Constraint-Based Modeling and Symbolic Simulation of Hybrid Systems with HydLa and HyLaGI

Yunosuke Yamada(✉), Masashi Sato, and Kazunori Ueda

Department of Computer Science and Engineering, Waseda University, 3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
{yunosuke,masashi,ueda}@ueda.info.waseda.ac.jp

Abstract. Hybrid systems are dynamical systems that include both continuous and discrete changes. Modeling and simulation of hybrid systems can be challenging due to various kinds of subtleties of their behavior. The declarative modeling language HydLa aims at concise description of hybrid systems by means of constraints and constraint hierarchies. HyLaGI, a publicly available symbolic simulator of HydLa, featured error-free computation with symbolic parameters. Based on symbolic computation, HyLaGI provides various functionalities including nondeterministic execution, handling of infinitesimal quantities, and construction of hybrid automata. Nondeterministic execution in the framework of constraint programming enables us to solve inverse problems by automatic parameter search. This paper introduces these features by means of example programs. This paper also discusses our experiences with HydLa programming, which is unique in that its data and control structures are both based on constraint technologies. We discuss its expressive power and our experiences with modeling using constraint hierarchies.

Keywords: Hybrid systems · Constraints · Symbolic simulation

1 Introduction

Hybrid systems [12] are dynamical systems which include both continuous and discrete changes. To put it differently, hybrid systems are dynamical systems whose description involves case analysis. Because of the case analysis, simulation of hybrid systems can easily go qualitatively wrong, and techniques for rigorous simulation are very important.

Modeling of hybrid systems, as opposed to continuous systems or discrete systems, is itself a challenge. The best-known modeling technique is hybrid automata [9] with an explicit notion of states, but designing fundamental language constructs, especially those for *declarative* (as opposed to procedural) modeling seems to be an open problem. Although there have been a number of proposals of modeling languages (see [5] for a comprehensive survey), most high-level languages aim for the modeling of complex hybrid systems [1,16], leaving the quest for fundamental modeling constructs rather unexplored.

R. Chamberlain et al. (Eds.): CyPhy 2019/WESE 2019, LNCS 11971, pp. 153–178, 2020.
https://doi.org/10.1007/978-3-030-41131-2_8

We take a *constraint-based* approach to the above two questions, i.e., rigorous simulation and modeling constructs. Constraints are a *necessary* ingredient in any modeling technique of hybrid systems in that they all handle differential equations. However, virtually all high-level modeling languages come with other language constructs to provide the language with control structures. For instance, Modelica [16] appears to be close to our goal in that its main feature is *non-causal*, constraint-based modeling, but Modelica also supports imperative constructs to simulate models for which explicit sequencing of events is necessary. Another high-level language, Zélus [4], builds on the framework of synchronous programming into which ordinary differential equations (ODEs) were integrated. Accordingly, the research question we are going to address is:

"Are constraints and constraint solving adequate, by themselves, for the concise modeling and rigorous simulation of hybrid systems?"

The modeling language HydLa [20,21] and its implementation HyLaGI [13, 14] were built as an attempt to answer that question.

Constraint programming for hybrid systems is not new; for example, Hybrid CC [8] was born as an extension of concurrent constraint programming. While Hybrid CC retained the flavor of process calculi, HydLa, inspired by Hybrid CC, adopted *constraint hierarchy* [3] for concise modeling of hybrid systems, as will be exemplified soon.

The constraint-based approach has another advantage—the ability to express *partial information* and handle it with *rigorous symbolic computation* based on consistency checking. Constraints include the notion of intervals such as $x \in [1.0, 3.5]$. As an important application, they also allow natural handling of *parametric* hybrid systems, i.e., hybrid systems with symbolic parameters, which is useful for the understanding, analysis and design of hybrid systems. Some verification tools such as KeYmaera X [7] and dReach [10] also took a rigorous, symbolic approach. Unlike these and like Acumen [18], HyLaGI was designed as a simulation tool whose primary goal was to help understanding of hybrid systems (as opposed to the solving of decision problems). There are other tools for rigorous simulation. For instance, Acumen [18] and Flow* [6] adopt (numerical) interval enclosure techniques while we take a symbolic approach to handle parametric systems. Another symbolic simulator was reported in [17], but unlike it our algorithm provides exhaustive search.

1.1 HydLa by Example

Let us introduce HydLa by a simple example.

Figure 1 shows a HydLa model of a bouncing particle. In HydLa, each variable is treated as a function of time; for example, a variable `y` is an abbreviation of a function $\mathtt{y}(t)$ $(t \geq 0)$ and represents the height of the particle, while `y'` and `y''` stand for its speed and acceleration, both being functions of time.

The first three lines are the definitions of named constraints (called *constraint modules* or simply *modules*) represented using differential equations and logical

```
1 INIT   <=> 7 < y < 12 & y' = 0.
2 FALL   <=> [](y'' = -10).
3 BOUNCE <=> [](y- = 0 => y' = -4/5 * y'-).
4
5 INIT, (FALL << BOUNCE).
6 //#hylagi -p10
```

Fig. 1. A bouncing particle model in HydLa with an uncertain initial state.

connectives. `INIT` stands for a constraint defining the (uncertain) initial position and the speed of the particle. `FALL` represents free fall, while `BOUNCE` represents bouncing. The temporal logic operator `[]`, called "always", indicates that a constraint holds and keeps holding after it is generated. The postfix operator `-` indicates the left limit of the value of the variable; for example, `y-`(t) stands for $\lim_{t' \to t-0} \mathtt{y}(t')$. The connective `=>` is logical implication. Line 5 declares how the three modules are composed. We can declare relative strength of modules: in our case, `FALL` is declared to be weaker than `BOUNCE` and is ignored when it is inconsistent with `BOUNCE`. Line 6 is a comment line showing default options given to HyLaGI (Sect. 3). Further details of HydLa will be described in Sect. 2.

1.2 HyLaGI and WebHydLa

We are developing an implementation HyLaGI to simulate HydLa programs. HyLaGI, available from GitHub[1], is implemented in C++ and uses the Boost library. We currently use Mathematica as a constraint solver and perform simulations by symbolic computation. This opens up various applications including the simulation and reasoning about models with symbolic parameters, handling of infinitesimal quantities, and checking of the inclusion properties of the sets of trajectories. Symbolic simulation assumes the existence of closed-form solutions of ODEs, which might sound like a rather strong restriction, but ODEs without closed-form solutions could be rigorously approximated using a family of ODEs with symbolic parameters (to enclose approximation errors) that have closed-form solutions, which is among our future work.

Figure 2 shows the output of HyLaGI from the program of Fig. 1. HyLaGI simulates a program in phases, which are an alternating sequence of point phases (PPs) and interval phases (IPs). A point phase represents discrete change, and an interval phase represents continuous change. HyLaGI represents the uncertain initial condition of `y` by generating a symbolic parameter `p[y,0,1]`, meaning a parameter representing the 0th derivative of `y` of the first phase. HyLaGI performs case analysis for uncertain models, but for this example it returns only one case with 10 phases. Information for a point phase includes the time and the values of variables, while that of an interval phase includes the time interval and trajectories (as functions of time) over that interval. In addition, it provides

[1] https://github.com/HydLa/.

```
------ Result of Simulation ------
---------parameter condition(global)---------
p[y, 0, 1] : (7, 12)
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
y : p[y, 0, 1]
y' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->5^(-1/2)*p[y, 0, 1]^(1/2)
y : t^2*(-5)+p[y, 0, 1]
y' : t*(-10)
y'' : -10
---------2---------
---------PP 3---------
unadopted modules: {FALL}
unsat modules : {BOUNCE, FALL}
unsat constraints : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-
negative  :
t : 5^(-1/2)*p[y, 0, 1]^(1/2)
y : 0
y' : 5^(-1/2)*8*p[y, 0, 1]^(1/2)
---------IP 4---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
t : 5^(-1/2)*p[y, 0, 1]^(1/2)->5^(-1/2)*p[y, 0, 1]^(1/2)*13/5
y : t^2*(-5)+18*5^(-1/2)*t*p[y, 0, 1]^(1/2)+p[y, 0, 1]*(-13)/5
y' : t*(-10)+18*5^(-1/2)*p[y, 0, 1]^(1/2)
y'' : -10

... (omitted up to PP 9) ...

---------IP 10---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
t : 5^(-1/2)*p[y, 0, 1]^(1/2)*613/125->5^(-1/2)*p[y, 0, 1]^(1/2)
    *3577/625
y : t^2*(-5)+5^(-1/2)*t*p[y, 0, 1]^(1/2)*6642/125+p[y, 0,
    1]*(-2192701)/78125
y' : t*(-10)+5^(-1/2)*p[y, 0, 1]^(1/2)*6642/125
y'' : -10
---------parameter condition(Case1)---------
p[y, 0, 1] : (7, 12)
# number of phases reached limit
```

Fig. 2. Simulation results of the bouncing particle up to 10 phases.

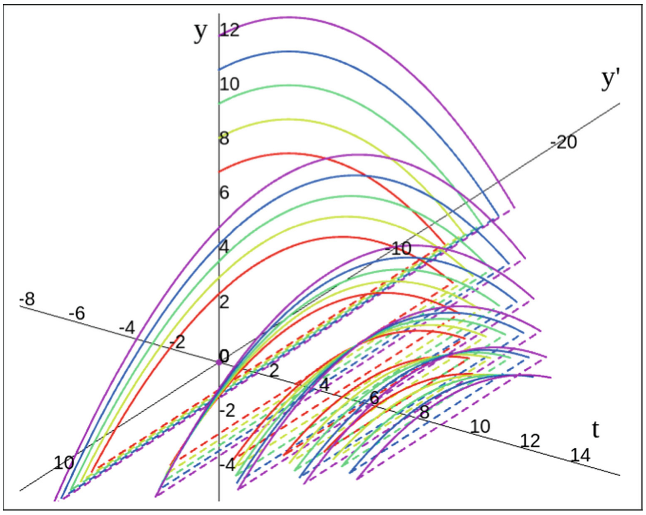

Fig. 3. Output of webHydLa for the bouncing particle model. Note that simulation was executed only once, after which the family of trajectories were rendered by the visualizer.

information about the constraints that determined these values or trajectories, which turned out to be extremely useful in debugging and construction of hybrid automata, as discussed in Sects. 3 and 4.

The output of Fig. 2 suggests that a visualization tool for the understanding of results is highly desirable. We have developed webHydLa[2] as an IDE for HydLa that can visualize simulation results in 2D and 3D. For example, the simulation result of the program in Fig. 1 is visualized as in Fig. 3.

1.3 Purpose and Outline of the Paper

HydLa and HyLaGI has been available for quite some time, but except for the language definition [20] and implementation techniques [13,14], the consequences and implications of the design and functionalities of HydLa and HyLaGI in the light of the modeling of various hybrid systems have not been reported. Reports on various ideas that went into our system were scattered over rather short papers (some of which in Japanese). Thus the purpose of the present paper is to report our constraint-based approach in a comprehensive way with a number of examples, discussing important details and findings not addressed by previous papers.

The rest of the paper is organized as follows. Section 2 briefly introduces the constructs of HydLa. Section 3 introduces functionalities of HyLaGI by various examples. Section 4 describes our experiences with constraint-based modeling

[2] http://webhydla.ueda.info.waseda.ac.jp/.

with HydLa. Section 5 introduces further examples involving parameter search. Section 6 concludes the paper.

2 The Constraint-Based Language HydLa

We briefly overview the modeling language HydLa. Please refer to [20] for further details of basic constructs and [13] for extended features implemented in HyLaGI and some subtle points.

As exemplified by the bouncing particle model of Fig. 1, a typical HydLa program consists of the definitions of constraints followed by the declaration of constraint hierarchy formed by the defined modules. Constraint hierarchy refers to a partially ordered set whose elements are combinations of modules allowed by the declaration and whose order is a set inclusion relation. Constraint hierarchy allows us to represent *ordinary or default* behavior and *special or exceptional* behavior in a concise manner. Constraints in the module `INIT` are defined without the `[]` operator and hold only at time 0. A constraint with `=>` expresses a conditional constraint (also called a *guarded* constraint) whose consequent is enabled only when the antecedent (called a *guard*) holds. The constraint hierarchy declared in Line 5 indicates that `BOUNCE` is stronger than `FALL` and also that `BOUNCE` and `INIT` have the highest priority. At each point of time, HydLa adopts a maximal consistent set (MCS) of modules that respects constraint hierarchy. In this example, while the particle is floating, the set {`INIT`, `FALL`, `BOUNCE`} is adopted (note that `INIT` is vacuously satisfied after time 0, and `BOUNCE` is vacuously satisfied because of the false guard) and that when it collides with the floor, the MCS changes to {`INIT`, `BOUNCE`}. Note that a module which is not weaker than any other module in the constraint hierarchy is called a *required* module and is always enabled.

2.1 Syntax

The syntax of HydLa is shown in Fig. 4, where *dname*, *cname*, *vname* are symbolic names representing definitions, constraints and variables, respectively.

Here we describe language features not covered by Fig. 1. The definition *Def* says that we can define named declarations (that may include constraint hierarchies) as well as named constraints. It also says that definitions may have formal parameters $\overrightarrow{X}$. The syntax of a constraint C allows an *always* ($\Box$) constraint to occur in the consequent of a guarded constraint. Examples of its use will be shown in Sect. 5. Note that, in the declaration *Decl*, the operator "$\ll$" binds tighter than the operator "," that imposes no relative priority. For example, $A \ll B, C$ is equal to $(A \ll B), C$.

Table 1 shows the correspondence between the abstract syntax of Fig. 4 and the concrete syntax used in example programs.

$$
\begin{array}{rll}
\text{(HydLa program)} & P & ::= (\mathit{Def} \mid \mathit{Decl})^* \\
\text{(definition)} & \mathit{Def} & ::= \mathit{dname}(\vec{X})\{\mathit{Decl}\} \mid \mathit{cname}(\vec{X}) \Leftrightarrow C \\
\text{(constraint)} & C & ::= A \mid C \wedge C \mid G \Rightarrow C \mid \exists \mathit{vname}.C \mid \Box C \mid \mathit{cname}(\vec{E}) \\
\text{(guard)} & G & ::= A \mid G \wedge G \mid G \vee G \mid \neg G \\
\text{(atomic constraint)} & A & ::= E\ \mathit{Rop}\ E \\
\text{(relational operator)} & \mathit{Rop} & ::= = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
\text{(expression)} & E & ::= E\ \mathit{Aop}\ E \mid \mathit{Prev} \mid \mathit{constant} \\
\text{(arithmetic operator)} & \mathit{Aop} & ::= + \mid - \mid \times \mid \div \mid \hat{} \\
\text{(previous)} & \mathit{Prev} & ::= D \mid D- \\
\text{(derivative)} & D & ::= \mathit{vname} \mid D' \\
\text{(declaration)} & \mathit{Decl} & ::= M \mid \mathit{Decl}, \mathit{Decl} \mid \mathit{Decl} \ll \mathit{Decl} \\
\text{(module)} & M & ::= C \mid \mathit{dname}(\vec{E})
\end{array}
$$

Fig. 4. Syntax of HydLa.

Table 1. Correspondence between abstract and concrete syntax.

Abstract	Concrete	Abstract	Concrete	Abstract	Concrete	Abstract	Concrete
$\ll$	`<<`	$\leq$	`<=`	$\vee$	`\/` or `|`	$\exists$	`\`
$\Leftrightarrow$	`<=>`	$\neq$	`!=`	$\wedge$	`/\` or `&`		
$\geq$	`>=`	$\neg$	`!`	$\Box$	`[]`		

List Expressions. We often need to generate multiple objects (such as balls and cars) with the same property in the modeling of hybrid systems. As an extension of the syntax in Fig. 4, HydLa provides a list notation to simplify the description of such models. Here we explain their use by examples, leaving the full syntax with list expressions to the Appendix.

We introduce two types of list notation. The first type is the list of arithmetic expressions, which can be written extensionally or in a list comprehension notation. Range expressions of the form $\{l\,..\,h\}$ are also allowed. Range expressions have two applications; one is to express a list of consecutive values (such as {`2*3+1..10`}) and the other is to express a list of variables whose names end with consecutive digits. For instance, {`x0..x4`} stands for the list of variables {`x0, x1, x2, x3, x4`}. The second type of list notation is to declare multiple instances of constraints. A list of priority declarations can also be written extensionally or in a list comprehension notation as in the example below.

For example, consider a road congestion model with five cars, of which the cars except the first one accelerates and deaccelerates depending on the distance from the car in front (Fig. 5). Figure 6 shows its HydLa model. Line 1 defines `X` to be the list of cars for which the notation `X[`i`]` is available to access its ith element. Lines 3–6 define named constraints describing the properties of the cars. Lines 8–12 declare constraints imposed by the five cars, where `|X|` represents

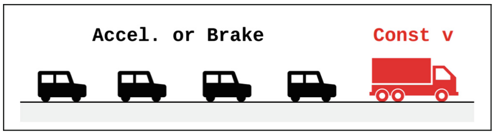

Fig. 5. A road congestion model.

```
X := {x1..x5}.

INIT(x,x0,v0) <=> x = x0 & x' = v0.
CONST(x)      <=> [](x'' = 0).
BRAKE(x,xf)   <=> [](x'- >  0 & xf- - x- < 30 => x'' = -5).
ACC(x, xf)    <=> [](x'- < 15 & xf- - x- > 50 => x'' =  3).

{ INIT(X[i],100*i+i,4) | i in {1..|X|-1} }.
INIT(X[|X|],100*|X|,8).
{ CONST(X[i]) << (ACC(X[i],X[i+1]), BRAKE(X[i],X[i+1]))
                                         | i in {1..|X|-1} }.
CONST(X[|X|]).
//#hylagi -p40
```

Fig. 6. A road congestion model in HydLa.

the cardinality of the list X. When we run this program, the distance between two cars is kept neither too close nor too distant as shown in Fig. 7.

Existential Quantifier. HydLa features *existential quantifiers* to generate variables dynamically.

Constraints with existential quantifiers are typically written in the consequents of guarded constraints and generate new trajectories when the guards hold. Quantified variables are given fresh names when the constraints containing those variables are expanded. In the modeling of HydLa, dynamic variables are often used as temporary variables that propagate constraints.

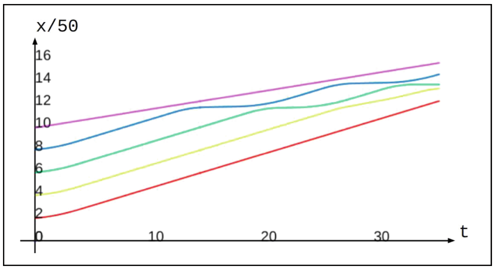

Fig. 7. Simulation result of the road congestion model in HydLa.

```
INIT      <=> p = 65 & mode = 0.
OFF       <=> [](mode = 0 => p' = -2).
ON        <=> [](mode = 1 => p' = 1).
MODE(l,r,m) <=> \x.(l < x < r & [](x' = -1)
                    & [](x- = 0 => mode = m)).
SWITCHOFF <=> [](p- = 68 & mode = 1 => MODE(0.2,0.5,0)).
SWITCHON  <=> [](p- = 62 & mode = 0 => MODE(0.2,0.5,1)).

INIT, [](mode' = 0) << (SWITCHON, SWITCHOFF).
OFF, ON.
```

Fig. 8. A thermostat model with delay in HydLa.

```
FACTORIAL(n, ans)
        <=> (n = 0 => ans = 1)
          & (n > 0 => \x.(ans = n * x & FACTORIAL(n-1, x))).
CALC_F  <=> [](timer- = 1 => FACTORIAL(5, ans)).
TIMER   <=> timer = 0 & [](timer' = 1).

TIMER, CALC_F.
```

Fig. 9. A model to calculate the factorial of 5.

Figure 8 is an thermostat model using an existential quantifier. The variable p represents the temperature. SWITCHON and SWITCHOFF are fired when the temperature reaches 62 or 68°, respectively, to switch the mode that decides the differential equation of p. The existentially quantified variables, written with \ instead of ∃, are used in the consequents as local timers to express the delay of mode change. Whenever p reaches 62 or 68°, a new instance of x is generated, is initialized to $[0.2, 0.5]$, decreases linearly, and changes the mode when it reaches 0.

A recursive constraint is another important use of existential quantifiers. Figure 9 is a somewhat contrived example to calculate the factorial of 5 at time 1. The module FACTORIAL consists of two guarded constraints, the base case and the recursive case. When the guard of CALC_F holds, FACTORIAL is expanded and its second guarded constraint is enabled. Then FACTORIAL is expanded recursively until the second argument reaches 0. Each time FACTORIAL is expanded, a fresh intermediate variable is created, and a network of constraints is constructed to propagate the calculation result. In this way, existential quantifiers for dynamic variable creation provide us with an alternative technique to superdense time for the modeling of multi-step instantaneous computation.

2.2 Semantics: Overview

In constraint-based languages, the natural plan for the study of the semantics would be to consider what a program represents (declarative semantics) first and then its computational aspects.

Declarative Semantics. The declarative semantics of a HydLa model is the set of trajectories allowed by the constraints given in the model, where HydLa takes maximal consistent sets of modules at each point of time, as stated in the beginning of Sect. 2 with an example. Here we describe some important aspects of the semantics.

Firstly, HydLa naturally allows models with uncertainties. This comes from the fact that (i) a set of constraints may have multiple solutions, most typically due to initial values given as intervals, and that (ii) a maximal consistent set of modules may not be uniquely determined. It is important to note that, in hybrid systems, quantitative uncertainties may result in qualitative uncertainties. For example, when a particle bounces on a floor with a hole, whether or not the particle eventually enters the hole and how many times it bounces before it enters the hole depend on the initial position and velocity of the ball. HyLaGI described in Sect. 3 is able to compute all possible solutions by case splitting.

Multiple solutions may occur even without parametric uncertainties. For instance, Fig. 10 is a program with a nondeterministic switch that may take the value 0 or 1 every time the value of `timer` reaches 1. Note that `ON` and `STAY` are given the second-to-highest priority because modules with the highest priority are *required* modules.

```
INIT  <=> switch = 0 & timer = 0.
CONST <=> [](switch' = 0).
TIMER <=> [](timer' = 1).
ON    <=> [](timer- = 1 => switch = 1 & timer = 0).
STAY  <=> [](timer- = 1 => switch = 0 & timer = 0).
TRUE  <=> [](1 = 1).

INIT, (CONST, TIMER) << (ON, STAY) << TRUE.
//#hylagi --fnd -p6
```

Fig. 10. A model with a nondeterministic switch.

Secondly, the constraints explicitly given in programs are usually not enough to determine solution trajectories. For instance, in the bouncing particle model of Fig. 1, we are implicitly assuming a *frame axiom* that the position of the ball is continuous except when discontinuity is deduced from explicitly given constraints; otherwise we cannot conclude that the particle starts to move from the floor after bouncing. We call it the *principle of implicit continuity*, and refer the readers to [13] for the details of how it is built into HydLa's constraint framework.

Expressive Power and Computable Trajectories. The syntax of HydLa allows a model `[](x + y = 0)`, which might make sense as a specification but not as an "executable" program. Indeed, no hybrid automaton corresponding to this model is likely to exist. It is therefore meaningful to consider what HydLa models (or programs) are executable. We propose that an *executable* program is a program whose set of trajectories can be represented in *explicit* form defined as follows, where we assume that t stands for the current time:

Definition 1. A trajectory of variables $x_1, \ldots, x_n$ is in *explicit* form if it is piecewisely represented as a (finite or inifinite) set of equations $x_1 = E_{i1}, \ldots, x_n = E_{in}$ associated with a time interval T_i $(i = 1, 2, \ldots)$ during which the above set of equations is effective. Each E_{ij} is a continuous function of t on the interval T_i. E_{ij} may also contain symbolic parameters $p_1, \ldots, p_m (m \geq 0)$ but not $x_1, \ldots, x_n$. The ends of each time interval T_i are also given using expressions that may contain $p_1, \ldots, p_m$. The set of allowed values of the parameters $p_1, \ldots, p_m$ are given as constraints (including equations and inequations), but these constraints must not contain t. The T_i's must be mutually disjoint, and $\bigcup_i T_i$ must be a single interval starting from time 0.

The purpose of simulation is to convert the constraints imposed by a HydLa program into this explicit form, whose example can be found in the simulation result of Fig. 2.

3 HyLaGI: A Symbolic Implementation of HydLa

HyLaGI is an implementation of HydLa that features rigorous simulation of possibly uncertain hybrid systems. The central technique to achieve this is symbolic constraint satisfaction. HyLaGI also employs interval computation internally in order to compute the time of the earliest possible discrete changes efficiently.

The nondeterministic simulation algorithm of HyLaGI repeats point phases (PPs) and interval phases (IPs) alternately until a termination condition (time limit or the number of phases) is satisfied. Calculation of IPs involves (i) solution of possibly parameterized ODEs and (ii) calculation of the time of the next discrete change as a minimization problem. Uncertainties represented by symbolic parameters may result in qualitative difference of trajectories as discussed in Sect. 2.2. In that event, HyLaGI automatically performs case analysis, narrowing the range of parameter values into each qualitatively different case. This symbolic case analysis is supported by quantifier elimination of the constraint solver. The readers are referred to [13] for the detailed simulation algorithm of HyLaGI.

The rest of this section will explain three key functionalities of HyLaGI enabled by the symbolic approach.

3.1 Assertion

HyLaGI provides an `ASSERT` construct using constraints, which can be used for bounded model checking of reachability properties. A property can be stated by

`ASSERT`(G), where G stands for a guard. The declarative meaning of `ASSERT`(G) is `[]G` or `[](!G => false)`, but we provide `ASSERT` as a separate construct to be able to distinguish verification conditions from model descriptions. `ASSERT`(G) stops simulation of the current branch of nondeterministic simulation if G becomes false. Assertion can be used not only for verification but also for solving inverse problems, as will be described in Sect. 5.

3.2 Epsilon Mode

The simulation of hybrid systems, say those modeling physical phenomena, may fall into a situation not considered by textbook laws of physics. For example,

1. a ball bouncing inside a box hits the wall and the floor at the same time,
2. a ball in contact with another ball is hit by the third ball, and
3. force is continually applied to an object in contact with another object to move both.

As for the first example, even if the simultaneity may happen with zero probability in reality, a *family* of trajectories of uncertain hybrid systems may well include it. One way of handling that situation is to consider the limit of situations where the ball hits the wall or the floor slightly earlier. The second and the third examples could also be considered as the limit of the situations where the two objects are slightly apart. HyLaGI is able to simulate such models by taking the limit of 'normal' situations, and it is called the *epsilon mode* [22].

In the epsilon mode (specified by the option "`-e`n"), we can use a variable `eps` as an infinitesimal parameter as shown in Fig. 12. Here, n specifies the highest-order terms to be retained for `eps`, for which 1 is enough except when higher-order effects of `eps` need to be considered. In the epsilon mode, after the maximal consistent set of constraints and the current values of variables are computed in each phase, higher-order terms of `eps` are deleted (after performing Taylor expansion when necessary). When the simulation of all phases are completed, HyLaGI takes the limit (w.r.t. `eps`) of the expressions representing the trajectories of all phases.

For the example of three-body collision, HyLaGI will report "unsatisfiable constraints" (Sect. 4) at the time of collision because the law of two-body collision is not prepared for this situation. However, simulation can be performed if the two touching balls are slightly parted. For example, three-body collision shown in

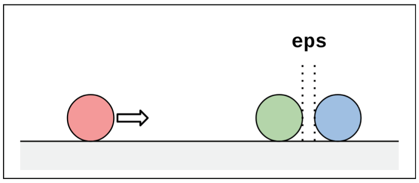

Fig. 11. Collision of three bodies.

```
INIT <=> x1 = 0 & x2 = 5 & x3 = 6+eps
         & x1' = 1 & x2' = 0 & x3' = 0.
EPS  <=> 0 < eps < 0.1 & [](eps' = 0).
CONST(x) <=> [](x'' = 0).
COLLISION(xa, xb) <=>
   [](xa- = xb- - 1 => xa' = xb'- & xb' = xa'-).

INIT, EPS.
(CONST(x1),CONST(x2),CONST(x3))
   << (COLLISION(x1,x2), COLLISION(x2,x3)).
//#hylagi --fnd -p6 -e1
```

Fig. 12. HydLa model of three-body collision.

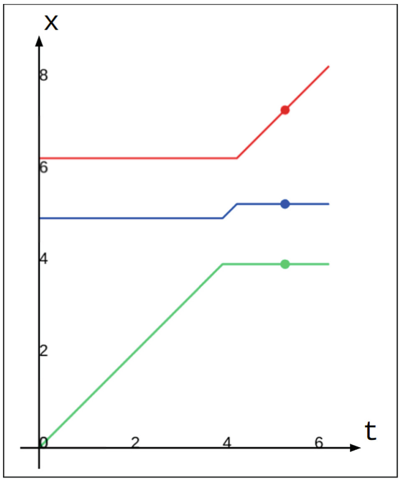

Fig. 13. Simulation result of Fig. 12.

Fig. 11 can be described as a HydLa program in Fig. 12. Three balls of diameter 1 are aligned in a straight line, where x2 and x3 are apart by eps, and x1 moves towards x2 at speed 1. The simulation result of the program of Fig. 12, which still retains eps, is shown in Fig. 13, where the horizontal axis represents time and the vertical axis represents the position x. When the value of eps is not too small, we can see from Fig. 13 that there are two collisions. The text output of the same simulation tells us that x2 will not move in the limit.

Lee et al. discussed the same model in detail in [11] (Fig. 8, p. 806) as a motivating example of their constructive modeling. Their (non-symbolic) approach introduces superdense time to handle simultaneous collisions, while we adopt functions of standard, real-valued time to represent trajectories and handle simultaneity by symbolic perturbation.

There are various variations of the three-body collision. Consider another three-ball model in which the central ball is hit from both sides simultaneously,

```
 1 INIT <=> x1 = 0 & x2 = 5 & x3 = 10+eps
 2          & x1' = 1 & x2' = 0 & x3' = -1.
 3 EPS  <=> -0.1 < eps < 0.1 & [](eps' = 0).
 4 MASS <=> [](m1 = 0.2 & m2 = 1 & m3 = 5).
 5 CONST(x) <=> [](x'' = 0).
 6 COLLISION(xa,ma,xb,mb) <=>
 7     [](xa- = xb- - 1 =>
 8        xa' = (xa'- *(ma-mb) + 2*mb*xb'-)/(ma+mb)
 9      & xb' = (xb'- *(mb-ma) + 2*ma*xa'-)/(ma+mb)).
10
11 INIT. EPS. MASS.
12 (CONST(x1),CONST(x2),CONST(x3))
13    << (COLLISION(x1,m1,x2,m2), COLLISION(x2,m2,x3,m3)).
14 //#hylagi --fnd -p12 -e1
```

Fig. 14. Collision of three bodies with different masses.

a problem discussed also by Lee et al. in [11] (Fig. 11, p. 807). Suppose the balls have mass as shown in Line 4 of Fig. 14. For this problem, the result differs depending on whether the value of `eps` is positive or negative, as shown in Fig. 15 (look at the trajectory of the central ball). Actually, the right-hand limit and left-hand limit do not coincide, and HyLaGI's automatic case analysis generates three cases depending on the sign of `eps` including the case of `eps` $= 0$ that gets stuck.

The Dirac delta function can also be represented using the epsilon mode. The (shifted) delta function can be considered as the limit $\lim_{\varepsilon\to+0}$ of a function whose value is $1/$`eps` in a certain interval of width `eps` and 0 elsewhere as shown in Fig. 16. The function was used successfully for the simulation of impulse force in mechanics and impulse response of electrical circuits.

Another application of infinitesimal parameters is the simulation of analysis of hybrid systems that cause numerous discrete changes in a finite period of time. Although not integrated into the main branch of HyLaGI due to its experimental nature, the work reported in [2] analyzed the symbolic output of HyLaGI to recognize chattering behavior, including a physical model in [11] (p. 808), by the analysis of loop invariants.

Finally, we note that future applications of the epsilon mode is expected to include the handling of robustness and sensitivity at a symbolic level.

3.3 Hybrid Automaton Mode

HyLaGI performs symbolic simulation for a given number of phases or a given period of simulation time. However, we often see cases where different point phases or different interval phases are 'similar' to each other (as in the bouncing particle example) in the sense that they differ only in the values or trajectories of variables. Given that HyLaGI maintains the values of variables as constraints, we can check if the system's state of some phase is subsumed by the system's

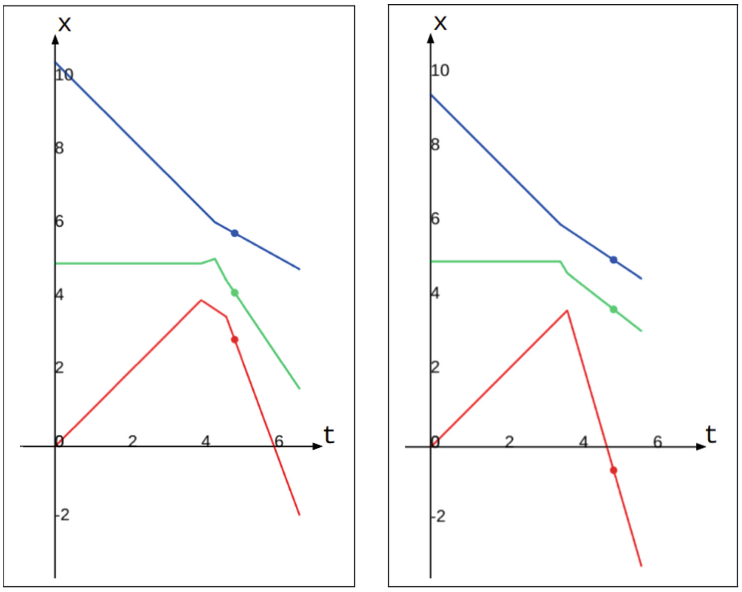

Fig. 15. Simulation result of Fig. 14.

```
TIMER <=> timer = 0 & [](timer’ = 1).
EPS   <=> 0 < eps < 0.1 & [](eps’ = 0).
OFF   <=> [](v = 0).
ON    <=> []((1 < timer < 1+eps) => v = 1/eps).

TIMER, EPS, (OFF << ON).
//#hylagi -e1
```

Fig. 16. HydLa model of an impulse function.

state of one of the previous phases, where a *state* can be defined to consist of (i) the values (or trajectories in the case of interval phases) of variables and (ii) the set of adopted modules. These two also determine (iii) whether each guard of the adopted guarded constraints holds or not. The checking of state subsumption can be done by constraint entailment checking, which can be translated into inconsistency checking by the relation $(P \Rightarrow Q) \equiv \neg(P \wedge \neg Q)$, and we can construct a possibly finite phase transition diagram representing infinite phase transitions. This feature has been implemented in HyLaGI as an optional *hybrid automaton mode*, an experimental mode for future work towards optimized simulation and unbounded model checking. We can see that the items displayed in each phase of HyLaGI’s simulation result (Fig. 2) are sufficient to represent the current state of the model. An initial report on detailed algorithms for constructing hybrid automata can be found in [19], which discusses various subtleties in the construction. Note that we must properly parameterize the initial values of

```
INIT   <=> y > 0.
FALL   <=> [](y'' = -10).
BOUNCE <=> [](y- = 0 => y' = -4/5 * y'-).

INIT, FALL << BOUNCE.
//#hylagi --fha
```

Fig. 17. A bouncing particle model with parameterized initial height.

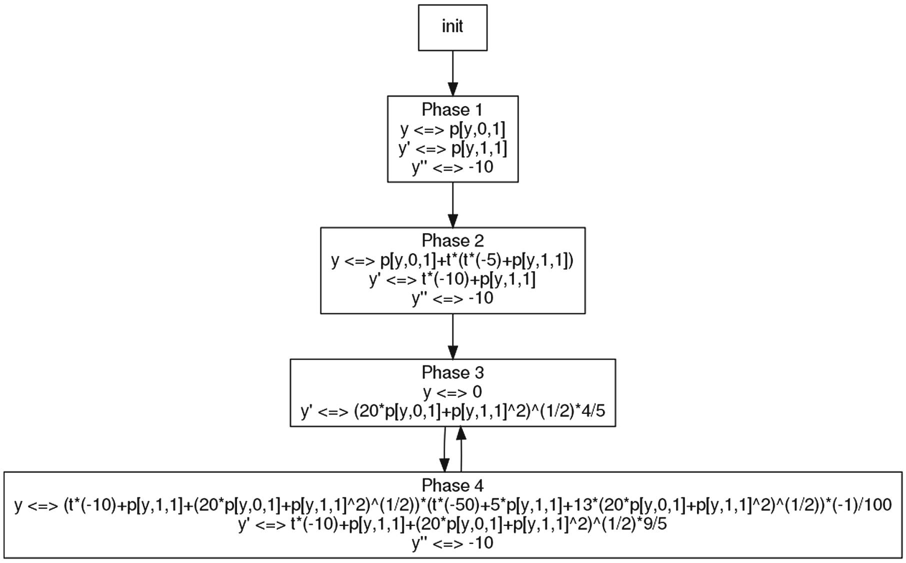
init
Phase 1
y <=> p[y,0,1]
y' <=> p[y,1,1]
y'' <=> -10
Phase 2
y <=> p[y,0,1]+t*(t*(-5)+p[y,1,1])
y' <=> t*(-10)+p[y,1,1]
y'' <=> -10
Phase 3
y <=> 0
y' <=> (20*p[y,0,1]+p[y,1,1]^2)^(1/2)*4/5
Phase 4
y <=> (t*(-10)+p[y,1,1]+(20*p[y,0,1]+p[y,1,1]^2)^(1/2))*(t*(-50)+5*p[y,1,1]+13*(20*p[y,0,1]+p[y,1,1]^2)^(1/2))*(-1)/100
y' <=> t*(-10)+p[y,1,1]+(20*p[y,0,1]+p[y,1,1]^2)^(1/2)*9/5
y'' <=> -10

Fig. 18. State transition graph induced by HyLaGI from a HydLa model.

the variables in order to construct hybrid automata. For example, for a bouncing particle on the floor, we fully parameterize the initial height as shown in Fig. 17. In the hybrid automaton mode, the results can be obtained in the Graphviz format. Figure 18 shows the state transition graph obtained from the program in Fig. 17, where the odd-numbered phases represent discrete changes and the even-numbered phases represent continuous evolution.

4 Experiences with Constraint-Based Modeling

4.1 Discrete Asks and Continuous Asks

Guarded constraints in HydLa are one of the two main constructs that provides the language with control structure (the other construct being constraint hierarchy), corresponding to conditionals in other languages. From our experience with HydLa programming, we have learned that guarded constraints used to describe

```
INIT  <=> p = 65 & [](k1 = 1) & [](k2 = 2) & on = 0.
CONST <=> [](on' = 0).
ON    <=> [](on = 1 => p' =  k1).
OFF   <=> [](on = 0 => p' = -k2).
SWITCHON  <=> [](p- = 62 & on- = 0 => on = 1).
SWITCHOFF <=> [](p- = 68 & on- = 1 => on = 0).

INIT, ON, OFF, CONST << (SWITCHON, SWITCHOFF).
//#hylagi -p10
```

Fig. 19. A thermostat model in HydLa.

practical hybrid systems are categorized into two patterns. We call them *discrete ask* and *continuous ask*, after the terminology in concurrent constraint programming. *Discrete ask* is a guarded constraint which is enabled at isolated time points and triggers discrete changes. An example is `BOUNCE` in Fig. 1. Since discrete ask cancels a differential constraint at certain time points, it is usually given a higher priority than differential constraints. Continuous ask is a constraint whose guard continues to hold for a certain period of time during which the model makes continuous change according to the enabled consequent of the constraint. A thermostat model in Fig. 19 contains an example of continuous ask. The variable `on` represents the state of the thermostat whose value is discretely changed when the temperature `p` is about to exceed a certain threshold. The modules `ON` and `OFF` represent continuous asks; they refer to the variable `on` and have differential equations on the temperature in the consequents. The value of `on` is changed by `SWITCHON` and `SWITCHOFF` which are discrete asks. This example shows a design pattern in which a variable, called a state variable, can be used to represent the discrete state of a system and is referred to from the guards of continuous ask. In our experience, it is a good practice to write HydLa programs keeping the different roles of discrete ask and continuous ask in mind.

Note that HyLaGI does not allow existential quantifiers in the consequent of a continuous ask because such a consequent would generate an infinite number of variables and constraints. HyLaGI does not handle such cases and stops simulation.

4.2 Common Mistakes in Modeling

The design principle of HydLa is to take a constraint-centric approach to allow declarative and concise description of hybrid systems. In particular, constraint hierarchies are expected to autonomously impose the 'right amount' of constraints on variables so that the set of enabled constraints does not become over- or under-constrained. Still, we have seen many programs which do not compute trajectories or which compute unintended trajectories. In these cases, the debugging of declarative programs turned to be highly nontrivial to novice programmers.

```
INIT   <=> (x = 0 & y = 10 & y' = 0).
FALL   <=> [](x' = 1 & y'' = -10).
BOUNCE <=> [](y- = 0 => y' = -y'-).

INIT, (FALL << BOUNCE).
//#hylagi -p10
```

Fig. 20. A model with an unconstrained variable (1).

This motivated us to record a service log of webHydLa and analyze program errors, where each record contained (i) the HydLa program, (ii) the contents of `stdout` and `stderr`, and (iii) the 'hydat' file for visualization.

We analyzed 1017 HydLa programs after recording their standard output, error output and 766 hydat files passed to the webHydLa visualizer. The simulation results were divided into three categories:

1. normally terminated simulation (regardless of whether the result is intended or not),
2. simulation aborted by unsatisfiable constraints (in which the set of active constraints became inconsistent and none of them could be disabled), and
3. simulation in which some variable became totally unconstrained (which is semantically allowed but regarded as unintended).

We focus on the second and the third categories because they are specific to constraint programming.

Completely Unconstrained. 'Completely unconstrained' means that the constraints on the value of some variable become totally lost. HyLaGI does not stop execution for this event but generates a warning. The following are considered as possible causes of unconstrainedness.

1. a module that is defined but not declared (i.e., used),
2. lack of initial value constraints,
3. lack of 'always' (`[]`) constraints.

Since this is a warning not found in ordinary languages, we explain these causes using an example.

The first cause means that one simply forgot to use the defined constraint. On the other hand, the second and the third causes indicate insufficiency of constraints. For example, in the program of Fig. 20, `x` will become completely unconstrained after the second PP because the constraint on `x` is totally lost when the consequent of `BOUNCE` is enabled and `FALL` becomes unadopted temporarily. To fix the problem, we must either add a constraint `x' = 1` to the consequent of `BOUNCE` or move `x' = 1` from `FALL` to a new module.

Since this 'completely unconstrained' problem occurred much more frequently than expected, it was considered important to provide an *explanation*

```
INIT   <=> (y = 10 & y' = 0).
FALL   <=> [](y'' = -10).
BOUNCE <=> [](y- = 0 => y' = -y'-).

INIT, FALL, BOUNCE.
```

Fig. 21. A model causing inconsistency (1).

```
INIT  <=> (a = 0 & b = 0).
CONST <=> [](a' = 0).
CLOCK <=> [](b' = 1).
JUMP  <=> [](b = 3 => a = a + 1 & b = 0).

INIT, (CONST, CLOCK) << JUMP.
```

Fig. 22. A model causing inconsistency (2).

of the reason of unconstrainedness. We improved HyLaGI to infer and report whether the initial value constraint was insufficient or the *always* constraint was insufficient based on when the unconstrainedness occurred. If the first PP leaves any variable unconstrained, some initial value constraint is missing. If a variable becomes completely unconstrained after the first phase, we find, for each such variable, the module that caused the unconstrainedness. When the cause is the weakest module, HyLaGI displays a message "`WARNING:` x `is completely unconstrained in a default module`" because the module is supposed to represent default behavior. Otherwise, HyLaGI displays a message "`WARNING:` x `is completely unconstrained in a non-default module`".

Unsatisfiable Constraints. 'Unsatisfiable constraints' means that HyLaGI could not find a consistent set of constraint modules that respects constraint hierarchy. The following causes can be considered.

1. forgetting to define appropriate constraint hierarchy,
2. some of the required constraint modules, i.e., ones at the top of the constraint hierarchy, are mutually inconsistent or self-inconsistent.

For example, in Fig. 21, `FALL` and `BOUNCE` conflict with each other when the ball collides with the floor, but because there is no constraint hierarchy and all

```
Possible causes...
* {a} in {JUMP}
* {b} in {JUMP}
```

Fig. 23. Execution result of Fig. 22.

top-level modules are handled as required constraints (Sect. 2), `FALL` cannot be rejected and computation stops. In this case, the error can be easily resolved by declaring a hierarchy `FALL << BOUNCE`.

Consider another example in Fig. 22. When `b` becomes 3, `a` is incremented and `b` is reset to 0 by `JUMP`, so `a` looks like a counter and `b` looks like a clock. However, `JUMP` becomes inconsistent when (and only when) `b` $= 3$ because all variables of HydLa are immutable functions of time. As suggested by previous examples, an equation for discrete changes should mention the left limit values of variables, that is, `JUMP` should be written as `[](b- = 3 => a = a- + 1 & b = 0)`.

When simulation generated unsatisfiable constraints, the reason of inconsistency is not easy to figure out in many cases. We thus let HyLaGI show which variables are in conflict within which modules. For example, given the program of Fig. 22, a message like Fig. 23 will be displayed in addition to the standard error message. The analysis is done as follows. The `unsat modules` line of the output (such as Fig. 2) tells the names of mutually inconsistent modules, and the corresponding `unsat constraints` line contains information about mutually inconsistent constraints. HyLaGI extracts variables from each of `unsat constraints` and collects corresponding modules from `unsat modules`. In this way, for each set of variables, a set of modules that make those variables over-constrained is derived as shown in Fig. 23.

5 Solving Inverse Problems

Inverse problems are to obtain initial conditions that yield given final goals. Inverse problems of hybrid systems are more intriguing than those of continuous systems in that initial and final states may be related by qualitatively different trajectories, e.g., trajectories of balls with different numbers of bounces. HyLaGI can solve inverse problems of hybrid systems by combining assertions and symbolic constraint solving with automatic case analysis of parameters. Note that our approach is based on forward symbolic simulation rather than reverse simulation from the goal state.

5.1 A Simple Example

Let us consider how to shoot a golf ball to make a hole-in-one (Fig. 24). A program is shown in Fig. 25. We parameterize the `x` component of the initial velocity, while the `y` component is defined so that the initial speed (norm of the velocity) is constant. The ball moves at a constant speed in the `x` direction, while it behaves like a bouncing ball in the `y` direction.

We use `ASSERT` to find the range of parameters for hole-in-one, Assume that the cup is 9.5 to 10 meters ahead. The constraint to be `ASSERT`ed is the negation of the desired goal, i.e., `!(y = 0 & 9.5 <= x <= 10)`, so that HyLaGI may find counterexamples.

Table 2 shows the behavior of the ball and the corresponding parameter ranges obtained from the program of Fig. 25, where *bounce* means that the ball bounces and *cup-in* means that the ball enters the cup.

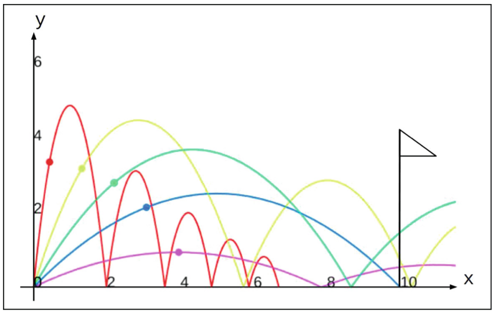

Fig. 24. A hole-in-one problem.

```
INIT    <=> x = 0 & y = 0 & 1 < x' < 9 & y' = (100 - x'^2)^0.5.
AXCONST <=> [](x'' = 0).
FALL    <=> [](y'' = -10).
BOUNCE  <=> [](y- = 0 => y' = -0.8*y'-).

ASSERT(!(y = 0 & 9.5 <= x <= 10)).
INIT, AXCONST, FALL << BOUNCE.
//#hylagi --fnd -p10
```

Fig. 25. Finding parameters for hole-in-one.

Table 2. Execution result of Fig. 25.

Behavior	Parameter range
Cup-in	$\left[\sqrt{\frac{5(20-\sqrt{39})}{2}}, \sqrt{\frac{5(20+\sqrt{39})}{2}}\right]$
Bounce, cup-in	$\left[\frac{5\sqrt{\frac{36-\sqrt{935}}{2}}}{3}, \frac{5\sqrt{14}-10}{3}\right]$
Bounce, bounce, cup-in	$\left[5\sqrt{\frac{244-\sqrt{50511}}{122}}, 5\sqrt{\frac{2(61-6\sqrt{86})}{61}}\right]$
Bounce, bounce, bounce, cup-in	$\left[\frac{5\sqrt{\frac{1476-\sqrt{1952951}}{82}}}{3}, \frac{5\sqrt{\frac{2(369-2\sqrt{30134})}{41}}}{3}\right]$
Bounce, bounce, bounce, bounce	Others

5.2 Examples with Persistent Consequents

HydLa's syntax allows an *always* constraint $\Box C$ to appear in the consequent of an implication. Such a constraint is called a *persistent consequent*. A persistent

```
1 INIT   <=> 5 < y < 10 & y' = 0 & d = 0.
2 FALL   <=> [](y'' = -10).
3 CONST  <=> [](d' = 0).
4 BOUNCE <=> [](y- = 0 => y' = -4/5 * y'- & d = d- + y'-^2 / 100).
5 BREAK  <=> [](d >= 4 => [](y'' = -10 & d' = 0)).
6
7 INIT, (FALL, CONST) << BOUNCE << BREAK.
8 ASSERT(y >= 0).
9 //#hylagi -p12 --fnd
```

Fig. 26. A bouncing ball damaging the floor.

Table 3. Execution result of Fig. 26.

Behavior	Parameter range
Bounce, bounce, bounce, bounce, bounce	(5, 7812500/968561)
Bounce, bounce, bounce, bounce, break	[7812500/968561, 312500/36121)
Bounce, bounce, bounce, break, through	[312500/36121, 12500/1281)
Bounce, bounce, break, through	[12500/1281, 10)

consequent $\square(G \Rightarrow \square C)$ is different from normal guarded constraints in that once the antecedent G holds, the consequent C continues to hold. The constraint $\square C$ with the same priority as the original constraint is expanded in the constraint hierarchy. Since constraints once expanded are not removed, persistent consequents can represent irreversible effects or changes of the system.

Figure 26 is a model in which the floor of a bouncing ball accumulates damage from the ball and is eventually broken. In `BOUNCE` of Line 4, damage proportional to the square of the velocity at each collision is accumulated on the floor. If the accumulated damage exceeds a certain threshold, the ball keeps falling, meaning that the floor is broken. To figure out in which conditions the floor breaks, we assert the constraint that the height of the ball is non-negative. Table 3 shows the system behavior and corresponding parameter ranges computed by HyLaGI from the program in Fig. 26. Here, *bounce* means that the ball bounces on the floor, *break* means that the ball bounces and the floor breaks, and *through* means that the ball passes through the broken floor.

Finally, we show an example with constraint hierarchy with three strengths. Figure 27 is a model that searches for a winning strategy of a chicken race: we want to stop the car exactly at the goal position by keeping acceleration to a certain point and then braking. The braking position is parameterized. We have two persistent consequents, `BRAKE` and `STAY`, where `STAY` is given higher priority so that the car will not move backwards after stop. The `ASSERT`ed constraint specifies the negation of the winning condition, and HyLaGI finds that the assertion fails when the parameter value is `625/2`, from which we can see that the winning strategy is to start braking at 312.5 m from the starting point.

```
INIT  <=> x = 0 & x' = 0 & 0 < brkpt < 500 & [](brkpt' = 0).
ACC   <=> [](x'' = 3).
BRAKE <=> [](x- = brkpt- => [](x'' = -5)).
STAY  <=> [](x'- = 0     => [](x'' = 0)).

INIT, ACC << BRAKE << STAY.
ASSERT(!(x = 500 & x' = 0)).
//#hylagi --fnd
```

Fig. 27. Chicken race program.

6 Conclusion

In this paper, we first discussed our constraint-based approach to hybrid systems embodied as a modeling language HydLa and introduced various functionalities of HyLaGI, a symbolic simulator of hybrid systems expressed in HydLa. These functionalities, including nondeterministic execution, handling of infinitesimal quantities, and construction of hybrid automata, are realized since HyLaGI adopts symbolic computation. Then, we discussed several findings and experiences in the constraint-based modeling of hybrid systems including two different uses of guarded constraints and modeling errors mostly resulting from improper use of constraint hierarchy. Finally, we showed that HyLaGI could solve some inverse problems of hybrid systems and that persistent consequents are useful for modeling inverse problems.

Although HydLa has many unique features as described above, hybrid models that can be handled by the current version of HyLaGI are limited to relatively simple ones due to various limitations. In our experience, computation of the time of the next discrete change is the most difficult part for the constraint engine, due to which many models with closed-form solutions of ODEs could not be fully simulated to the end. To address this problem, Matsumoto et al. [15] reports how we can integrate symbolic versions of Affine arithmetic and the interval Newton method into our framework. Also, in order to reduce the complexity of constraints submitted to the constraint engine to improve the power of constraint solving, we have incorporated a number of optimization techniques into HyLaGI.

There are many other issues including the handling of models with many parameters and models with complicated differential equations such as DAEs and nonlinear ODEs. Still, we feel that the usefulness of our constraint-based framework is being established. Our future goal is to extend our framework by introducing useful results in the field of constraint programming and hybrid systems.

Acknowledgments. We would like to thank Shota Matsumoto for leading the development of the core part of HyLaGI. Early versions of HyLaGI are due to previous colleagues including Ken-ichi Hirose. Of the developers of many experimental features, Akira Takeguchi and Yoshiaki Wakatsuki were initial designers and implementors of the features mentioned in this paper. We thank all members of the project, including the above members, for discussions, development, and debugging. Thanks go also to anonymous reviewers for their detailed and constructive comments. The work is partially supported by Grant-in-Aid for Scientific Research (B) JP18H03223, JSPS, Japan.

A Appendix

Figure 28 shows the syntax of HydLa with the list notation. As a key extension from Fig. 4, we newly introduce *PL* (priority list), *EL* (expression list), *LC* (list condition) and a list binding notation ":=". Both *PL* and *EL* consist of extensional and list comprehension notations. In the list comprehension notation, one can enumerate elements that satisfy conditions specified by *LC*. We can generate variables with successive serial numbers using range expressions (*RE* in Fig. 28) and bind them to upper-case variables using ":=". We can generate a list of module declarations in a similar manner. *MPname*, *ELname*, *PLname*, and *Iname* stand for names for module priority definitions, expression lists, priority lists, and elements from iterators, respectively.

$$
\begin{array}{rll}
\text{(HydLa program) } P & ::= & (\mathit{Def} \mid \mathit{Decl})^* \\
\text{(definition) } \mathit{Def} & ::= & \mathit{MPname}(\overrightarrow{X})\{\mathit{MP}\} \mid \mathit{cname}(\overrightarrow{X}) \Leftrightarrow C \\
 & & \mid \mathit{ELname} := \mathit{EL} \mid \mathit{PLname} := \mathit{PL} \\
\text{(constraint) } C & ::= & A \mid C \wedge C \mid G \Rightarrow C \mid \exists \mathit{vname}\,.\, C \mid \Box C \mid \mathit{cname}(\overrightarrow{E}) \\
\text{(list condition) } \mathit{LC} & ::= & \mathit{MPname} \in \mathit{PL} \mid \mathit{Iname} \in \mathit{EL} \mid E \neq E \\
\text{(priority list) } \mathit{PL} & ::= & \{\mathit{MP}(,\mathit{MP})^*\} \mid \{\mathit{MP} \mid \mathit{LC}(,\mathit{LC})^*\} \mid \mathit{PLname} \\
\text{(module priority) } \mathit{MP} & ::= & C \mid \mathit{MPname}(\overrightarrow{E}) \mid \mathit{MP}, \mathit{MP} \mid \mathit{MP} \ll \mathit{MP} \\
\text{(guard) } G & ::= & A \mid G \wedge G \mid G \vee G \mid \neg G \\
\text{(atomic constraint) } A & ::= & E \; \mathit{Rop} \; E \\
\text{(relational operator) } \mathit{Rop} & ::= & = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
\text{(expression) } E & ::= & E \; \mathit{Aop} \; E \mid \mathit{Prev} \mid \mathit{constant} \mid \mathit{EL}[E] \\
\text{(expression list) } \mathit{EL} & ::= & \{E(,E)^*\} \mid \{E \mid \mathit{LC}(,\mathit{LC})^*\} \mid \mathit{ELname} \mid \mathit{RE}\,..\,\mathit{RE} \\
\text{(arithmetic operator) } \mathit{Aop} & ::= & + \mid - \mid \times \mid \div \mid \hat{} \\
\text{(previous) } \mathit{Prev} & ::= & D \mid D- \\
\text{(derivative) } D & ::= & \mathit{vname} \mid D' \\
\text{(declaration) } \mathit{Decl} & ::= & M \mid \mathit{Decl}, \mathit{Decl} \mid \mathit{Decl} \ll \mathit{Decl} \\
\text{(module) } M & ::= & C \mid \mathit{dname}(\overrightarrow{E}) \mid \mathit{PL} \mid \mathit{PL}[E]
\end{array}
$$

Fig. 28. Syntax of HydLa with list notation.

References

1. Alur, R., et al.: Hierarchical hybrid modeling of embedded systems. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 14–31. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45449-7_2
2. Betsuno, K., Matsumoto, S., Ueda, K.: Symbolic analysis of hybrid systems involving numerous discrete changes using loop detection. In: Berger, C., Mousavi, M.R., Wisniewski, R. (eds.) CyPhy 2016. LNCS, vol. 10107, pp. 17–30. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51738-4_2
3. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. LISP Symb. Comput. **5**(3), 233–270 (1992)
4. Bourke, T., Pouzet, M.: Zélus: a synchronous language with ODEs. In: HSCC 2013, pp. 113–118. ACM (2013)
5. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and tools for hybrid systems design. Found. Trends Electron. Des. Autom. **1**(1/2), 1–193 (2006)
6. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
7. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
8. Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D.G.: Programming in hybrid constraint languages. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1994. LNCS, vol. 999, pp. 226–251. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60472-3_12
9. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE Computer Society (1996)
10. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
11. Lee, E.A.: Constructive models of discrete and continuous physical phenomena. IEEE Access **2**, 797–821 (2014)
12. Lunze, J.: Handbook of Hybrid Systems Control: Theory, Tools, Applications. Cambridge University Press, Cambridge (2009)
13. Matsumoto, S.: Validated simulation of parametric hybrid systems based on constraints. Ph.D. thesis, Waseda University (2017)
14. Matsumoto, S., Kono, F., Kobayashi, T., Ueda, K.: HyLaGI: symbolic implementation of a hybrid constraint language HydLa. Electron. Notes Theor. Comput. Sci. **317**, 109–115 (2015)
15. Matsumoto, S., Ueda, K.: Symbolic simulation of parametrized hybrid systems with affine arithmetic. In: TIME 2016, pp. 4–11. IEEE Computer Society (2016)
16. Modelica Association: Modelica - Unified Object-Oriented Language for Systems Modeling: Language Specification (Version 3.4) (2007). https://modelica.org/documents/ModelicaSpec34.pdf
17. Ñañez, P., Risso, N., Sanfelice, R.G.: A symbolic simulator for hybrid equations. In: Proceedings of SummerSim 2014, pp. 18:1–18:8 (2014)

18. Taha, W., et al.: Acumen: an open-source testbed for cyber-physical systems research. In: Mandler, B., et al. (eds.) IoT360 2015. LNICST, vol. 169, pp. 118–130. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47063-4_11
19. Takeguchi, A., Wada, R., Matsumoto, S., Hosobe, H., Ueda, K.: An algorithm for converting hybrid constraint programs to hybrid automata. In: The 29nd JSSST Annual Conference, 2A-3 (2012). https://www.ueda.info.waseda.ac.jp/~ueda/pub/takeguchi_jssst_PPL2012.pdf. (in Japanese)
20. Ueda, K., Hosobe, H., Ishii, D.: Declarative semantics of the hybrid constraint language HydLa. Comput. Softw. **28**(1), 306–311 (2011). English translation: http://arxiv.org/abs/1910.12272
21. Ueda, K., Matsumoto, S., Takeguchi, A., Hosobe, H., Ishii, D.: HydLa: a high-level language for hybrid systems. In: 2nd Workshop on Logics for System Analysis (LfSA 2012, affiliated with CAV 2012), pp. 3–17 (2012)
22. Wakatsuki, Y., Matsumoto, S., Ito, T., Wada, T., Ueda, K.: Model analysis by using micro errors in hybrid constraint processing system HyLaGI. In: The 32nd JSSST Annual Conference (2015). http://jssst.or.jp/files/user/taikai/2015/GENERAL/general6-4.pdf. (in Japanese)