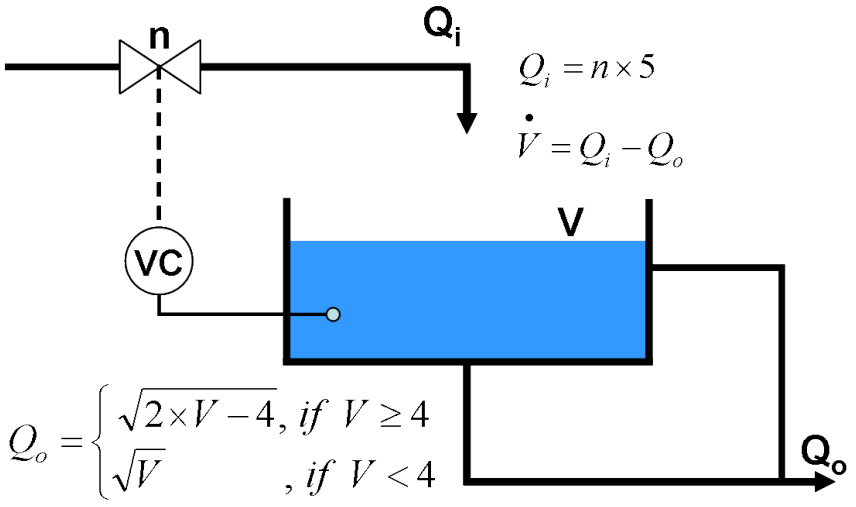

Figure 7: Scheme of the simple tank controller.

6.2 Tank controller

The tank controller model features a controller as well as hybrid dynamics. It consists of a tank, with an inflow of with volume flow Q_i that is regulated by valve n, and the outgoing volume flow Q_o, which dynamics change discrete depending on the volume of the tank. The controller of the tank must maintain the volume of the tank V between 2 and 10, by opening and closing the valve when appropriate. The scheme of the model is depicted in Figure 7.

The gPROMS specification corresponding to the model described above is presented in Listing 6.

Listing 6: Model of a bouncing ball.

```
DECLARE TYPE
 CIF_REAL_TYPE = 1.0 : -1.0E+200 : 1.0E+200
END

MODEL TankController_Model

VARIABLE
 pc1 AS CIF_REAL_TYPE
 pc2 AS CIF_REAL_TYPE
 pc3 AS CIF_REAL_TYPE
 n AS CIF_REAL_TYPE
 V AS CIF_REAL_TYPE
 Qi AS CIF_REAL_TYPE
 Qo AS CIF_REAL_TYPE
 t1 AS CIF_REAL_TYPE

SELECTOR
 stateSelector_pc3_0 AS (state_pc3_1, state_pc3_2)

EQUATION
 $t1 = 1;
 $V = Qi - Qo ;
 Qi = n * 5 ;
 CASE stateSelector_pc3_0 OF
  WHEN state_pc3_1 :
   Qo = SQRT(ABS(V + (V - 4))) ;
   SWITCH TO state_pc3_2 IF pc3 = 2 ;
  WHEN state_pc3_2 :
   Qo = SQRT(V) ;
   SWITCH TO state_pc3_1 IF pc3 = 1 ;
 END
END

PROCESS TankController

UNIT
 mainModel AS TankController_Model

ASSIGN
 mainModel.n := 0 ;
 mainModel.pc1 := 1 ;
```

```
 mainModel.pc2 := 1 ;
 mainModel.pc3 := 1 ;

INITIALSELECTOR
 mainModel.stateSelector_pc3_0 := mainModel.state_pc3_1;

INITIAL
 mainModel.t1 = 0;
 mainModel.V = 10 ;

SCHEDULE
 PARALLEL
  WHILE TRUE DO
   PARALLEL
    SEQUENCE
     CONTINUE UNTIL mainModel.pc2 = 1 AND mainModel.V <= 2
     RESET
      mainModel.pc2 := 2 ;
     END
     RESET
      mainModel.n := 1 ;
     END
    END
    SEQUENCE
     CONTINUE UNTIL mainModel.pc2 = 2 AND mainModel.V >= 10
     RESET
      mainModel.pc2 := 1 ;
     END
     RESET
      mainModel.n := 0 ;
     END
    END
    SEQUENCE
     CONTINUE UNTIL mainModel.pc3 = 1 AND mainModel.V < 4
     RESET
      mainModel.pc3 := 2 ;
     END
    END
    SEQUENCE
     CONTINUE UNTIL mainModel.pc3 = 2 AND mainModel.V >= 4
     RESET
      mainModel.pc3 := 1 ;
     END
    END
   END
  END
  SEQUENCE
   CONTINUE UNTIL mainModel.t1 > 10
   STOP
  END
 END
END
```

The simulation results of the controlled tank model specification are shown in Figure 8. This gPROMS specification is transformed to a hybrid automaton, and simulated using the CIF simulator. The simulation results for this automaton are shown in Figure 9. These graphs show that the obtained behavior is similar modulo numerical inaccuracies, which are the result of the implementation of both simulators.

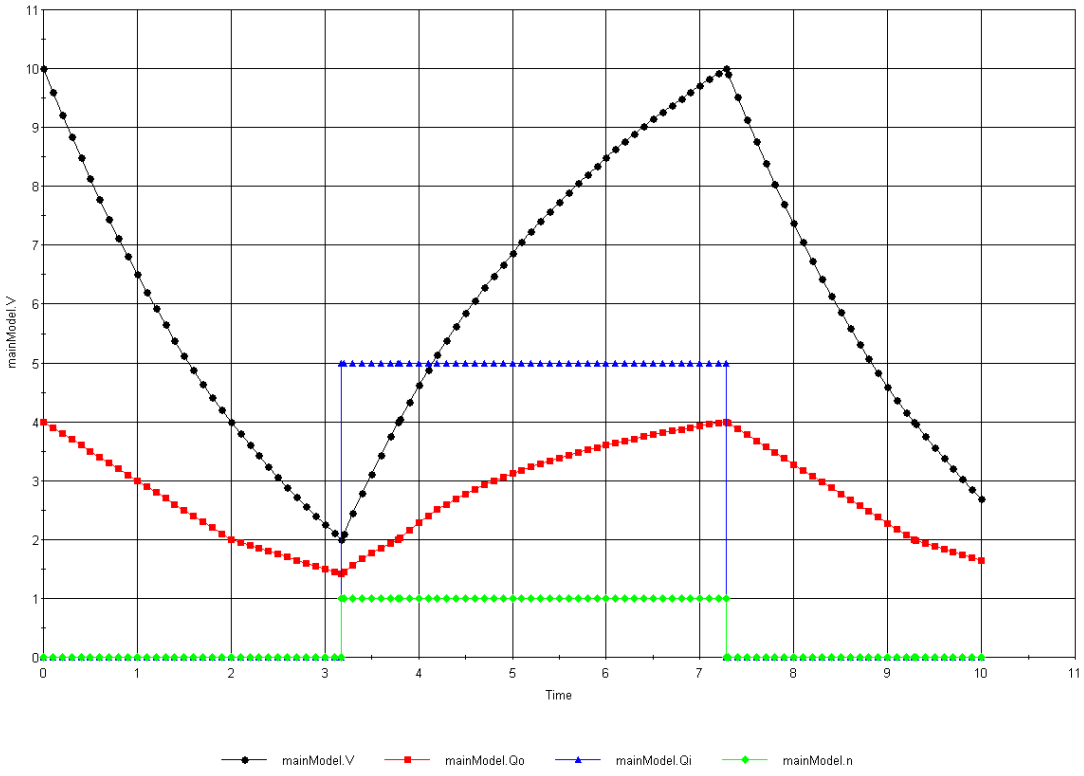

Figure 8: Simulation results of the controlled tank.

7 Conclusions

In this paper we have defined the formal semantics of an expressive subset of gPROMS, obtaining a simple and lean semantics for the behavioral constructs, where non-behavioral aspects of the language are formalized using denotational semantics. We hope that this in itself constitutes a significant result for the community that plans to develop semantic-preserving model transformations to and from this language.

We addressed the problem of validating the semantic model of the language by using a procedure for obtaining hybrid automata out of gPROMS compositions. This procedure uses our SOS rules, and is proven to yield an equivalent model that can be interpreted using simulation and model checking

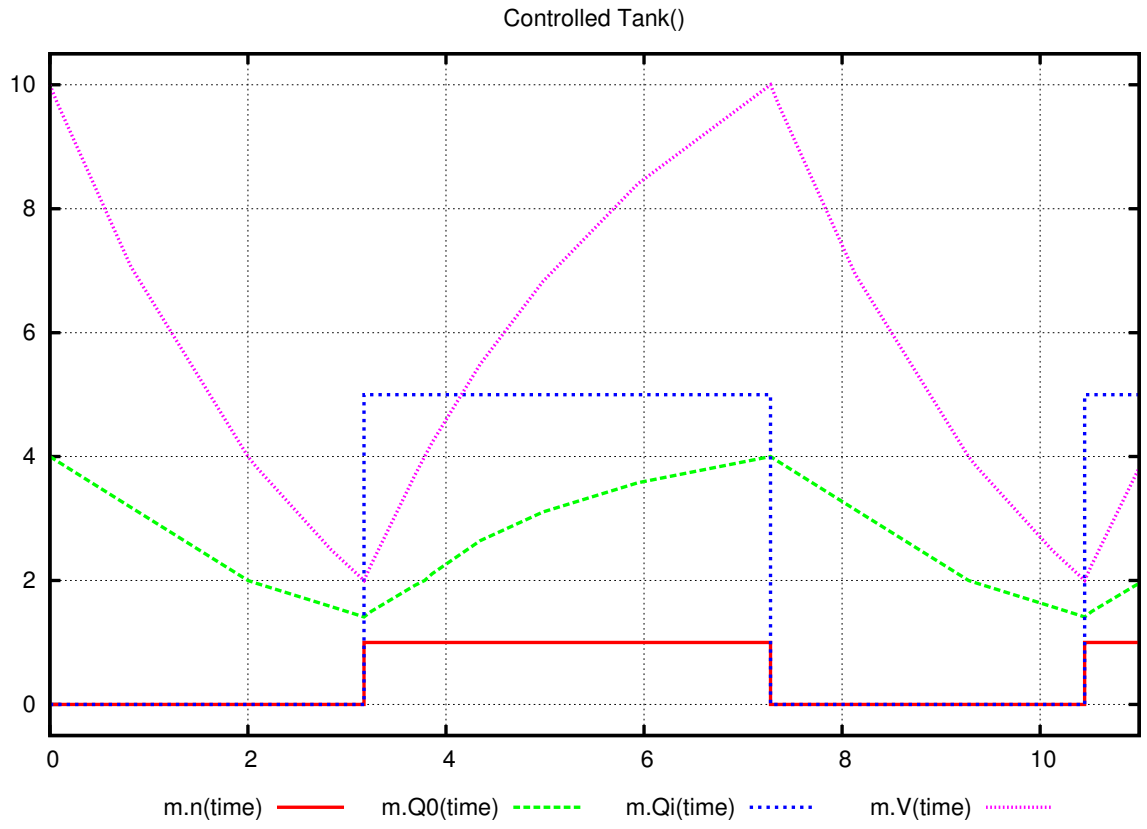

Figure 9: Simulation results of the automaton representation of the controlled tank.

tools. As an interesting side-effect, we obtained a semantic-preserving transformation from gPROMS compositions to hybrid automata.

We believe that this technique of validation can be used for inferring the SOS semantics of other modeling languages, whose sole semantic description is the implementation of a simulator. One of our long term goals is to be able to generalize and automate as much of this process as possible.

Currently, we were only able to use the CIF simulator as our interpreter, since model checking tools such as SpaceEx are limited in the conditions they allow in the invariants and guards of the input automata. In the future we would like to translate STS's into a form that satisfies these constrains.

This work represents a first step towards adding support for validation of gPROMS specifications. We are aware that the model checking of gPROMS models is limited to hybrid automata with dynamics that can be analyzed using model-checking tools for hybrid systems. Further research into abstraction techniques is another interesting line of research.

Acknowledgments

The authors wish to thank Dennis Hendriks for providing technical support while implementing the model transformations, and during the use of CIF 2 simulator.

References

[1] Damian Nadales Agut and Michel Reniers. Linearization of cif through sos. In Bas Luttik and Frank Valencia, editors, *EXPRESS*, volume 64 of *EPTCS*, pages 74–88, 2011.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[3] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, 1st edition, December 2009.

[4] J.C.M. Baeten, D.A. van Beek, D. Hendriks, A.T. Hofkamp, D.E. Nadales Agut, J.E. Rooda, and R.R.H. Schiffelers. Multiform Deliverable D1.1.2 Report describing the extended CIF functionality. `http://www.multiform.bci.tu-dortmund.de/images/stories/multiform/deliverables/multiform_d112.pdf`, 2010.

[5] D. A. van Beek, P. Collins, D. E. Nadales, J.E. Rooda, and R. R. H. Schiffelers. New concepts in the abstract format of the Compositional Interchange Format. In A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, editors, *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255, Zaragoza, Spain, 2009.

[6] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129–210, 2006.

[7] D. A. van Beek, M. A. Reniers, R. R. H. Schiffelers, and J. E. Rooda. Foundations of an interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio Butazzo, editors, *Hybrid Systems: Computation and Control, 10th International Workshop*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600, Pisa, 2007. Springer-Verlag.

[8] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jrg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin / Heidelberg, 2004.

[9] Francesco Casella and Alberto Leva. Modelling of thermo-hydraulic power generation processes using modelica. *Mathematical and Computer Modelling of Dynamical Systems*, 12(1):19–33, 2006.

[10] P. J. L. Cuijpers and M. A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005.

[11] P.J.L. Cuijpers and M.A. Reniers. Lost in translation: Hybrid-time flows vs real-time transitions. In *HSCC 2008*, volume 4981 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 2008.

[12] Goran Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. PhD thesis, Universiteit of Nijmegen, 2005.

[13] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer-Verlag, 2005.

[14] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[15] Manish Goyal and Goran Frehse. Translation between CIF and SpaceEx/PHAVer. Technical report, MULTIFORM consortium, 2011.

[16] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag.

[17] M. Hennessy and H. Lin. Symbolic bisimulations. In *Selected papers of the meeting on Mathematical foundations of programming semantics*, pages 353–389, Amsterdam, The Netherlands, The Netherlands, 1995. Elsevier Science Publishers B. V.

[18] T. A. Henzinger. The theory of hybrid automata. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Science*, pages 265–292. Springer-Verlag, New York, 2000.

[19] Karsten-Ulrich Klatt and Wolfgang Marquardt. Perspectives for process systems engineering - personal views from academia and industry. *Computers & Chemical Engineering*, 33(3):536–550, 2009.

[20] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata revisited. In *Proceedings Fourth International Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, pages 403–417. Springer-Verlag, 2001.

[21] M. R. Mousavi, M. A. Reniers, and J. F. Groote. Notions of bisimulation and congruence formats for SOS with data. *Information and Computation*, 200(1):107–147, 2005.

[22] MULTIFORM consortium. Integrated multi-formalism tool support for the design of networked embedded control systems MULTIFORM. `http://www.multiform.bci.tu-dortmund.de`, 2008.

[23] D. E. Nadales Agut and M. A. Reniers. Deriving a simulator for a hybrid language using SOS rules. http://se.wtb.tue.nl/sewiki/cif/publications2, May 2011.

[24] D.E. Nadales Agut, M. A. Reniers, R.R.H Schiffelers, K.Y. Jørgensen, and D. A. van Beek. A semantic-preserving transformation from the Compositional Interchange Format to UPPAAL. In *18th Triennial World Congress of the International Federation of Automatic Control*, Milano, 2011. CD-ROM.

[25] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer, 1993.

[26] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[27] PSE. gPROMS. `http://www.psenterprise.com/gproms/`, 2011.

[28] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, New York, NY, USA, 1999.

[29] C. Sonntag. Modeling, simulation, and optimization environments. In *Handbook of Hybrid Systems Control - Theory, Tools, Applications*, pages 325–360. Cambridge University Press, 2009.

[30] Systems Engineering Group TU/e. CIF 2 tooling. `http://devel.se.wtb.tue.nl/trac/cif`, 2011.

A Transforming transition systems to hybrid automata

In this section we describe the syntax and semantics of the hybrid automata that can be analyzed by PHAVer [13] (and hence by SpaceEx), and we give a transformation from gPROMS specifications to this formalism. We start by defining hybrid automata, based on the characterization of [12].

Definition 11 (Hybrid automaton)**.** *A hybrid automaton H is a tuple*

$$(Loc, Var, Lab, Edg, Act, Inv, Init)$$

where:

- $Loc \subseteq \mathscr{L}$ *is a set of locations,* $\mathscr{L}$ *is the set of all locations.*
- $Var \subseteq \mathscr{V}$ *is a set of variables.*
- $Lab \subseteq \mathscr{A}_\tau$ *is a set of synchronization labels,* $\mathscr{A}$ *is the set of all actions,* $\tau \notin \mathscr{A}$ *is the silent action, and* $\mathscr{A}_\tau \triangleq \mathscr{A} \cup \{\tau\}$.
- $Edg \subseteq (Loc \times Lab \times \mathscr{P} \times Loc)$ *is a set of edges.*

- *Act* $\in (Loc \to \mathscr{P})$ *is the activity function, which associates a flow predicate with each location.*
- *Inv* $\in (Loc \to \mathscr{P})$ *is the invariant function, which associates an invariant with each location.*
- *Init* $\in (Loc \to \mathscr{P})$ *is the initial predicate function, which associates initial conditions to each locations.*

Unlike the definition in [12], we use predicates to define relations between pairs of valuations, and also to define allowed values of variables, and trajectories. The two definitions are equivalent if we assume the following:

1. For each set of pairs of valuations $R \subseteq \Sigma \times \Sigma$, there is a predicate $u \in \mathscr{P}$ such that for each σ, and σ'
$$(\sigma, \sigma') \in R \Leftrightarrow \mathbf{old}(\sigma) \cup \sigma' \models u.$$
2. For each set of functions $F \subseteq (\mathbb{T} \rightharpoonup \Sigma)$, there is a predicate $f \in \mathscr{P}$ such that for each $flow$:
$$\rho \in F \Leftrightarrow \rho \models_{\text{flow}} f.$$
3. For each set of valuations $I \subseteq \Sigma$ there is a predicate $u \in \mathscr{P}$ such that for each σ:
$$\sigma \in I \Leftrightarrow \sigma \models u.$$

The semantics of a hybrid automaton is defined in terms of *timed transition systems*, that is, every hybrid automaton induces a *timed transition system* (TTS), which constitutes its semantic domain. The transition system space of hybrid automata is generated according to the rules of Table 24, where we use the following conventions:

$$\alpha = (Loc, Var, Lab, Edg, Act, Inv, Init)$$
$$\alpha[\ell] = (Loc, Var, Lab, Edg, Act, Inv, \mathrm{id}_\ell)$$

and id_v denotes the function that returns true only if the location equals v. More specifically, $\mathrm{id}_v \in Loc \to \mathbb{B}$, and $\mathrm{id}_v(w) \triangleq v \equiv w$, where $\equiv$ denotes syntactic equivalence.

$$\frac{\sigma \models Init(\ell), \qquad (\ell, a, r, \ell') \in Edg, \qquad \mathbf{old}(\sigma) \cup \sigma' \models r, \qquad \sigma \overset{\overline{\mathrm{NFV}(r)}}{=} \sigma', \qquad \sigma \models Inv(\ell), \qquad \sigma' \models Inv(\ell')}{(\alpha, \sigma) \xrightarrow{a} (\alpha[\ell'], \sigma')} \; 81$$

$$\frac{\sigma \models Init(\ell), \qquad \mathrm{dom}(\rho) = [0,t], \qquad 0 < t, \qquad \rho \models_{\text{flow}} Act(\ell), \qquad \rho \models Inv(\ell), \qquad \sigma = \rho(0) \restriction_{\mathscr{V}}}{(\alpha, \sigma) \xrightarrow{t} (\alpha[\ell], \rho(t) \restriction_{\mathscr{V}})} \; 82$$

Table 24: Semantic rules of hybrid automata

Note that in the SOS rules we are using automata as the first component of the states of the hybrid transition system, instead of just locations. This is necessary to compare gPROMS process terms and automata: to check bisimilarity we cannot just compare locations, since locations are meaningless without an accompanying structure. We also need to compare the entire automaton.

Given a gPROMS composition, and the symbolic transition system space of gPROMS transitions (which is obtained from the symbolic SOS rules), we can construct a *hybrid automaton*. The idea is to

translate every symbolic action transition into an edge, and to use the symbolic time transitions to create invariants and activity functions.

The main difficulty in this transformation is the presence of switching conditions in the symbolic transition system. In the definition of hybrid automaton there is no corresponding concept, and therefore we must use existing ones to represent it. Activities cannot be used for this purpose, since they can contain inequalities and expressions that are not supported by hybrid automata analysis tools (activities are used to specify differential equations and not switching conditions). Then, we are only left with hybrid automata invariants as our only concept to represent gPROMS switching conditions.

One of the problems with the use of invariants is the fact that they are checked once the system enters a new location, and this is not the case for switching conditions. Using an idea presented in [25], in the resulting automaton we introduce a fresh clock variable c, and we translate every switching condition p into a corresponding invariant $c = 0 \vee p$. This requires c to be reset every time a transition is executed.

Property 6. *Let e be a **continuous-closed** predicate, and $\rho \in [0,t] \to \$\Sigma$, $0 < t$, then the following implications hold:*

$$(\rho \restriction_{[0,t)} \models_{\text{flow}} e) \Rightarrow (\rho(t) \models_{\text{flow}} e)$$
$$(\rho \restriction_{(0,t]} \models_{\text{flow}} e) \Rightarrow (\rho(0) \models_{\text{flow}} e)$$
$$(\rho \restriction_{[0,t)} \models e) \Rightarrow (\rho(t) \models e)$$
$$(\rho \restriction_{(0,t]} \models e) \Rightarrow (\rho(0) \models e)$$

The following definition describes how to obtain a hybrid automaton out of a gPROMS symbolic transition system.

Definition 12 (Hybrid automaton induced by a gPROMS composition)**.** *Given a gPROMS composition p, the automaton induced by p is a tuple*

$$(Loc, Var, Lab, Edg, Act, Inv, Init)$$

where:

- *c is a variable that does not appear free in p.*
- *The initial predicate function is given by* [4]*:*

$$Init(q) = \begin{cases} \text{iconds}(p) & \text{if } q \equiv p \\ \text{false} & \text{otherwise} \end{cases}$$

- *The set of locations is defined as the least set that satisfies:*
  1. *$p, \bot, \checkmark \in Loc$ where $\checkmark$, and $\bot$ are special symbols that do not belong to $\mathscr{G}$.*
  2. *If $q \in Loc$ and $\langle q \rangle \xrightarrow{b,r} \langle x \rangle$ then $x \in Loc$, where $x \in \mathscr{G} \cup \{\checkmark, \bot\}$.*
- *The set of edges is defined as the least set that satisfies:*
  - *If $q \in Loc$ and $\langle q \rangle \xrightarrow{b,r} \langle x \rangle$ then $(q, \tau, \mathbf{old}(b) \wedge r \wedge c = 0, x) \in Edg$, where $x \in Loc$.*
- *The activity function is defined as:*

$$Act = \{(q,f) \mid q \in Loc \wedge f \equiv \$c = 1 \wedge \langle \bigvee f_i, g_i : \langle q \rangle \xrightarrow{f_i, g_i} : f_i \rangle\}$$

[4] Note that: for a composition other than a gPROMS specification, $\text{iconds}(p) \equiv \text{true}$.

- *The invariant function is defined as:*

$$Inv = \{(q,g) \mid q \in Loc \wedge g \equiv c = 0 \vee \langle \bigvee f_i, g_i : \langle q \rangle \xrightarrow{f_i, g_i} : g_i \rangle\}$$

- $Lab = \{\tau\}$

Let α be the automaton induced by p, then we denote this as

$$\alpha = \mathscr{T}(p)$$

The set of all automata obtained from gPROMS compositions, using this definition, is denoted as $\mathscr{H}_g$.

Note that if a location has no time transitions, then the resulting invariant will be $c = 0$, which means that no time can pass.

To be able to compute a hybrid automaton from a gPROMS composition, it is necessary that the resulting transition system is finite.

Property 7 (Finiteness of the induced STS)**.** *The symbolic transition system induced by a gPROMS composition is finite.*

Proof. The proof goes via structural induction on the gPROMS process terms.

For the atomic gPROMS compositions (equalities, assignments, continue, and stop statements) the property holds trivially.

The inductive case can be proven easily. The only non-trivial case are the `WHILE` statements. Assume:

$$p \equiv \textbf{WHILE } b \textbf{ DO } s \textbf{ END}$$

By induction hypothesis, we know that the symbolic transition system induced by s is finite. Then, the only reachable states from p, are those of the form:

- $\langle \textbf{SEQUENCE } s : [p] \textbf{ END} \rangle$.
- $\langle \textbf{SEQUENCE } s' : [p] \textbf{ END} \rangle$, where s' is reachable from s.
- $\langle p \rangle$ itself.

Since the number of reachable states from s is finite, the number of states reachable from p is also finite. And since s is finitely branching, and the symbolic rules for while statements only add two edges, the symbolic transition system induced by p is finitely branching, and therefore finite. This concludes the proof for this case.

It is interesting to observe that the above argument does not work if the rule for sequential composition allows transitions of the form:

$$\langle \textbf{SEQUENCE } p : [q] \textbf{ END} \rangle \xrightarrow{b,r} \langle \textbf{SEQUENCE } [q] \textbf{ END} \rangle$$

since from p we would reach the state **SEQUENCE** $[p]$ **END**. In fact, if such transitions were allowed, the induced symbolic transition system would be infinite. These kind of transitions are prevented by our set of rules. □

The following corollary follows from the previous property.

Corollary 2 (Finiteness of the induced HA)**.** *The hybrid automaton induced by a gPROMS composition is finite.*

It is our goal to show that for every gPROMS specification there is a hybrid automaton such that they are equivalent. It does not seem possible to prove stateless bisimulation, thus the notion of equivalence we use is state-based bisimulation. To this end, we define the notion of state-based bisimilarity between gPROMS compositions and hybrid automata.

Definition 13 (gPROMS $\underline{\leftrightarrow}$ HA)**.** *A relation $R \subseteq (\mathscr{G} \times \Sigma) \times (\mathscr{H}_g \times \Sigma)$ is a* state-based bisimulation *relation if and only if for all p, α, σ_0, σ_1 such that $((p,\sigma_0),(\alpha,\sigma_1)) \in R$ we have $\sigma_0 = \sigma_1$ and:*

1. $\langle \forall p', \sigma' :: (p,\sigma_0) \to (p',\sigma') \Rightarrow \langle \exists \alpha' :: (\alpha,\sigma_1) \xrightarrow{\tau} (\alpha',\sigma') \wedge ((p',\sigma'),(\alpha',\sigma')) \in R \rangle \rangle$
2. $\langle \forall \alpha', \sigma' :: (\alpha,\sigma_1) \xrightarrow{\tau} (\alpha',\sigma') \Rightarrow \langle \exists p' :: (p,\sigma_0) \to (p',\sigma') \wedge ((p',\sigma'),(\alpha',\sigma')) \in R \rangle \rangle$
3. $\langle \forall p', \sigma', \rho :: (p,\sigma_0) \overset{\rho}{\longmapsto} (p',\sigma') \Rightarrow \langle \exists \alpha' :: (\alpha,\sigma_1) \overset{\sup(\mathrm{dom}(\rho))}{\longmapsto} (\alpha',\sigma') \wedge ((p',\sigma'),(\alpha',\sigma')) \in R \rangle \rangle$
4. $\langle \forall \alpha', \sigma', t :: (\alpha,\sigma_1) \overset{t}{\longmapsto} (\alpha',\sigma') \Rightarrow \langle \exists p', \rho :: \mathrm{dom}(\rho) = [0,t] \wedge (p,\sigma_0) \overset{\rho}{\longmapsto} (p',\sigma') \wedge ((p',\sigma'),(\alpha',\sigma')) \in R \rangle \rangle$

A gPROMS composition p and a hybrid automaton α are state-based bisimilar*, denoted as $p \underline{\leftrightarrow} \alpha$ if and only if there exists a state-based bisimulation relation R such that $(p,\alpha) \in R$.*

In the definition of the atomic automaton induced by a gPROMS composition we have used a fresh variable c, which acts as a clock. To prove that a gPROMS specification and its induced automaton are state-based bisimilar, we need to abstract away from the values of this variable. Formally, this is done by means of the *variable scope operator*.

Given a variable x, a value v, and a automaton α, the composition $[\![_{\mathrm{V}}\, x = v :: \alpha\,]\!]$ behaves as α, except that all changes to variable x are not visible outside the scope operator (the value of variable x remains constant). Value v represents the local value of variable x. The rules for this operator are given in Table 25.

$$\frac{(\alpha, \{x \mapsto v\} \succ \sigma) \to (\alpha', \{x \mapsto v'\} \succ \sigma'), \qquad \sigma(x) = \sigma'(x)}{([\![_{\mathrm{V}}\, \{x \mapsto v\} :: \alpha\,]\!], \sigma) \to ([\![_{\mathrm{V}}\, \{x \mapsto v'\} :: \alpha'\,]\!], \sigma')}\ 83$$

$$\frac{(\alpha, \{x \mapsto v\} \succ \sigma) \overset{t}{\longmapsto} (\alpha', \{x \mapsto v'\} \succ \sigma'), \qquad \sigma(x) = \sigma'(x)}{([\![_{\mathrm{V}}\, \{x \mapsto v\} :: \alpha\,]\!], \sigma) \overset{t}{\longmapsto} ([\![_{\mathrm{V}}\, \{x \mapsto v'\} :: \alpha'\,]\!], \sigma')}\ 84$$

Table 25: SOS rules for the variable scope operator

Using the notion of state-based bisimilarity, we state the main theorem of this section.

Theorem 2 (Relation between gPROMS and hybrid automata)**.** *For each gPROMS composition, $p \in \mathscr{G}$, its induced hybrid automaton is state-based bisimilar. This is:*

$$p \underline{\leftrightarrow} [\![_{\mathrm{V}}\, c \mapsto v :: \mathscr{T}(p)\,]\!]$$

where c is the fresh clock variable used to translate switching conditions into invariants, and v is an arbitrary value.

Proof. Henceforth we assume:

$$\mathscr{T}(p) = (Loc, Var, Lab, Edg, Act, Inv, Init)$$

Then, it can be proven that relation R defined as:

$$R \triangleq \{((q,\sigma),(\llbracket_{\mathrm{V}}\, c = v :: \mathscr{T}(p)[q]\,\rrbracket,\sigma)) \mid q \in Loc \wedge v \in \Lambda \wedge \{c \mapsto v\} \succ \sigma \models Inv(q)\}$$

is a state-based bisimulation relation.

The proof of the above fact requires Property 6 (continuous-closed predicates), to be able to infer that the validity of a urgency condition u in $[0,t)$ implies the validity of u (invariant) in $[0,t]$. Another crucial point in the proof is the fact that, in the symbolic Rule 55 condition $v = x$ is added to the flow condition as well to the urgency condition, since this allow us to ensure the existence of a symbolic time transition for each time transition in the atomic automaton.

With the use of the soundness and completeness results this property can be easily proven, since from each explicit transition generated by a gPROMS composition it can be inferred that there is a corresponding symbolic transition, and therefore a corresponding edge in the automaton, which generates an explicit transition of the same nature as the composition. The converse reasoning can be used to prove that every explicit transition in the induced automaton corresponds with an explicit transition in the originating composition. □

B Syntax of gPROMS

B.0.1 EBNF of gPROMS$_{core}$

⟨gPROMS$_{core}$⟩ → ⟨DECLARE$_{core}$⟩* ⟨MODEL$_{core}$⟩ ⟨PROCESS$_{core}$⟩.

⟨DECLARE$_{core}$⟩ →
DECLARE TYPE ⟨declareType$_{core}$⟩
END.

⟨declareType$_{core}$⟩ →
⟨typeID⟩ **=** ⟨initialValue⟩ : ⟨lowerBound⟩ : ⟨upperBound⟩.

⟨MODEL$_{core}$⟩ →
MODEL ⟨ID⟩
PARAMETER ⟨PARAMETER$_{core}$⟩*
VARIABLE ⟨VARIABLE$_{core}$⟩*
SELECTOR ⟨SELECTOR$_{core}$⟩*
EQUATION ⟨EQUATION$_{core}$⟩*
END.

⟨PROCESS$_{core}$⟩ →
PROCESS ⟨ID⟩

UNIT ⟨UNIT$_{core}$⟩
SET ⟨SET$_{core}$⟩*
ASSIGN ⟨ASSIGN$_{core}$⟩*
INITIALSELECTOR ⟨INITIALSELECTOR$_{core}$⟩*
INITIAL ⟨INITIAL$_{core}$⟩*
SCHEDULE ⟨SCHEDULE$_{core}$⟩
END.

⟨typeID⟩ → ⟨ID⟩.

⟨initialValue⟩ → ⟨REAL⟩.

⟨lowerBound⟩ → ⟨REAL⟩.

⟨upperBound⟩ → ⟨REAL⟩.

⟨text⟩ → ⟨STRING⟩.

⟨PARAMETER$_{core}$⟩ →
⟨ID⟩ **AS** (**INTEGER**|**REAL**|**LOGICAL**).

⟨VARIABLE$_{core}$⟩ →
⟨ID⟩ **AS** ⟨typeID⟩.

⟨SELECTOR$_{core}$⟩ →
⟨ID⟩ **AS** (⟨ID⟩ (**,** ⟨ID⟩)*).

⟨SET$_{core}$⟩ →
⟨ID⟩ **:=** ⟨expression$_{core}$⟩ .

⟨ASSIGN$_{core}$⟩ →
⟨ID⟩ **:=** ⟨expression$_{core}$⟩ .

⟨INITIALSELECTOR$_{core}$⟩ →
⟨pathID⟩ **:=** ⟨pathID⟩ **;**.

⟨INITIAL$_{core}$⟩ → ⟨expression$_{core}$⟩ **=** ⟨expression$_{core}$⟩**;**.
⟨UNIT$_{core}$⟩ → ⟨ID⟩ **AS** ⟨typeID⟩.

⟨pathID⟩ → ⟨ID⟩ (**.** ⟨ID⟩)*.

⟨ID⟩ → (**a**..**z**|**A**..**Z**) (**a**..**z**|**A**..**Z**|_|**0**..**9**)*.

⟨dim⟩ → ⟨ID⟩ | ⟨number⟩.

⟨number⟩ →
(**-**)? (**0**|(**1**..**9**)(**0**..**9**)*)(**e** (**-** | **+**)? (**0**..**9**)+)?.

⟨REAL⟩ →
(**-**)? (**0**|(**1**..**9**)(**0**..**9**)*)(**.**? (**0**..**9**)+ (**e** (**-** | **+**)? (**0**..**9**)+)?).

⟨STRING⟩ → Any character which is not a comment sign or a keyword.

⟨EQUATION$_{core}$⟩ →
(⟨expression$_{core}$⟩ **=** ⟨expression$_{core}$⟩) | ⟨conditionalEquation$_{core}$⟩ **;**.

⟨expression$_{core}$⟩ → Without PARTIAL and INTEGRAL TODO! CIF compatible?.

⟨conditionalEquation$_{core}$⟩ → ⟨CASE-WHEN-construct$_{core}$⟩.

⟨CASE-WHEN-construct$_{core}$⟩ →
CASE ⟨ID⟩ **OF** (**WHEN** ⟨ID⟩ : ⟨EQUATION$_{core}$⟩+ (**SWITCH TO** ⟨ID⟩ **IF** ⟨logicalExpression$_{core}$⟩ **;**)+)+ **END**.

⟨logicOperator⟩ → **AND** | **OR**.

⟨logicalExpression$_{core}$⟩ → logical/boolean expression

⟨SCHEDULE$_{core}$⟩ →
⟨SEQUENCE$_{core}$⟩ | ⟨PARALLEL$_{core}$⟩ | ⟨controlStructures$_{core}$⟩ | ⟨CONTINUE$_{core}$⟩ | ⟨RESET$_{core}$⟩ | ⟨REINITIAL$_{core}$⟩ | ⟨SWITCH$_{core}$⟩ | ⟨STOP⟩.

⟨SEQUENCE$_{core}$⟩ → **SEQUENCE** ⟨SCHEDULE$_{core}$⟩+ **END**.

⟨PARALLEL$_{core}$⟩ → **PARALLEL** ⟨SCHEDULE$_{core}$⟩+ **END**.

⟨controlStructures$_{core}$⟩ → ⟨WHILE$_{core}$⟩ | ⟨IF-THEN-ELSE$_{core}$⟩.

⟨WHILE$_{core}$⟩ → **WHILE** ⟨logicalExpression⟩ **DO** ⟨SCHEDULE$_{core}$⟩ **END**.

⟨IF-THEN-ELSE$_{core}$⟩ →
IF ⟨logicalExpression$_{core}$⟩ **THEN** ⟨SCHEDULE$_{core}$⟩ (**ELSE** ⟨SCHEDULE$_{core}$⟩)? **END**.

⟨CONTINUE$_{core}$⟩ →

CONTINUE UNTIL ⟨logicalExpression$_{core}$⟩ **;**.

⟨RESET$_{core}$⟩ →
RESET ⟨pathID⟩ **:=** ⟨expression$_{core}$⟩ **END**.

⟨REINITIAL$_{core}$⟩ →
REINITIAL ⟨pathID⟩ **WITH** ⟨EQUATION$_{core}$⟩**END**.

⟨SWITCH$_{core}$⟩ →
SWITCH ⟨pathID⟩ **:=** ⟨pathID⟩ **; END**.

⟨STOP⟩ → **STOP**.

C Implementation of gs2gsd and iconds

```
{-
  Some considerations:

  - Whenever I see a star following a constructor in Martin's grammar,
    I use a list. Whenever I see a plus I also use a list. This keeps
    notation simple, but ommits the fact that lists must have at least one
    element. So beware.

-}

-- The SourceCode class contains a function for showing an indented
-- piece of code.
class SourceCode a where
  -- showInd shows the piece of code. The firstn parameter is the tab
  -- stop. The block SHOULD NOT end in new line.
  showInd ::  String -> a -> String
  showInd tab x = "TODO"
  -- showBlocksInd shows a sequence of blocks, separated by newlines.
  showBlocksInd :: String -> [a] -> String
  showBlocksInd tab xs = foldr (++) "" (map (++"\n") (map (showInd tab) xs))

class Assignment a where
  -- This function returns the predicate associated to a given
  -- assignment. The returned predicate must be of the form 'Eql a b'
  asPred :: a -> Predicate

-- gPROMS specifications
data GpromsCore
     = GpromsSpec [Declare] Model Process

-- gPROMS system descriptions
```

```
data GpromsSD = MkSD [Equation] Schedule

data Declare
     = DeclareType Id Float Float Float

data Model
     = Model Id [Parameter] [Variable] [Selector] [Equation]

data Parameter
     = As Id Type

data Selector
     = MkSelector Id [Id]

data Equation
     = Equal Expression Expression
     | IF Predicate [Equation] [Equation] -- Equations seem to be
                                          -- separated by semicolons.
                                          -- I'm modelling them as
                                          -- lists.
     | CASE Id [WhenClause]

data WhenClause
     = WHEN Id [Equation] [SwitchClause]

data SwitchClause
     = SWITCHTO Id Predicate

-- Predicates and expressions
data Predicate
     = Eql Expression Expression
     | LessThan Expression Expression -- a < b is equal to LessThan a b
     | Leq Expression Expression -- a < b is equal to LessThan a b
     | Not Predicate
     | And Predicate Predicate
     | Or Predicate Predicate
     | TRUE

data Expression
     = Var Id
     | OldVar Id
     | F Float
     | I Int
     | Uminus Expression -- Unary minus: - e  = Uminus e
     | Plus Expression Expression
     | Minus Expression Expression
     | Times Expression Expression
     | Div Expression Expression

data Type = INTEGER | REAL | LOGICAL

data Variable = AsV Id TypeId
```

```
type Id = String

showId :: Id -> String
showId i = (rmchar quoteChar (show i))

-- Remove a specific charachter
rmchar c xs = filter (/= c) xs

type TypeId = Id

-- Gproms processes
data Process
     = Proc
       Id
       [UnitCore]
       [SetCore]
       [AssignCore]
       [InitialSelectorCore]
       [InitialCore]
       Schedule

-- This is used in gPROMS for equation instanciation
data UnitCore = AsU Id Id

-- This is used in gPROMS for parameter instanciation
data SetCore = Assign Id Expression

-- This is used to give initial values to DISCRETE equation variables
-- TODO: ask Martin whether this is correct.
type AssignCore = SetCore

-- This is used to initialize the case variables
data InitialSelectorCore = InitSelector Id Id

-- This is used to give initial values to CONTINUOUS
data InitialCore = InitEq Expression Expression

-- The schedule part
data Schedule
     = SEQUENCE [Schedule]
     | PARALLEL [Schedule]
     | WHILE Predicate Schedule
     | IFc Predicate Schedule Schedule
     | CONTINUE_UNTIL Predicate
     | RESET Id Expression
     | REPLACE Id Id Expression
     | REINITIAL Id Equation
     | SWITCH Id Id
     | STOP

-- The transformation function
```

```
gs2gsd :: GpromsCore -> GpromsSD
gs2gsd (GpromsSpec ds
        (Model j ps vs ks es)
        (Proc i us ts as is ls s)
       )
  = MkSD (concat [map (instantiate x) es | (AsU x m) <- us]) s

-- The function that extracts the list of initial conditions.

-- The list that is returned consists of elements of the form 'Eql a
-- b'.
iconds :: GpromsCore -> [Predicate]
iconds (GpromsSpec _ _ (Proc i us ts as is ls s))
  = (map asPred ts)++(map asPred as)++(map asPred is)++(map asPred ls)


class Instantiable a where
  -- instanciate adds "x." to all variables of the expression of the
  -- second parameter. In case a variable is of the form $v it will be
  -- replaced by $x.v
  instantiate :: Id -> a -> a
  instantiate x e = e -- By default nothing is done

instance Instantiable (Equation) where
  instantiate x (Equal a b) = Equal (instantiate x a) (instantiate x b)
  instantiate x (IF p xs ys) = (IF (instantiate x p) xs' ys')
    where xs'=map (instantiate x) xs
          ys'=map (instantiate x) ys
  instantiate x (CASE i ws) = CASE (prepend x i) ws'
    where ws'=map (instantiate x) ws

instance Instantiable (WhenClause) where
  instantiate x (WHEN i es ts) = (WHEN (prepend x i) es' ts')
    where es'=map (instantiate x) es
          ts'=map (instantiate x) ts

instance Instantiable (SwitchClause) where
  instantiate x (SWITCHTO i p) = (SWITCHTO (prepend x i) (instantiate x p))

instance Instantiable (Expression) where
  instantiate x (Var y) = (Var (prepend x y))
  instantiate x (OldVar y) = (OldVar (prepend x y))
  instantiate x (F f) = F f
  instantiate x (I i) = I i
  instantiate x (Uminus a) = (Uminus (instantiate x a))
  instantiate x (Plus a b) = (Plus (instantiate x a) (instantiate x b))
  instantiate x (Minus a b) = (Minus (instantiate x a) (instantiate x b))
  instantiate x (Times a b) = (Times (instantiate x a) (instantiate x b))
  instantiate x (Div a b) = (Div (instantiate x a) (instantiate x b))

instance Instantiable (Predicate) where
  instantiate x (Eql a b) = (Eql (instantiate x a) (instantiate x b))
```

```
  instantiate x (LessThan a b) = (LessThan (instantiate x a) (instantiate x b))
  instantiate x (Leq a b) = (Leq (instantiate x a) (instantiate x b))
  instantiate x (Not a) = (Not (instantiate x a))
  instantiate x (And a b) = (And (instantiate x a) (instantiate x b))
  instantiate x (Or a b) = (Or (instantiate x a) (instantiate x b))
  instantiate x TRUE = TRUE

prepend :: Id -> Id -> Id
prepend x ('$':xs) = '$':(x++"."++xs)
prepend x xs = x++"."++xs

{- Function show and showInd instanciations -}
-- Tab separator
ts :: String
ts = "␣␣␣␣"

prnt xs = "("++ xs ++")"

instance Show (GpromsCore) where
  show = showInd ""

instance SourceCode (GpromsCore) where
  showInd tab (GpromsSpec ds m p) = (showBlocksInd tab ds)
                                    ++(showInd tab m)
                                    ++ (showInd tab p)

instance SourceCode (Declare) where
  showInd tab (DeclareType t i l u) = tab++"DECLARE␣TYPE"++"\n"
                                      ++tab++ts++(showId t)++"="
                                      ++(show i)++":"
                                      ++(show l)++":"
                                      ++(show u)++"\n"
                                      ++tab++"END"

instance SourceCode (Model) where
  showInd tab (Model i ps vs ls es) = tab++"MODEL␣"++(showId i)++"\n"
                                      ++tab++ts++"PARAMETER"++"\n"
                                      ++(showBlocksInd (tab++ts++ts) ps)
                                      ++tab++ts++"VARIABLE"++"\n"
                                      ++(showBlocksInd (tab++ts++ts) vs)
                                      ++tab++ts++"SELECTOR"++"\n"
                                      ++(showBlocksInd (tab++ts++ts) ls)
                                      ++tab++ts++"EQUATION"++"\n"
                                      ++(showBlocksInd (tab++ts++ts) es)
                                      ++tab++"END"++"\n"

instance SourceCode (InitialCore) where
  showInd tab (InitEq a b) = tab++(show a)++"="++(show b)++";"

instance SourceCode (InitialSelectorCore) where
  showInd tab (InitSelector a b) = tab++(showId a)++":="++(showId b)++";"
```

```
instance SourceCode (Parameter) where
  showInd tab (As i t) = tab++(showId i)++"␣AS␣"++(show t)

instance SourceCode (Variable) where
  showInd tab v = tab++(show v)

instance SourceCode (Selector) where
  showInd tab (MkSelector i xs) = tab++(showId i)++"␣AS␣"++(showList xs)
    where showList [] = "()" -- "("++(map (++",") xs)++ ")"
          showList (x:xs) = "("++(foldl (++) x (map (",␣"++) xs))++ ")"

instance SourceCode (Equation) where
  showInd tab (Equal a b) = tab++(show a) ++ "=" ++ (show b)++";"
  showInd tab (IF p xs ys) = tab++"IF␣"++(show p)++"␣THEN\n"
                             ++(showBlocksInd (tab++ts) xs)
                             ++tab++"ELSE"++"\n"
                             ++(showBlocksInd (tab++ts) ys)
                             ++tab++"END"
  showInd tab (CASE i ws) = tab++"CASE␣"++(showId i)++"␣OF\n"
                            ++(showBlocksInd (tab++ts) ws)
                            ++tab++"END"

instance SourceCode (WhenClause) where
  showInd tab (WHEN i es ws) = tab++"WHEN␣"++(showId i)++":␣\n"
                               ++ (showBlocksInd (tab++ts) es)
                               ++ rmlastnl (showBlocksInd (tab++ts) ws)
                                 where rmlastnl xs =
                                         if (xs!!((length xs)-1)) == '\n'
                                           then take ((length xs)-1) xs
                                           else xs

instance SourceCode (SwitchClause) where
  showInd tab (SWITCHTO i p) = tab++"SWITCH␣TO␣"++ (showId i)
                               ++ "␣IF␣" ++ (show p) ++";"

instance SourceCode (Process) where
  showInd tab  (Proc i us es as ls is s) =
    tab ++"PROCESS␣"++(showId i)++"\n"
    ++tab++ ts ++ "UNIT\n"
    ++(showBlocksInd (ts++ts) us)
    ++tab++ ts ++ "SET\n"
    ++(showBlocksInd (ts++ts) es)
    ++tab++ ts ++ "ASSIGN\n"
    ++(showBlocksInd (ts++ts) as)
    ++tab++ ts ++ "INITIALSELECTOR\n"
    ++(showBlocksInd (ts++ts) ls)
    ++tab++ ts ++ "INITIAL\n"
    ++(showBlocksInd (ts++ts) is)
    ++tab++ ts ++ "SCHEDULE\n"
    ++ showInd (tab++ts) s++"\n"
    ++tab++"END"
```

```
instance SourceCode (Schedule) where
  showInd tab (SEQUENCE xs) = tab++"SEQUENCE\n"
                              ++ (showBlocksInd (ts++tab) xs)
                              ++tab++"END"
  showInd tab (PARALLEL xs) = tab++"PARALLEL\n"
                              ++ (showBlocksInd (ts++tab) xs)
                              ++tab++"END"
  showInd tab (WHILE p s) = tab++"WHILE␣"++(show p)++"␣DO\n"
                            ++(showInd (ts++tab) s)++"\n"
                            ++tab++"END"
  showInd tab (IFc p s t) = tab++"IF␣"++(show p)++"␣THEN\n"
                            ++(showInd (ts++tab) s)++"\n"
                            ++tab++"ELSE\n"
                            ++(showInd (ts++tab) t)++"\n"
                            ++tab++"END"
  showInd tab (CONTINUE_UNTIL p) = tab++"CONTINUE␣UNTIL␣"++show p++";\n"
  showInd tab (RESET i e) = tab++"RESET\n"
                            ++tab++ts++(showId i)++":="++(show e)++";\n"
                            ++tab++"END"
  showInd tab (REPLACE x y e) = tab++"REPLACE\n"
                          ++tab++ts++(showId x)++"\n"
                          ++tab++"WITH\n"
                          ++tab++ts++(showId y)++":="++(show e)++";\n"
                          ++tab++"END"
  showInd tab (REINITIAL i e) = tab++"REINITIAL\n"
                          ++tab++ts++(showId i)++"\n"
                          ++tab++"WITH\n"
                          ++tab++ts++(show e)++"\n"
                          ++tab++"END"
  showInd tab (SWITCH x y) = tab++"SWITCH\n"
                       ++tab++ts++(showId x)++":="++(showId y)++";\n"
                       ++tab++"END"
  showInd tab STOP = tab++"STOP"

instance Show (Schedule) where
  show s = showInd "" s

instance Show (Equation) where
  show  = showInd ""

instance Show (WhenClause) where
  show =  showInd ""

instance SourceCode (UnitCore) where
  showInd tab  (AsU i j) = tab++(showId i)++"␣AS␣"++(showId j)

instance SourceCode (SetCore) where
  showInd tab  (Assign i f) = tab++(showId i)++":="++(show f)++";"

instance Show (Process) where
  show = showInd ""
```

```
instance Show (Expression) where
  show (Var v) = rmchar quoteChar (show v)
  show (OldVar v) = "OLD(" ++ (show (Var v)) ++ ")"
  show (F f) = show f
  show (I i) = show i
  show (Uminus i) = "-" ++ (prnt (show i))
  show (Plus a b) = prnt ((show a) ++ "+" ++ (show b))
  show (Minus a b) =  prnt ((show a) ++ "-" ++ (show b))
  show (Times a b) =  prnt ((show a) ++ "*" ++ (show b))
  show (Div a b) =  prnt ((show a) ++ "/" ++ (show b))

instance Show (Predicate) where
  show (Eql a b) = (show a) ++ "=" ++ (show b)
  show (LessThan a b) = (show a) ++ "<" ++ (show b)
  show (Leq a b) = (show a) ++ "<=" ++ (show b)
  show (Not a) = "NOT␣"++ prnt (show a)
  show (And a b) = prnt ((show a) ++ "␣AND␣" ++ (show b))
  show (Or a b) = prnt ((show a) ++ "␣OR␣" ++ (show b))
  show TRUE = "TRUE"

instance Show (Type) where
  show INTEGER = "INTEGER"
  show REAL = "REAL"
  show LOGICAL = "LOGICAL"

instance Show (Selector) where
  show (MkSelector i xs) = (showId i)++"␣AS␣"++(show xs)

instance Show (Parameter) where
  show p = showInd "" p

instance Show (Variable) where
  show (AsV v t) = (showId v)++"␣AS␣"++(showId t)

instance Show (UnitCore) where
  show = showInd ""

instance Show (SetCore) where
  show = showInd ""

instance Show (GpromsSD) where
  show (MkSD es s) = (showBlocksInd "" es)++"\n"++(show s)
{- End Function show and showInd instanciations -}

{- Assignment instantiations -}
instance Assignment (InitialCore) where
  asPred (InitEq a b) = Eql a b

instance Assignment (InitialSelectorCore) where
  asPred (InitSelector a b) = Eql (Var a) (Var b)

instance Assignment (SetCore) where
```

```
  asPred (Assign x e) = Eql (Var x) e
{- End Assignment instantiations -}

{- Examples -}

a = INTEGER

e0 = Times (Uminus (Var "b.e")) (OldVar "b.v")

guard = Not (And (Leq (Var "h") (F 0)) (Leq (Var "v") (F 0)))

guard1 = And (Leq (Var "h") (F 0)) (Leq (Var "v") (F 0))

airswitch = SWITCHTO "ground" guard

groundswitch = SWITCHTO "air" guard1

dotv = Var "$v"
g = Var "g"

falling = (Equal dotv (Uminus g))
fallen = (Equal dotv (F 0))

ifeq = (IF guard
        [falling]
        [fallen])

caseq = CASE "ballPossition"
        [wclause0
        ,wclause1
        ]

wclause0 = WHEN "air"
           [falling]
           [airswitch]

wclause1 = WHEN "ground"
           [fallen]
           [groundswitch]

-- Bouncing Ball model

maxInt = 4294967295.0

maxReal = 1e80

bbal = (GpromsSpec
        [(DeclareType "height" 0.0 (-500.0) 500.0)
        ,(DeclareType "number" 0.0 0.0 maxInt)
        ,(DeclareType "timer" 0.0 0.0 maxInt)
        ,(DeclareType "velo" 0.0 (-500.0) maxInt)]
        -- Model declaration
```

```
(Model "Ball"
 -- Parameters
 [(As "e" REAL)
 ,(As "g" REAL )]
 -- Variables
 [(AsV "h" "height")
 ,(AsV "v" "velo")
 ,(AsV "n" "number")
 ,(AsV "t" "timer")]
 -- Selectors
 [(MkSelector "ballPossition" ["air", "ground"]) ] --[(MkSelector "" [])]
 -- Equations
 [(Equal (Var "$h") (Var "v"))
 -- ,(IF (Not (And (Leq (Var "h") (F 0.0)) (Leq (Var "v") (F 0.0))))
 --   [(Equal (Var "$v") (Uminus (Var "g")))]
 --   [(Equal (Var "$v") (F 0.0))]
 --  )
 ,(CASE "ballPosition"
   [(WHEN "air" [(Equal (Var "$v") (Uminus (Var "g")))]
     [(SWITCHTO "ground" (And (Leq (Var "h") (F 0.0)) (Leq (Var "v") (F 0.0))
    )
   ,(WHEN "ground" [(Equal (Var "$v") (F 0.0))]
     [(SWITCHTO "air" (Not (And (Leq (Var "h") (F 0.0)) (Leq (Var "v") (F 0.0
    )
   ]
  )
 ,(Equal (Var "$t") (F 1))
 ]
)
-- End Model declaration
-- Process declararion
(Proc "Bouncing"
 -- Units
 [(AsU "b" "Ball")]
 -- Set
 [(Assign "b.e" (F 0.8))
 ,(Assign "b.g" (F 9.81))
 ]
 -- Assign
 [(Assign "b.n" (F 0))]
 -- Initial Selector
 [(InitSelector "b.ballPosition" "b.air")]
 -- Initial
 [(InitEq (Var "b.h") (F 10))
 ,(InitEq (Var "b.v") (F 0))
 ,(InitEq (Var "b.t") (F 0))
 ]
 -- Schedule
 (PARALLEL
  [(WHILE TRUE (
       SEQUENCE
       [(CONTINUE_UNTIL (LessThan (Var "b.h") (F 0.0)))
```

```
                ,(REINITIAL "b.v" (Equal
                                   (Var "b.v")
                                   (Times (Uminus (Var "b.e")) (OldVar "b.v"))
                                  )
                 )
                ,(REINITIAL "b.h" (Equal (Var "b.h") (F 0.0)))
                ,(RESET "b.n" (Plus (OldVar "b.n") (F 1)))
                ]
                )-- END SEQUENCE
            )-- END WHILE
            ,(SEQUENCE
              [(CONTINUE_UNTIL (Leq (F 12.0) (Var "b.t")))
              ,(STOP)
              ]
             )
            ])-- END PARALLEL
          )-- End Process declaration
        )

sched = (PARALLEL
          [(WHILE TRUE (
               SEQUENCE
               [(CONTINUE_UNTIL (LessThan (Var "b.h") (F 0.0)))
               ,(REINITIAL "b.v" (Equal
                                  (Var "b.v")
                                  (Times (Uminus (Var "b.e")) (OldVar "b.v"))
                                 )
                )
               ,(REINITIAL "b.h" (Equal (Var "b.h") (F 0.0)))
               ,(RESET "b.n" (Plus (OldVar "b.n") (F 1)))
               ]
               )-- END SEQUENCE
           )-- END WHILE
           ,(SEQUENCE
             [(CONTINUE_UNTIL (Leq (F 12.0) (Var "v.t")))
             ,(STOP)
             ]
            )
           ])

-- The replace example
replex = (GpromsSpec
          [(DeclareType "realType" 0.5 (-maxReal) maxReal)]
          -- Model declaration
          (Model "Replace␣Example"
           -- Parameters
           [] -- Nothing to see here
           -- Variables
           [(AsV "x" "realType")
           ,(AsV "y" "realType")
           ,(AsV "t" "realType")]
           -- Selectors
```

```
            [] -- It is lonely in here
            -- Equations
            [(Equal (Var "$t") (F 1.0))
            ,(Equal (Plus (Var "x") (Var "y")) (Var "t"))
            ]
           )
            -- Process declaration
            (Proc "ReplaceExample"
             -- Units
             [(AsU "r" "ReplaceExample")]
             -- Sets
             []
             -- Assign
             [(Assign "r.x" (F 5))]
             -- Initial Selector
             []
             -- Initial
             [(InitEq (Var "r.t") (F 0))]
             -- Schedule
             (SEQUENCE
               [(CONTINUE_UNTIL (Leq (F 3) (Var "r.t")))
               ,(REPLACE "r.x" "r.y" (F 7))
               ,(CONTINUE_UNTIL (Leq (F 6) (Var "r.t")))
               ,STOP
               ]
             )
            )-- End Process declaration
          )
{- End Examples -}

-- To keep listings happy
quoteChar = '"'
```

D Computing the symbolic transition system of a gPROMS composition

This section presents the pseudo-algorithms for computing the symbolic transition system induced by a gPROMS specification. Algorithm 1 computes the symbolic transition system associated to a gPROMS specification. The first step of the algorithm is to compute the action transition relation induced by p. This is done by means of the function gs2ats, defined in Algorithm 2. The states of the STS are those reachable through action transitions, which are contained in the domain and range of $\rightarrow$. Next, the time transition relation is computed for each state in the STS using the time successor function, tsucc, which is defined in Algorithm 14.

Algorithm 2 computes the reachable transition system via an action transition. This transition system is represented through relation R. Note that this algorithm can be generalized to compute any transition system induced by an arbitrary process term. The only requirement is that we know how to compute the action successors of a given state, which is the case for gPROMS compositions, as shown in Algorithm 3.

Algorithm 3 computes the set of successor action transitions. To be able to obtain a compact representation, we have split it intro several auxiliary functions, which are defined in Algorithms 4, and 6.

Algorithm 1 gs2sts(p)

Input: $p \in \mathscr{G}$, a gPROMS composition.
Output: $(Q, \rightarrow, \longmapsto)$ the symbolic transition system associated to p.
$\rightarrow$:= gs2ats(p)
$Q := \mathrm{dom}(\rightarrow) \cup \mathrm{ran}(\rightarrow)$
$\longmapsto := \emptyset$
for all $q \in Q$ **do**
  for all $(f,g) \in \mathrm{tsucc(q)}$ **do**
    $\longmapsto := \longmapsto \cup \{(q,f,g)\}$
  end for
end for

Algorithm 2 gs2ats(p)

Input: $p \in \mathscr{G}$, a gPROMS composition.
Output: $R \in \mathscr{G} \times (\mathscr{P} \times \mathscr{P}) \times \mathscr{G}$, the action transition relation induced by p.
Ensure: $\langle p \rangle \rightarrow^* \langle p' \rangle \xrightarrow{b,r} \langle p'' \rangle$ iff $\langle p' \rangle \rightarrow \langle p'' \rangle \in R$
$R := \emptyset$
$Q := \{q\}$ {Q is the set unvisited vertexes}
$T := \emptyset$ {T is the set of visited vertexes}
while $Q \neq \emptyset$ **do**
  pick an element $q \in Q$
  for all $(b,r,q') \in \mathrm{asucc}(q)$ **do**
    $R := R \cup \{(q,b,r,q')\}$
    $Q := Q \cup (\{q\} \setminus T)$
  end for
  $Q := Q \setminus \{q\}$
  $T := T \cup \{q\}$
end while
return R

No-switch and initialization transitions are computed only for equations or lists of equations.
The time successors of a state are computed as described in Algorithm 14

Algorithm 3 asucc(p)

Input: $p \in \mathscr{G}$, a gPROMS composition.
Output: $S \in \mathscr{P} \times \mathscr{P} \times \mathscr{G}$, the set of action successors of p.
Ensure: $\langle p \rangle \xrightarrow{b,r} \langle p' \rangle$ iff $(b, r, p') \in S$

```
if p is an atomic statement then
  return asuccAs(p)
else if p is a control structure then
  return asuccCs(p)
else if p is a parallel composition then
  return asuccPc(p)
else if p is a sequential composition then
  return asuccSc(p)
else if p is a list of equations then
  return asuccEql(p)
else if p is an equation then
  return asuccEq(q)
else if p is a system description then
  return asuccSd(p)
else if p is a specification then
  return asuccSp(p)
end if
```

Algorithm 4 asuccAs(p)

```
if p is of the form RESET x := e END then
  return {(true, x = old(e), ✓)}
else if p is of the form SWITCH x :=y END then
  return {(true, x = y, ✓)}
else if p is of the form REINITIAL y WITH y = e END then
  return {(true, x = old(e), ✓)}
else if p is STOP then
  return {(true, true, ⊥)}
end if
```

Algorithm 5 asuccCs(p)

```
if p is of the form WHILE b DO s END then
  return {(b, true, SEQUENCE s : p END), (¬b, true, ✓)}
else if p is of the form IF b THEN s0 ELSE s1 END then
  return {(b, true, s0), (¬b, true, s1)}
else if p is of the form CONTINUE UNTIL b ; then
  return {(b, true, ✓)}
end if
```

Algorithm 6 asuccPc(p)

$S := \emptyset$
let $p =$ **PARALLEL** zs **END**
for all i such that $0 \leq i < \#zs$ **do**
  for all $(b,r,x) \in \text{asucc}(zs.i)$ **do**
    if $x \notin \{\bot, \checkmark\}$ **then**
      $S := S \cup \{(b,r,$**PARALLEL** $zs[i:x]$ **END**$)\}$
    else if $x = \checkmark$ **then**
      if $3 \leq \#zs$ **then**
        $S := S \cup \{(b,r,$**PARALLEL** $zs \setminus i$ **END**$)\}$
      else if $\#zs = 2$ **then**
        $S := S \cup \{(b,r,(zs \setminus i).0)\}$
      end if
    else if $x = \bot$ **then**
      $S := S \cup \{(b,r,\bot)\}$
    end if
  end for
end for
return S

Algorithm 7 asuccSc(p)

$S := \emptyset$
let $p =$ **SEQUENCE** $s : zs$ **END**
for all $(b,r,x) \in \text{asucc}(s)$ **do**
  if $x \notin \{\checkmark, \bot\}$ **then**
    $S := S \cup \{(b,r,$**SEQUENCE** $s' : zs$ **END**$)\}$
  else if $x = \checkmark$ **then**
    if $2 \leq \#zs$ **then**
      $S := S \cup \{(b,r,$**SEQUENCE** zs **END**$)\}$
    else if $\#zs = 1$ **then**
      $S := S \cup \{(b,r,zs.0)\}$
    end if
  else if $x = \bot$ **then**
    $S := S \cup \{(b,r,\bot)\}$
  end if
end for
return S

Algorithm 8 asuccEql(p)

if p is of the form $[\,]$ **then**
  return $\{(\text{true}, \text{true}, [\,])\}$
else if p is of the form $e : es$ **then**
  return $\text{asucc}(e) \cup \text{asucc}(es)$
end if

Algorithm 9 asuccEq(p)

$S := \emptyset$
if p is of the form **CASE** v **OF** ws **END then**
  for all (**WHEN** x : es_x ts_x) $\in ws$ **do**
    for all (**SWITCH TO** y **IF** c;) $\in ts_x$ **do**
      let (**WHEN** y : es_y ts_y) $\in ws$
      for all $(c_y, n) \in \text{isucc}(es_y)$ **do**
        $S := S \cup \{(v = x \wedge c \wedge c_y, v = y \wedge n, p)\}$
      end for
    end for
    for all $(b_e, r_e, q) \in \text{asucc}(es)$ **do**
      let $b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts : \neg c \rangle$
      $S := S \cup \{(v = x \wedge b \wedge b_e, r_e, p)\}$
    end for
  end for
end if
return S

Algorithm 10 asuccSd(p)

$S := \emptyset$
let $p = \langle es, S \rangle$
for all $(b, r, es) \in \text{asucc}(es)$ **do**
  $S := S \cup \{(b, r, \langle es', S \rangle)\}$
end for
for all $c \in \text{nssucc}(es)$ **do**
  for all $(b, r, x) \in \text{asucc}(S)$ **do**
    if $x \neq \bot$ **then**
      $S := S \cup \{(c \wedge b, r, \langle es, x \rangle)\}$
    else if $x = \bot$ **then**
      $S := S \cup \{(c \wedge b, r, \bot)\}$
    end if
  end for
end for
return S

Algorithm 11 asuccSp(p)

return asucc(gs2gsd(p))

Algorithm 12 nssucc(p)

$S := \emptyset$
if p is of the form $e_0 = e_1$ **then**
  return {true}
else if p is of the form **CASE** v **OF** ws **END then**
  for all (**WHEN** x : es_x ts_x) $\in ws$ **do**
    let $b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x : \neg c \rangle$
    for all $c_x \in$ nssucc(es_x) **do**
      $S := S \cup \{v = x \wedge b \wedge c_x\}$
    end for
  end for
else if p is of the form $e : es$ **then**
  for all $c \in$ nssucc(e) **do**
    for all $cs_s \in$ nssucc(es) **do**
      $S := S \cup \{c \wedge cs_s\}$
    end for
  end for
end if
return S

Algorithm 13 isucc(p)

$S := \emptyset$
if p is of the form $e_0 = e_1$ **then**
  $S := \{(\text{true}, \text{true})\}$
else if p is of the form **CASE** v **OF** ws **END then**
  for all (**WHEN** x : es_x ts_x) $\in ws$ **do**
    for all (**SWITCH TO** y **IF** c;) $\in ts_x$ **do**
      let (**WHEN** y : es_y ts_y) $\in ws$
      for all $(c_y, n) \in$ isucc(es_y) **do**
        $S := S \cup \{(c \wedge c_y, v = y \wedge n)\}$
      end for
    end for
  end for
else if p is of the form $e : es$ **then**
  for all $(c, n) \in$ isucc(e) **do**
    for all $(c_s, n_s) \in$ isucc(es) **do**
      $S := S \cup \{(c \wedge c_s, n \wedge n_s)\}$
    end for
  end for
end if
return S

Algorithm 14 tsucc(p)

if p is of the form **CONTINUE UNTIL** b ; **then**
  return $\{(\text{true}, \text{closeNeg}(\neg b))\}$
else if p is a parallel composition **then**
  return tsuccPc(p)
else if p is a sequential composition **then**
  return tsuccSc(p)
else if p is a list of equations **then**
  return tsuccEql(p)
else if p is an equation **then**
  return tsuccEq(q)
else if p is a system description **then**
  return tsuccSd(p)
else if p is a specification **then**
  return tsuccSp(p)
else
  return $\emptyset$
end if

Algorithm 15 tsuccPc(p)

let $p =$ **PARALLEL** zs **END**
$S := \emptyset$
if p is of the form $[s,t]$ **then**
  for all $(f_s, g_s) \in \text{tsucc}(s)$ **do**
    for all $(f_t, g_t) \in \text{tsucc}(t)$ **do**
      $S := S \cup \{(f_s \wedge f_t, g_s \wedge g_t)\}$
    end for
  end for
else if p is of the form $s : zs$ and $2 \leq \#zs$ **then**
  for all $(f_s, g_s) \in \text{tsucc}(s)$ **do**
    for all $(f_z, g_z) \in \text{tsucc}($**PARALLEL** zs **END**$)$ **do**
      $S := S \cup \{(f_s \wedge f_z, g_s \wedge g_z)\}$
    end for
  end for
end if
return S

Algorithm 16 tsuccSc(p)

let $p =$ **SEQUENCE** $s : zs$ **END**
return tsucc(s)

Algorithm 17 tsuccEql(p)

$S := \emptyset$
if p is of the form [ ] **then**
  $S := \{\text{true}, \text{true}\}$
else if p is of the form $e : es$ **then**
  for all $(f_e, g_e) \in \text{tsucc}(e)$ **do**
    for all $(f_s, g_s) \in \text{tsucc}(es)$ **do**
      $S := S \cup \{(f_e \wedge f_s, g_e \wedge g_s)\}$
    end for
  end for
end if
return S

Algorithm 18 tsuccEq(p)

$S := \emptyset$
if p is of the form $e_0 = e_1$ **then**
  $S := \{(e_0, e_1)\}$
else if p is of the form **CASE** v **OF** ws **END then**
  for all (**WHEN** x : es_x ts_x) $\in ws$ **do**
    let $b = \langle \bigwedge y, c : (\textbf{SWITCH TO } y \textbf{ IF } c;) \in ts_x : \text{closeNeg}(\neg c) \rangle$
    for all $(f_e, g_e) \in \text{tsucc}(es_x)$ **do**
      $S := S \cup \{(v = x \wedge f_e, v = x \wedge b \wedge g_e)\}$
    end for
  end for
end if
return S

Algorithm 19 tsuccSd(p)

let $p = \langle es, S \rangle$
$S := \emptyset$
for all $(f_e, g_e) \in \text{tsucc}(es)$ **do**
  for all $(f_s, g_s) \in \text{tsucc}(S)$ **do**
    $S := S \cup \{(f_e \wedge f_s, g_e \wedge g_s)\}$
  end for
end for
return S

Algorithm 20 tsuccSp(p)

return tsucc(gs2gsd(p))