Validated Simulation of Parametric Hybrid Systems Based on Constraints

February 2017

Shota MATSUMOTO

Validated Simulation of Parametric Hybrid Systems Based on Constraints

February 2017

Waseda University
Graduate School of Fundamental Science and Engineering
Department of Computer Science and Engineering
Research on Parallel Knowledge Information Processing

Shota MATSUMOTO

Abstract

The purpose of this research is to develop a highly reliable simulator of hybrid systems, i.e., systems involving both discrete changes and continuous evolutions. In particular, we aim at rigorous simulation of parametric hybrid systems, which enables not only the analysis of possible behavior of models but also the design of parameters that realize desired properties. Simulators with interval arithmetic can reliably compute a reachable set of states, but preserving the dependencies of uncertain quantities in models is still challenging.

In this thesis, first we discuss a simulation method that is based on symbolic formula manipulation. This method can simulate all possible trajectories of hybrid systems described by a constraint-based formalism. The results of computation is free from errors caused by floating-point arithmetic. This method can perform case analysis if a target model includes qualitative branching of trajecotries depending on parameters.

Next, we discuss integration of the symbolic method with conservative overapproximation by interval arithmetic. We focus on (i) reducing computational costs of complex symbolic formulas and (ii) computing zero-crossings of functions that cannot be handled analytically. This integrating method uses affine arithmetic, the interval Newton method and the mean value theorem. This method broadens the scope of applicable models and still preserves the first-order dependencies of uncertain quantities throughout simulation. Preservation of such dependencies improves the accuracy of the results because it reduces the shortcomings of naïve interval arithmetic.

We also discuss a symbolic simulator that implements the above methods. It features bounded model checking as a natural extension of the symbolic simulation. It is publicly available and has a web frontend that supports plotting of parametric trajectories. We show the performance of the implementation with example models.

Acknowledgments

The contributions in the thesis are achieved under great help of many people. I thank Prof. Kazunori Ueda very much for encouraging and leading me in the course of the research. His help has been very valuable and essential to complete this research. I am really grateful to Prof. Masahide Kashiwagi for his helpful comments about validated numerical computation. The simulation method in the thesis could not have been completed without his suggestions. I thank the thesis committee members, Prof. Yasuo Matsuyama and Prof. Toshiharu Sugawara for their comments and support to improve the thesis. I want give my great thanks to Dr. Daisuke Ishii and Prof. Hiroshi Hosobe. Discussions with them have been illuminating. I appreciate insightful opinions that the members of HydLa project and Ueda laboratory gave me in our daily discussions. Finally, I would like to express the deepest appreciation to my parents for their support over long years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Hybrid systems [23] are dynamical systems involving both discrete changes and continuous evolutions. Interactions between computer programs and physical environments can be naturally regarded as hybrid systems. Such systems are also called cyber-physical systems, which are important applications of hybrid systems. (In some context, the term "cyber-physical system" also has an aspect about control methods based on big data collected by IoT sensors, however it is different from our focus.) Hybrid systems include safety critical systems such as control of vehicles, nuclear plants, etc. Figure 1.1 shows an example of such hybrid systems, in which two aircrafts are continuously cruising and discretely change their courses to avoid collision. Therefore, safety is a major concern about hybrid systems. Rigorous handling of computational errors is important for the validation and verification of models because those errors may lead to qualitatively wrong results.

In the rigorous handling of hybrid systems, there are two major challenging points. The first challenge is to solve ordinary differential equations (ODEs), which can be nonlinear and have no analytic solutions (e.g., the Van der Pol oscillator). The second challenge is to compute cross points of continuous trajectories and boundaries at which discrete events happen. Also, it is important to be able to handle uncertainties in models, which are caused by modeling errors, observation errors, deterioration of models due to aging, etc. Many tools that enable rigorous analysis of such hybrid systems have been developed [5][10][16][30][39][3][31][27]. Most if not all of the tools are based on interval arithmetic [28], and can compute over-approximated

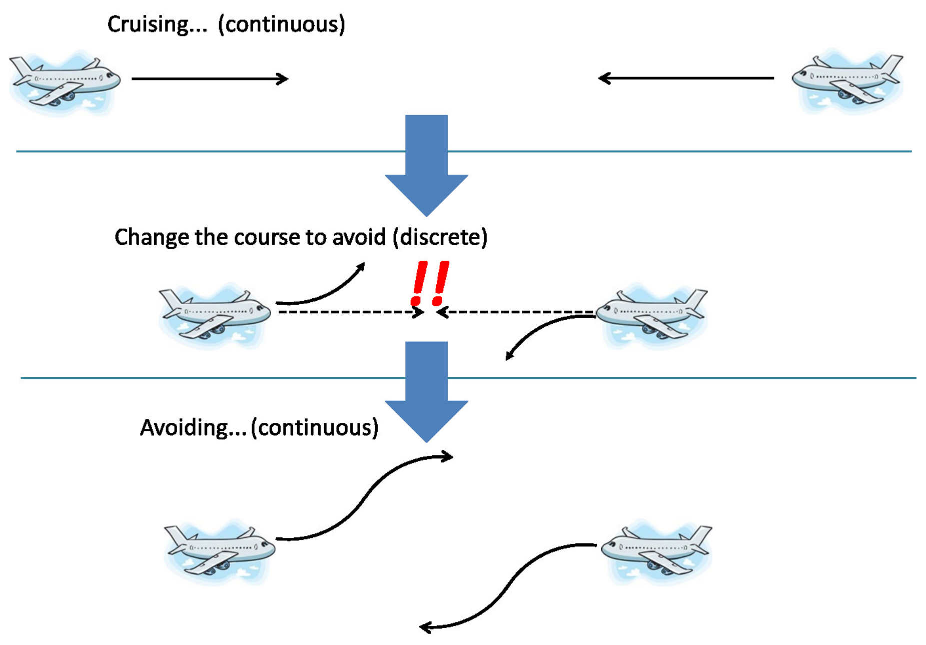


Figure 1.1: Collision avoidance of aircrafts

reachable sets of systems.

Parametric hybrid systems, namely hybrid systems containing symbolic parameters whose values may have lower and upper bounds, allow us to express a family of hybrid systems. They have diverse applications including the dependency-preserving modeling of uncertainties, the sensitivity analysis, and the design of models and controllers. In existing tools, the dependencies between uncertain quantities are ignored or handled only implicitly because they mainly focus on reachability analysis of models.

1.2 Research Goals and Contributions

The main goal of this research is to develop a simulator that handles the dependencies between uncertain quantities explicitly by parametric representations of models. Two challenges here are to solve parametric ordinary differential equations and to handle discrete changes of parametric hybrid systems. Regarding the former challenge, we assume that a parametric ODE is linear or can be safely enclosed by another linear parametric ODE (with additional parameters representing approximation errors). This thesis focuses on the latter challenge. Boundaries of discrete events are often expressed by nonlinear equations that have no analytical solutions, in which case we must employ interval techniques to compute time intervals of discrete changes.

The contributions of this research are described in the following subsections.

1.2.1 Symbolic Simulation Algorithm Based on Constraint Solving

The algorithm is based on constraint solving techniques [32] with symbolic formula manipulation. Symbolic formula manipulation has the following advantages.

1. It features error-free computation using analytic solutions.
2. It features symbolic execution of parametric hybrid systems.
3. Symbolic formulas can be regarded as the most general representation and other computation frameworks can be naturally involved in symbolic formula manipulation.

The input of the algorithm is in the form of programs written in a constraint-based language HydLa. The symbolic method performs on models with large uncertainties much better than numerical simulation. The algorithm can perform case analysis on demand. In particular, it can detect singular points in behavior of models and automatically avoid stuck of simulation. We discuss the detail of the algorithm in Chapter 4.

1.2.2 Integrating Symbolic Simulation with Interval Arithmetic

Symbolic simulation has the following drawbacks:

1. Data structures of symbolic formulas are more complex than that of numerical computation.
2. As computation goes on, the complexity of symbolic formulas such as the number of terms and the number of digits of coefficients increases and it leads to the explosive increase of computational costs.

Because of the drawbacks, the fully symbolic method itself is not applicable for many of parametric hybrid systems. Our experience with parametric hybrid systems showed that, even for rather simple hybrid systems with linear ODEs that can be analytically solved, symbolic simulation often gets stuck because it is not able to solve parametric equations representing the time of the next discrete event. We developed a method that integrates the symbolic method with interval arithmetic to overcome such drawbacks. The method has two key ideas. The first idea is to over-approximate symbolic formulas with affine arithmetic and use approximated values instead of original symbolic formulas. Affine arithmetic is an extended version of interval arithmetic and can preserve the first-order dependencies between quantities using symbolic parameters. It can reduce the computational costs caused by complex formulas. The second idea is solving zero-crossings of functions that describe the conditions of discrete events using the interval Newton method and the mean value theorem. This method can solve parametric algebraic equations that cannot be solved analytically, preserving the linear dependencies of parameters. The combination of these techniques turns out to work even when affine arithmetic alone does not work in computing discrete changes. We discuss the detail of the proposed method in Section 4.2.

1.2.3 Implementation of A Symbolic Simulator

We implemented above methods in our tool named HyLaGI. HyLaGI is written in C++ and Mathematica [37] and uses KV library [19] for interval and affine arithmetic. HyLaGI can compute all possible trajectories of parametric hybrid systems, which are expressed by a sequence of symbolic representations of states. It also features bounded model checking as a natural extension of symbolic simulation. It is publicly available on our website with a web frontend. We discuss the implementation in Chapter 5 and its performance in Chapter 6.

1.3 Outline

This thesis is organized as follows. In Chapter 2, we show our notational convention and describe the definitions of constraints, interval arithmetic and affine arithmetic. In Chapter 3, we introduce our modeling formalism of hybrid systems named HydLa, which is used as the input language of our simulation algorithm. In Chapter 4, we describe our symbolic simulation algorithm first. The symbolic algorithm has been published in [26]. We also describe how we integrate interval arithmetic with the symbolic simulation algorithm. The method of integration has been published in [25]. In Chapter 5, we introduce our implementation named HyLaGI. This chapter is based on the publications [20][22][24][26]. In Chapter 6, we evaluate our implementation through several example models. This chapter is partially based on the publications [26][25]. In Chapter 7, we mention related work and describe the position of our work. In Chapter 8, we review the summary of the thesis and describe future work.

Chapter 2

Preliminaries

In this chapter, we introduce basic notations and concepts that are used in the thesis.

2.1 Notations

Basic notations used in this thesis are listed in Table 2.1.

2.2 Constraints

In this thesis, we use the term *constraints* as logical formulas about real-valued variables. We allow "$=$", "$<$","$>$","$\leq$","$\geq$", and "$\neq$" as relational operators in constraints. A constraint that contains only one relational operator is called an *atomic constraint*. Each constraint consists of atomic constraints, their disjunctions and conjunctions. An *assignment* is a tuple that represents values of variables. For a constraint C, *assignments*(C) means a set of assignments that satisfies C. Note that we distinguish a constraint and the set of its possible assignments. A constraint is called *consistent* or *satisfiable* iff there is at least one assignment of variables that satisfies the constraint and otherwise called *inconsistent*. These conditions are denoted by *consistent*(C) and *inconsistent*(C). A constraint C_1 is called *entailed* by a constraint C_2 iff $C_2 \Rightarrow C_1$ is valid.

Example 1. *Let $X = \{x, y\}$ a set of variables. $0 < x \wedge x < 1 \wedge y = 2$ is a constraint that consists of three atomic constraints $\{0 < x, x < 1, y = 2\}$. $0 < x \wedge x < 1 \wedge y = 2$ is consistent and $\langle 1, 2\rangle$ is one of possible assignments,*

Table 2.1: Basic notations

Notation	Description
$n \in \mathbb{N}$	Natural numbers
$r \in \mathbb{R}$	Real numbers
$f \in \mathbb{F}$	Floating-point numbers
(a, b)	An open real interval $\{x \in \mathbb{R} \mid a < x < b\}$
$[a, b]$	A closed real interval $\{x \in \mathbb{R} \mid a \leq x \leq b\}$
$(a, b]$	A left-open real interval $\{x \in \mathbb{R} \mid a < x \leq b\}$
$[a, b)$	A right-open real interval $\{x \in \mathbb{R} \mid a \leq x < b\}$
$i \in \mathbb{I}$	Closed real intervals
$\overline{I}$	Upper bound of an interval I
$\underline{I}$	Lower bound of an interval I
$mid(I)$	Midpoint of I $(= \frac{\underline{I}+\overline{I}}{2})$
$rad(I)$	Radius of I $(= \frac{\overline{I}-\underline{I}}{2})$
$\{x_1, .., x_n\}$	Set
$\langle x_1, .., x_n\rangle$	Tuple
$\vec{x}$	Column vector
$\vec{x}^T$	Row vector
$x \models C$	An assignment x satisfies constraint C

which means $x = 1 \wedge y = 2$. *assignments(*$0 < x \wedge x < 1 \wedge y = 2$*) is equivalent to* $\{\langle i, 2\rangle \mid 0 < i < 1\}$. $x < 0 \wedge 1 < x$ *is inconsistent because there are no possible assignments. A constraint* $0 < x \wedge x < 1 \wedge y = 2$ *is entailed by a constraint* $(x = \frac{1}{4} \vee x = \frac{3}{4}) \wedge y = 2$.

2.3 Interval Arithmetic

Interval Arithmetic (IA) [28] is an arithmetic to guarantee the range of the possible results of computation. We denote the universal set of intervals by $\mathbb{I}$ in the thesis. Operations in IA are defined on closed intervals of real numbers. In IA, both the range of computational errors and uncertain quantities can be expressed by intervals.

Example 2. *Let A and B be real intervals. The basic four operations in IA are defined as follows:*

$$\begin{aligned}
&A + B = [\underline{A} + \underline{B}, \overline{A} + \overline{B}]\\
&A - B = [\underline{A} - \overline{B}, \overline{A} - \underline{B}]\\
&A \times B =\\
&\quad [min(\underline{A} \times \underline{B}, \overline{A} \times \underline{B}, \underline{A} \times \overline{B}, \overline{A} \times \overline{B}), max(\underline{A} \times \underline{B}, \overline{A} \times \underline{B}, \underline{A} \times \overline{B}, \overline{A} \times \overline{B})]\\
&A \;/\; B = A \times [1/\overline{B}, 1/\underline{B}] \qquad \mathit{if}\, 0 \notin B
\end{aligned}$$

In computation on actual computers, we use floating-point numbers complying with the IEEE 754 standard [17] as endpoints of intervals. We have to carefully handle rounding directions in computation of floating-point numbers to conservatively approximate the result intervals. For example, in the four operations in Example 2, we have to round down the results for lower bounds and round up the results for upper bounds.

If primitive operations in IA are provided, we can construct an inclusion function $F(X)$ of an arbitrary function $f(x)$ that can be expressed by combination of the primitive operations such that

$$F(X) \supseteq \{f(x) : x \in X\}.$$

However, ordinary IA has some disadvantages such as the wrapping effect and the dependency problem, which lead to increase of the widths of computed intervals. The wrapping effect is an effect that arises from the representation of uncertain values in IA, that is, axis-aligned boxes.

Example 3. *Consider a two-dimensional box $B_0 := ([-1, 1], [-1, 1])^T$ and rotated boxes defined by*

$$B_{n+1} := \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} B_n.$$

If we compute B_1 by IA, it results in

$$\begin{aligned}
B_1 &= (\frac{1}{\sqrt{2}} \times [-1, 1] - \frac{1}{\sqrt{2}} \times [-1, 1], \frac{1}{\sqrt{2}} \times [-1, 1] + \frac{1}{\sqrt{2}} \times [-1, 1])^T\\
&= ([-\sqrt{2}, \sqrt{2}], [-\sqrt{2}, \sqrt{2}])^T.
\end{aligned}$$

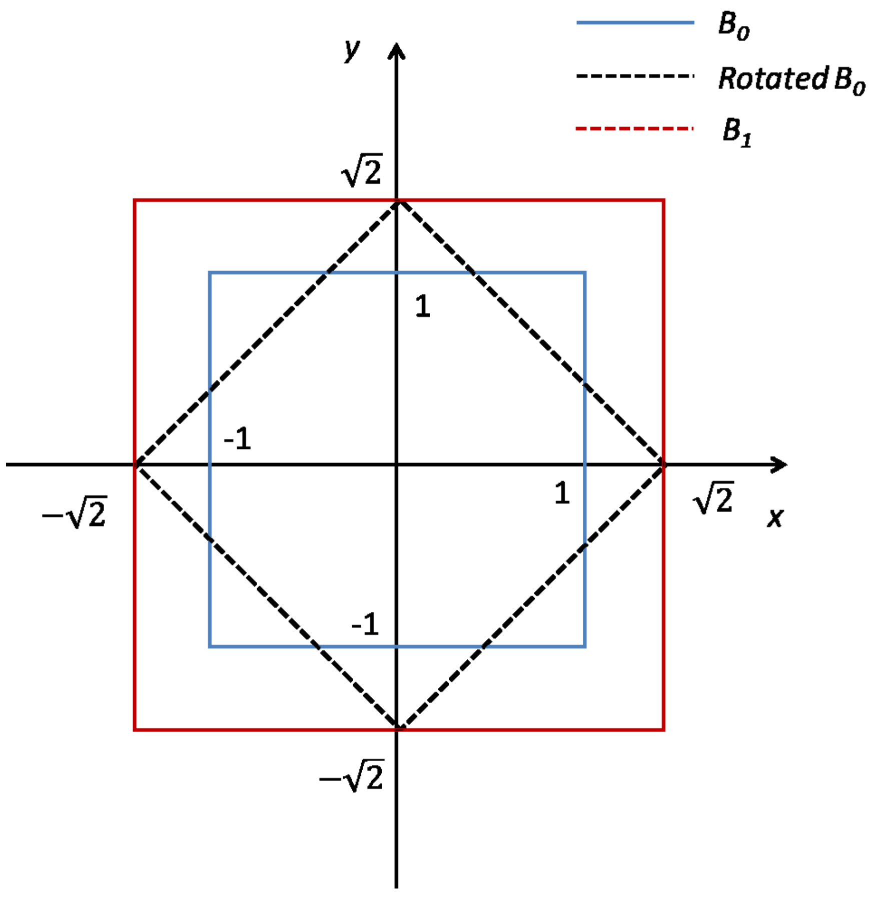


Figure 2.1: Example of wrapping effect

The rotation is graphically shown in Fig. 2.1. The area of B_1 *is twice as large as* B_0*.* B_1 *wraps rotated* B_0 *by an axis-aligned box, that is why it is called the "wrapping effect". If we continue to compute the series of the boxes, the area of* B_n *increases exponentially.*

The dependency problem is a problem that is caused by ignoring the dependencies between uncertain quantities.

Example 4. *An extreme case of the dependency problem arises in estimation of* $f_1(x) = x - x$ *in IA. Consider an input interval* $I_1 = [-2, 2]$*. The resultant estimation of the inclusion function* $F_1(X)$ *is* $F_1(I_1) = [-2, 2] - [-2, 2] = [-4, 4]$*, while the ideal value of* $f_1(x)$ *exactly equals zero for any inputs.*

In the above example, we can improve the answer by evaluating $x - x$ into

0 in advance. However, such a policy cannot be taken in general cases. The dependency problem takes an effect in any functions that include multiple occurrences of the same variables.

There have been many proposals that address these problems [14][1][28]. Affine arithmetic (AA) [8], one of those methods, is an extended version of IA focusing on the first-order dependencies of intervals.

2.4 Affine Arithmetic

Affine arithmetic (AA) [8] is a method to enclose the results of numerical computation. AA handles the dependency problem of interval arithmetic by preserving the first-order dependencies between quantities. In AA, a quantity X is represented with the form (called an affine form)

$$X = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n.$$

Here, the ϵ_i's are symbolic parameters called noise symbols representing uncertain values within $[-1, 1]$; x_0 is called the *central value*; and the x_i's ($i > 0$) are called the *partial deviations*. An ϵ_i occurring in two or more quantities represents the dependencies between them. Each quantity of AA can be transformed into an interval $[x_0 - \sum_{i=1}^{n} x_i, x_0 + \sum_{i=1}^{n} x_i]$.

Figure 2.2 shows a two-dimensional area expressed by affine forms, which is denoted by the blue shape. The dashed line shows a corresponding interval box. In general, a tuple of affine forms can express a *zonotope*, that is, a convex polytope that is symmetric with respect to the central point. In other words, each pair of opposite edges of a zonotope is parallel.

Example 5. *Consider two affine forms*

$$X = x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n$$
$$Y = y_0 + y_1\epsilon_1 + \cdots + y_n\epsilon_n.$$

Affine operations such as addition, subtraction and constant multiplication are defined as follows.

$$X \pm Y = (x_0 \pm y_0) + (x_1 \pm y_1)\epsilon_1 + \cdots + (x_n \pm y_n)\epsilon_n$$
$$aX = ax_0 + ax_1\epsilon_1 + \cdots + ax_n\epsilon_n.$$

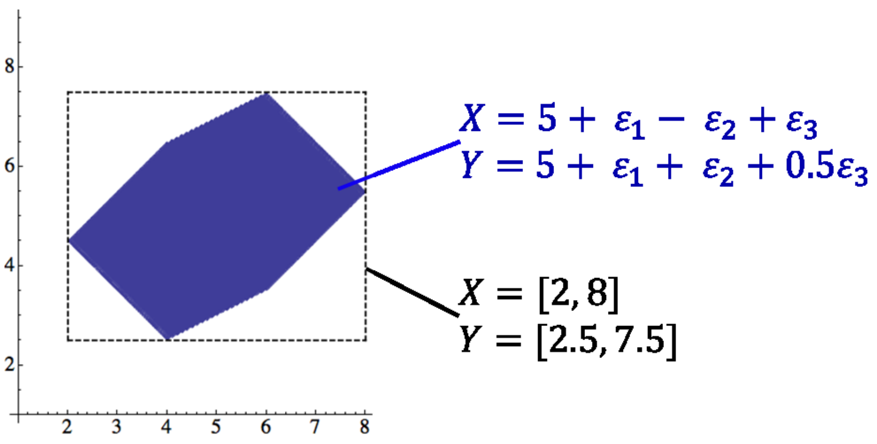


Figure 2.2: Comparison of interval box and affine form

For each operation, new noise symbols are introduced to handle rounding errors of floating-point arithmetic and approximation errors for non-affine operations. In AA, using more noise symbols may produce more accurate results but with higher computational costs. To control the trade-off between accuracy and cost, an algorithm to reduce noise symbols has been proposed [18]. The reduction algorithm removes a given number of noise symbols whose influences to quantities are the smallest. Our proposed method uses the reduction algorithm at each discrete change (see Chapter 4).

Chapter 3

HydLa: A Constraint-based formalism of Hybrid Systems

In this chapter, we introduce our formalism of hybrid systems named HydLa, which can be taken as an input by the proposed simulation algorithm (Chapter 4). Several modeling formalisms for hybrid systems have been proposed [3]. Hybrid automata [15] are the most famous, which express discrete changes by edges and continuous evolutions by nodes. Hybrid automata enable flexible modeling using invariants, flows, resets and guards. However, users have to enumerate all possible states of models as nodes. Acumen [39] and KeYmaera [30] employ different modeling methods based on imperative programming languages.

HydLa [34] is a constraint-based language for hybrid systems. HydLa employs a constraint-based formalism in modeling. HydLa directly uses mathematical notations as much as possible aiming at easy understanding by non-programmers such as mathematicians, physicists and so on. A constraint-based formalism is declarative but yet provides the language with control structures including synchronization and conditionals. Moreover, it allows us to handle uncertainties in a smooth way; that is, a constraint-based language naturally lends itself to symbolic execution of programs with symbolic parameters. There is another modeling language of hybrid systems based on constraints, named Hybrid CC[4][13]. The main difference between Hybrid CC and HydLa is that HydLa also features constraint hierarchies [2]. A constraint hierarchy in HydLa consists of a partially ordered set of constraints derived from priorities among them. This feature enables HydLa programmers to describe exceptional behaviors and default behaviors of models simply and naturally.

3.1 Syntax

Figure 3.1 shows the abstract syntax of HydLa. A HydLa program consists of *definitions* and *declarations* of constraints. In HydLa, all variables are functions w.r.t. time implicitly. A hydla program declares constraints that have to be satisfied by variables. Each constraint definition describes properties that hold at time 0.

In a definition *DF*, we can define a named constraint (*cname*) or a named constraint hierarchies (*dname*) with arguments ($\vec{X}$). The names *cname* and *dname* are denoted by capitalized strings. *Constraints* allow conjunctions of constraints and implications. The antecedents of implications are called *guards* and we can use disjunctions and conjunctions in a guard. "$\Box$" is a temporal operator which means that the constraint always holds from the time point at which the constraint is enabled. Each variable is denoted by a string starting with lower case (*vname*). The notation *vname′* means the derivative of *vname*, and *vname−* means the left-hand limit of *vname*. As a syntactic sugar, we can use chains of relational expressions such as $0 < x < y < 1$, which means $0 < x \wedge x < y \wedge y < 1$. Table 3.1 shows the correspondence between the abstract syntax and the concrete one.

In a declaration *DC*, we declare constraints with priorities between them. The operator "$\ll$" describes a weak composition of constraints. For example, $A \ll B$ means that the constraint A is weaker than B. If we declare a constraint without "$\ll$", it means that there is no priority about the constraint. The operator "$\ll$" has a higher precedence than ",". The unit of constraints that is prioritized is called a *module* or a *constraint module*. We describe detailed semantics of priorities in Section 3.2.1.

Example 6. *Figure 3.2 shows a HydLa program of a bouncing ball model. In this model,* `y` *represents the distance between the ball and the ground.* `INIT` *describes the initial position and the initial velocity of the ball.* `FALL` *describes the constant acceleration of the ball by the gravity.* `BOUNCE` *describes bouncing of the ball on the ground. The coefficient of restitution is* `4/5`*. The bottom line is the declaration of the constraint hierarchy of this model.*

$$
\begin{array}{rlll}
\text{(hydla program)} & P & ::= & (DF \mid DC)^* \\
\text{(definition)} & DF & ::= & dname(\vec{X})\{DC\} \mid cname(\vec{X}) \Leftrightarrow C \\
\text{(constraint)} & C & ::= & A \mid C \land C \mid G \Rightarrow C \mid \Box C \mid cname(\vec{E}) \\
\text{(guard)} & G & ::= & A \mid G \land G \mid G \lor G \\
\text{(atomic constraint)} & A & ::= & E\ RO\ E \\
\text{(relational operator)} & RO & ::= & = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
\text{(expression)} & E & ::= & E\ AO\ E \mid P \mid constant \\
 & & & \mid unary\_function(E) \\
\text{(arithmetic operator)} & AO & ::= & + \mid - \mid \times \mid \div \mid \hat{} \\
\text{(previous)} & P & ::= & D \mid D- \\
\text{(derivative)} & D & ::= & vname \mid vname' \\
\text{(declaration)} & DC & ::= & M \mid DC,\ DC \mid DC \ll DC \\
 & & & \mid dname(\vec{E}) \\
\text{(module)} & M & ::= & C
\end{array}
$$

Figure 3.1: Syntax of HydLa

Table 3.1: Correspondence between abstract and concrete syntax

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

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

INIT, FALL << BOUNCE.
```

Figure 3.2: HydLa program of bouncing ball

3.2 Semantics

In this section, we describe the semantics of HydLa. The declarative meaning of a HydLa program is a set of *hybrid trajectories* that satisfy the specification given in the program [35]. The definition of a hybrid trajectory is as follows.

Definition 1. *A hybrid trajectory is a finite sequence $\bar{x} = \langle \vec{x}_{d1}, \langle \vec{x}_{c1}(\tau), \tau_1 \rangle, \vec{x}_{d2}, \langle \vec{x}_{c2}(\tau), \tau_2 \rangle, \ldots, \vec{x}_{dn}, \langle \vec{x}_{cn}(\tau), \tau_n \rangle \rangle$, wherein each $\vec{x}_{di}$ $(1 \leq i \leq n)$ is a tuple that represents the values of variables at each time point and $\vec{x}_{ci}(\tau)$ $(1 \leq i \leq n)$ is a tuple of functions that represents the values of variables w.r.t. time for each time interval (τ_{i-1}, τ_i). Each τ_i means the end point of a time interval for each continuous change. For every τ_i, $\tau_{i-1} < \tau_i$ $(1 \leq i \leq n)$ holds wherein $\tau_0 = 0$.*

A trajectory $\bar{x}$ can take an argument of time and its definition is as follows.

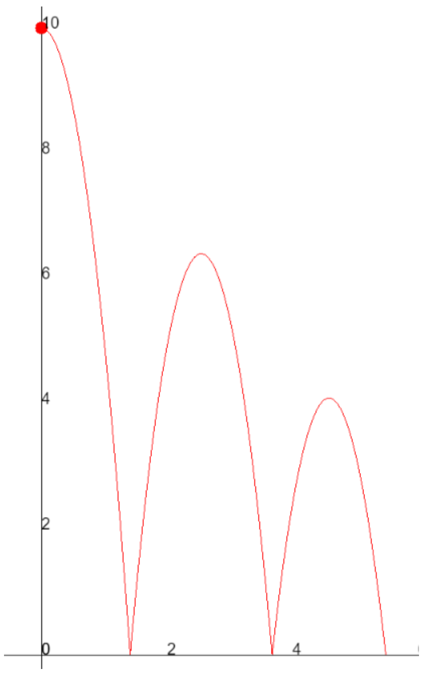


Figure 3.3: Trajectory of bouncing ball

Definition 2.

$$\bar{x}(t) = \begin{cases} \vec{x}_{di} & (t = \tau_{i-1}) \\ \vec{x}_{ci}(t) & (\tau_{i-1} < t < \tau_i) \end{cases}$$

Example 7. *Figure 3.3 shows a trajectory of a bouncing ball model for three bounces. The trajectory is expressed as a hybrid trajectory* $\bar{y}_{bb} = \langle 10, \langle 10 - 5t^2, \sqrt{2}\rangle, 0, \langle -(5t^2 - 18\sqrt{2}t + 26), 13/5\sqrt{2}\rangle, 0, \langle((-1/25) \times (125t^2 - 810\sqrt{2} + 2522), 97/25\sqrt{2}\rangle\rangle$*. All of the following conditions hold for* $\bar{y}_{bb}$*;* $\{\bar{y}_{bb}(0) = 10, \bar{y}_{bb}(1) = 5, \bar{y}_{bb}(\sqrt{2}) = 0\}$*.*

The semantics is given to basic HydLa, which can be obtained from original HydLa by transforming constraint hierarchies in HydLa programs into a partially ordered set of constraint module sets. Hence we first describe how we obtain a basic HydLa program from an original HydLa program.

3.2.1 Solving Constraint Hierarchy

A basic HydLa program is obtained from a HydLa program by translating the specification of priorities between individual constraint modules in Fig. 3.1

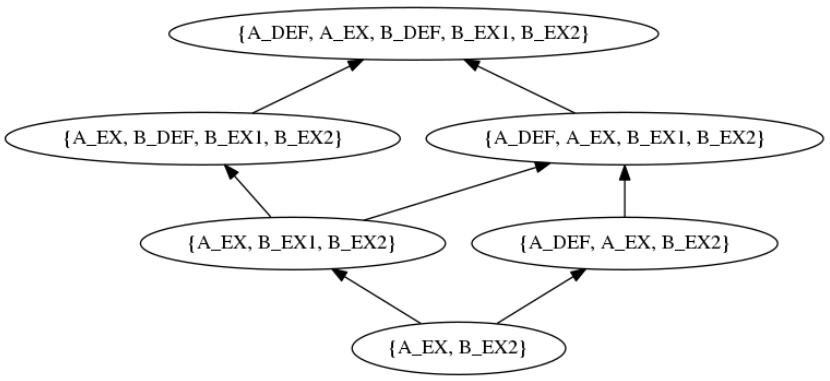

{A_DEF, A_EX, B_DEF, B_EX1, B_EX2}
{A_EX, B_DEF, B_EX1, B_EX2}
{A_DEF, A_EX, B_EX1, B_EX2}
{A_EX, B_EX1, B_EX2}
{A_DEF, A_EX, B_EX2}
{A_EX, B_EX2}

Figure 3.4: Example of constraint hierarchy

into a partially ordered set of the subsets of modules that observe the priority specification, where the partial order is given by subset inclusion. Each subset *MS* is called a *candidate module set* and it must satisfy the following conditions.

$$\forall M_1, M_2((M_1 \ll M_2 \wedge M_1 \in MS) \Rightarrow M_2 \in MS) \tag{3.1}$$

$$\forall M(\neg\exists(R \ll M) \Rightarrow R \in MS) \tag{3.2}$$

The intuitive meaning of Condition 3.1 is that if a module M_1 is contained in *MS* then all modules with higher priority than M_1 are also contained in *MS*. The intuitive meaning of Condition 3.2 is that if a module R has no modules with higher priority than R, R is necessarily contained in *MS*.

A basic HydLa program is a minimum partially ordered set that contains all candidate module sets that satisfy Conditions 3.1 and 3.2.

Example 8. *Consider a declaration of the following constraint hierarchy.*

$$A\_DEF \ll A\_EX, B\_DEF \ll B\_EX1 \ll B\_EX2.$$

In this hierarchy, A_EX and B_EX2 are contained in all candidate module sets because of Condition 3.2. B_DEF can be adopted only if B_EX1 is in the module set. As a result, we obtain the partially ordered set in Fig. 3.4.

3.2.2 Semantics of Basic HydLa

In this section, we define the semantics of basic HydLa by defining the relation between a trajectory and a basic HydLa program.

We regard constraints in a HydLa program as functions w.r.t. time, that is, $C(0) = C, C(t) = \{\}$ $(t > 0)$. We identify a set of constraints with a conjunction of them so that C_1, C_2 is equivalent to $C_1 \cup C_2$. For $C(t)$, we introduce $\Box$-closure $C^*(t)$ such that

- $\forall t(C(t) \subseteq C^*(t))$
- $\forall t(\Box a \in C^*(t) \Rightarrow \forall t' \geq t(a \subseteq C^*(t')))$
- $C^*(t)$ is the minimum set that satisfies above two conditions w.r.t. every t

As we can see, the intuitive meaning of $C^*(t)$ is a set of constraints that are valid at each time point. For example, let $C = \{x = 10, x' = 0, \Box(x'' = -10)\}$, then $C^*(0) = \{x = 10, x' = 0, x'' = -10, \Box(x'' = -10)\}$, $C^*(t) = \{x'' = -10\}$.

Figure 3.5 shows the semantics of basic HydLa, which is in the form of relation between a basic HydLa program P and a pair of a trajectory $\bar{x}$ and the following function $Q(M)(t)$. $Q(M)(t)$ is a set of *expanded* constraints for each module M at each t. A constraint is expanded in $Q(M)(t)$ iff the constraint has no guard or the guard of the constraint is entailed (described at lines $s3$). $Q(M)(t)$ is necessary to handle guarded constraints whose consequents involve "$\Box$" operators because the validness of such consequents are dependent on which guards are entailed in the past.

Example 9. *Consider* $Q_1(M_1)(t) = \{\Box(y- = 0 \Rightarrow \Box(y' = 0))\}$ $(0 < t < t_1)$. *If the guard* $y- = 0$ *is entailed at* t_1, *the constraint* $\Box(y' = 0)$ *is newly expanded, and* $Q_1(M_1)(t_1) = \{\Box(y- = 0 \Rightarrow \Box(y' = 0)), \Box(y' = 0)\}, Q_1(M_1)(t) \supseteq \{\Box(y- = 0 \Rightarrow \Box(y' = 0)), \Box(y' = 0)\}$ $(t > t_1)$ *holds.*

Condition (i) in Fig. 3.5 requires that $Q(M)$ is a $\Box$-closure of $Q(M)$ itself. Condition (ii) requires that $Q(M)$ is a $\Box$-closure for each M. The line $(s0)$ in Condition (iii) describes that a selected candidate module set can change at each t. The line $(s1)$ requires that the trajectory $\bar{x}(t)$ satisfies the selected modules. The lines $(s2)$ requires that there is no "better" trajectory $\bar{x}'(t)$ that is equivalent to $\bar{x}(t)$ before t and satisfies a module set with higher priority than $\bar{x}(t)$ at t. Here, if there is no consistent candidate module set with higher priority than a consistent candidate module set *MS*, *MS* is called a *maximal consistent module set.*

$$
\begin{array}{lr}
\langle \bar{x}(t), Q\rangle \models P \Leftrightarrow (i) \wedge (ii) \wedge (iii) \wedge (iv) \ \text{ wherein} & \\
\quad (i)\ \forall M \in P(Q(M) = Q(M)^*) & \\
\quad (ii)\ \forall M \in P(M^* \subseteq Q(M)) & \\
(iii)\ \forall t \exists E \in P(& (s0) \\
\qquad (\bar{x}(t) \models \{Q(M)(t) \mid M \in E\}) & (s1) \\
\qquad \wedge \neg \exists \bar{x}' \exists E' \in P(& (s2) \\
\qquad\quad \forall t' < t(\bar{x}'(t') = \bar{x}(t')) \wedge E \prec E' & (s2) \\
\qquad\quad \wedge\, \bar{x}'(t) \models \{Q(M)(t) \mid M \in E'\}) & (s2) \\
\qquad \wedge \forall d \forall e \forall M \in E((\bar{x}(t) \Rightarrow d) \wedge ((d \Rightarrow e) \in Q(M)(t)) & (s3) \\
\qquad\quad \Rightarrow e \subseteq Q(m)(t))) & (s3) \\
(iv)\ Q(M)(t) \text{ is the minimum set that satisfies} (i) - (iii) &
\end{array}
$$

Figure 3.5: The definition of $\langle \bar{x}(t), Q\rangle \models P$

3.3 Example Program

Figure 3.6 shows an overview of an example model cited from [7], and Fig. 3.7 shows a HydLa program corresponding to the model. There are two water tanks. The water of the first tank flows into the second tank through a pipe. The variables `x1` and `x2` represent the water level of each tank, while `v1` and `v2` represent the state of the valve in each tank, where v$i = 1$ (0) means that the valve is open (closed), respectively.

The constraint `INIT` describes the initial state of the system. In this model, the initial water level of the first tank is uncertain but satisfies $1.9 \leq$ `x1` ≤ 1.9001. The constraints `X1` and `X2` are about the continuous behavior of each water level. The variable `x1'` denotes the time derivative $dx1/dt$. The constraints `V1_CONST` and `V2_CONST` state that `v1` and `v2` are constant, which is the default behavior. `V1_OFF2ON`, `V1_ON2OFF`, `V1V2_OFF2ON` and `V2_ON2OFF` are constraints for opening and closing of valves. The post-fix minus sign in `x1-` denotes the left hand limit $\lim_{t_l \uparrow t} x1(t_l)$. The last three lines declare the priorities of all these constraints by introducing partial order using `<<`. They say that the states of the valves are constant except when the water level reaches the thresholds. Constraints other than Vi`_CONST` are always enabled because they have no modules with higher priorities.

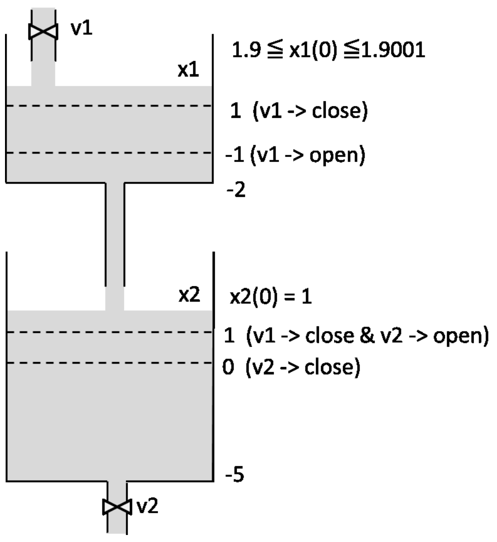


Figure 3.6: Model overview of two water tanks

```
INIT <=> 1.9 <= x1 <= 1.9001 /\ x2 = 1 /\ v1 = 0 /\ v2 = 1.
X1 <=> []((v1 = 0 => x1' = -x1 - 2) /\ (v1 = 1 => x1' = -x1 + 3)).
X2 <=> []((v2 = 0 => x2' = x1) /\ (v2 = 1 => x2' = x1 - x2 - 5)).
V1_CONST  <=> [](v1' = 0).
V2_CONST  <=> [](v2' = 0).
V1_OFF2ON   <=> [](v1- = 0 /\ x1- = -1 => v1 = 1).
V1_ON2OFF   <=> [](v1- = 1 /\ v2- = 1 /\ x1- = 1 => v1 = 0).
V1V2_OFF2ON <=> [](v2- = 0 /\ x2- = 1 => v2 = 1 /\ v1 = 0).
V2_ON2OFF   <=> [](v2- = 1 /\ x2- = 0 => v2 = 0).

INIT, X1, X2,
  (V1_CONST, V2_CONST)
  << (V1_OFF2ON, V1_ON2OFF, V1V2_OFF2ON, V2_ON2OFF).
```

Figure 3.7: HydLa program of two water tanks

3.4 List Notation

In modeling of hybrid systems, we often come across necessity to introduce multiple similar objects. We provide HydLa with a list notation to easily describe models with multiple objects. The syntax with the list notation is shown in Fig. 3.8. We introduce two types of lists here.

The first type is a list of priority definitions (*priority list*, *PL*). A priority list can be denoted by an extensional notation of the form $\{MP_1, MP_2, \ldots, MP_n\}$ or an intensional notation of the form $\{MP \mid LC_1, LC_2, \ldots, LC_n\}$. For example, `{INIT(i) | i in {1,2,3,4}}` is equivalent to `{INIT(1),INIT(2), INIT(3),INIT(4)}`. If a HydLa program includes declarations of priority lists, the elements of the lists are expanded, that is, a declaration of $\{A, B, C\}$ is equivalent to A, B, C.

The second type is a list of arithmetic expressions (*expression list*, *EL*). We can denote an expression list in an extensional or intensional notation as well as a priority list. In addition, we can use range expressions in the form of $\{RE .. RE\}$. RE is an arithmetic expression without variables or an arithmetic expression with a variable whose name terminates with a number such as `x0` and `y1`.

Example 10. *An expression list* `{1*2+1..5}` *is equivalent to* `{3,4,5}`*. An expression list* `{j | i in {1,2}, j in {i+1..4}}` *is equivalent to* `{2,3,4,3, 4}`*. An expression list* `{x1..x3}` *is equivalent to* `{x1, x2, x3}`*.*

We can access the n-th element of a list L by $L[n]$. The index allows an arbitrary expression that results in an integer. The size of a list L is denoted by $|L|$, which can be used as a constant value in a HydLa program. Figure 3.9 shows an example program with the list notation. The program expresses one-dimensional billiard with 10 balls. Note that a HydLa program with the list notation can be statically transformed into a HydLa program without the list notation. In our implementation (Chapter 5), the list notation is preprocessed and expanded in advance of simulation.

$$
\begin{array}{rlcl}
\text{(hydla program)} & P & ::= & (DF \mid DC)^* \\
\text{(definition)} & DF & ::= & \mathit{MPname}(\vec{X})\{MP\} \mid \mathit{cname}(\vec{X}) \Leftrightarrow C \\
 & & & \mid \mathit{ELname} := EL \mid \mathit{PLname} := PL \\
\text{(constraint)} & C & ::= & A \mid C \wedge C \mid \mathit{cname}(\vec{E}) \\
\text{(list condition)} & LC & ::= & \mathit{MPname} \text{ in } PL \mid \mathit{Iterator} \text{ in } EL \\
 & & & \mid E \neq E \\
\text{(priority list)} & PL & ::= & \{MP\ (, MP)^*\} \mid \{MP \mid LC\ (, LC)^*\} \\
 & & & \mid \mathit{PLname} \\
\text{(module priority)} & MP & ::= & M \mid \mathit{MPname}(\vec{E}) \mid MP, MP \\
 & & & \mid MP \ll MP \\
\text{(guard)} & G & ::= & A \mid G \wedge G \mid G \vee G \\
\text{(atomic constraint)} & A & ::= & E\ RO\ E \\
\text{(relational operator)} & RO & ::= & = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
\text{(expression)} & E & ::= & E\ AO\ E \mid P \mid \mathit{constant} \mid EL[E] \\
\text{(expression list)} & EL & ::= & \{E\ (, E)^*\} \mid \{E \mid LC (, LC)^*\} \\
 & & & \mid \mathit{ELname} \mid \{RE\, ..\, RE\} \\
\text{(arithmetic operator)} & AO & ::= & + \mid - \mid \times \mid \div \mid \hat{} \\
\text{(previous)} & P & ::= & D \mid D- \\
\text{(derivative)} & D & ::= & \mathit{vname} \mid \mathit{vname}' \\
\text{(declaration)} & DC & ::= & M \mid DC, DC \mid DC \ll DC \\
 & & & \mid \mathit{dname}(\vec{E}) \mid PL \mid PL[E] \\
\text{(module)} & M & ::= & C
\end{array}
$$

Figure 3.8: Syntax of HydLa with list notation

```
INIT(b, b0, vb0) <=> b = b0 /\ b' = vb0.
COL(b1, b2 )     <=> [](b1- = b2- => b1' = b2'- /\ b2' = b1'-).
CONST(b)         <=> [](b'' = 0).

X := {x0..x9}.
INITS := { INIT(X[i], 2*i-2, 0) | i in {2..|X|} }.
COL_HIERARCHY := { (CONST(X[i]), CONST(X[j]) ) << COL(X[i], X[j])
                        | i in {1..|X|-1}, j in {i+1..|X|} }.

INIT(X[1], 0, 1), INITS, COL_HIERARCHY.
```

Figure 3.9: A model of one-dimensional billiard with list notation

Chapter 4

Simulation Algorithm of Hybrid Systems

In this chapter, we introduce a simulation algorithm of hybrid systems. This simulation algorithms takes a basic HydLa program as input and computes trajectories of the HydLa program symbolically. First, we describe an algorithm that does not adopt interval arithmetic, that is, a fully symbolic algorithm. This algorithm performs symbolic execution of HydLa programs in which uncertainties of input models are handled as symbolic parameters. It is based on consistency checking of conjunctions of constraints and optimization techniques. The output of the algorithm is a set of all possible trajectories of the given program. The resultant set may consist of qualitatively different trajectories caused by the branching of the model's behavior. Such information is useful in performing accurate case analysis of HydLa models.

After that, we describe an algorithm that integrates interval arithmetic with the fully symbolic algorithm. This version computes an over-approximation of a set of possible trajectories by computing the enclosures of symbolic formulas.

4.1 Symbolic Simulation

Figure 4.1 shows the algorithm of the fully symbolic simulation. This algorithm takes a basic HydLa program as input and computes a set of trajectories as output. The resultant set of trajectories are computed implicitly from the sets of constraints that are enabled at each phase of the simulation. In this algorithm, we use a function named *GetElement* to express nondeterministic points of the procedure. *GetElement* nondeterministically chooses one element from a given set with which we continue simulation. To compute all possible trajectories, we trace all possibilities about each call to *GetElement*. The algorithm alternates two phases: Point Phase (PP) for handling discrete changes and Interval Phase (IP) for handling continuous evolutions. In this algorithm, we compute a *constraint store* S on each phase. A constraint store is a conjunction of constraints that has to be satisfied. E denotes a set of expanded consequents with "$\Box$" operators such as the consequent $\Box(y = 0)$ of a constraint $x = 0 \Rightarrow \Box(y = 0)$. Elements of E are added when the guards are entailed for the first time in simulation.

In each phase, we compute a maximal consistent set (MCS) of constraints in accordance with the declarative semantics of HydLa and put the set of constraints into S and conditions on symbolic parameters into P (lines 10 and 16). *MCS* is a higher-order function that takes a consistency-checking function (see Section 4.1.2 and Section 4.1.3) for the corresponding phase. If S is *false* as a result of *MCS* in lines 12 and 19, we can conclude that given HydLa program has no further valid trajectory because there are no consistent module sets. *Subst* in line 9 is to reflect the current time into the current constraint store S.

At the end of a Point Phase, we introduce parameters corresponding to uncertain values of variables at that time point by *AddParameters*. In an Interval Phase, we compute an MCS again, this time obtaining a set A_+ of enabled guards and a set A_- of disabled guards as well (line 16). Then we solve differential equations analytically in S (line 17) and compute the time of the next discrete change, called events (line 22). *Subst* reflects the solutions of ODEs in S into g and $\neg g$. In general, there may exist multiple candidates of the time of events due to uncertainties of parameter values. For example, consider the case where the trajectory of $x(t)$ is described by formulas $x(t) = -(t-1)^2 + x(0)$, $-1.5 \leq x(0) \leq 1.5$, and events occur when $x(t) = 1 \ \vee \ x(t) = -10$. In this case, two candidates of the time of events exist. The first candidate is $t = 1 - \sqrt{x(0) - 1}$ with $1 \leq x(0) \leq 1.5$ and the second candidate is $t = 1 - \sqrt{x(0) - 9}$ with $-1.5 \leq x(0) < 1$. Such multiple candidates appear only if the condition of parameters include a kind of corner cases (e.g., a ball touches a roof but does not collide with the roof). *FindMinTime* computes the earliest time when the given constraint is

```
Input: $HydLa$: basic HydLa program,
       $MaxT$: maximum simulation time
1: $MS := TopologicalSort(HydLa)$ // list of candidate sets of constraints
2: $V := GetVariables(HydLa)$
3: $T := 0$    // current time
4: $S := true$ // current constraint store
5: $P := true$ // constraints on symbolic parameters
6: $E := \emptyset$ // expanded consequents
7: while $T < MaxT$ do
8:    // Point Phase (PP)
9:    $S := Subst(S, T)$
10:   $(S, P, E, \_, \_) := MCS(S, MS, E, P, T, CheckConsistencyPP)$
11:   if $S = false$ then
12:      break
13:   end if
14:   $(S, P) := AddParameters(S, P, V)$
15:   // Interval Phase (IP)
16:   $(S, P, E, A_-, A_+) := MCS(S, MS, E, P, T, CheckConsistencyIP)$
17:   $S := SolveDifferentialEquation(S)$
18:   if $S = false$ then
19:      break
20:   end if
21:   $(MinT, P) := GetElement(CompareMinTime($
        $(\bigcup_{(g \Rightarrow c) \in A_-} FindMinTime(Subst(g, S), P))$
        $\cup (\bigcup_{(g \Rightarrow c) \in A_+} FindMinTime(Subst(\neg g, S), P))$
        $\cup \{(MaxT - T, true)\}))$
22:   $T := MinT + T$
23: end while
```

Figure 4.1: Algorithm of the symbolic simulation

satisfied using constraint optimization techniques. *FindMinTime* returns a set of pairs of candidate time and the condition of symbolic parameters where the candidate time is the earliest time. The specification of *FindMinTime* can be described as follows.

$$\begin{aligned}
FindMinTime(C, P) = \{(t_m, P_{new}) \mid\ & t_m > 0 \land (P_{new} \Rightarrow P) \\
& \land \forall p \in assignments(P_{new})(consistent(C \land t = t_m \land V_p = p)) \\
& \land \neg \exists t_o(0 < t_o < t_m \\
& \quad \land \exists p \in assignments(P_{new})(consistent(C \land t = t_o \land V_p = p)))\}
\end{aligned}$$

wherein V_p is a tuple of symbolic parameters.

CompareMinTime takes a set of results of *FindMinTime* as input and chooses the earliest time under each condition of parameters. The specification of *CompareMinTime* can be described as follows.

$$\begin{aligned}\mathit{CompareMinTime}(S) = \{(t_m, P) \mid {} & \exists (t_m, P_1) \in S((P \Rightarrow P_1) \\ & \wedge \neg\exists (t_2, P_2) \in S(t_2 < t_m \wedge \mathit{consistent}(P \wedge P_2)))\}\end{aligned}$$

If we have multiple candidates as a result of *CompareMinTime*, *GetElement* chooses one element from those candidates and simulation continues.

4.1.1 Calculation of Maximal Consistent Set

Figure 4.2 shows the algorithm of *MCS*. This function computes the maximal consistent set of each phase. We check the consistency of elements in *MS* from the largest one by *CalculateClosure*. The return value includes additional information about the maximal consistent set such as a set of entailed guards and so on.

4.1.2 Consistency Checking in Point Phase

Figure 4.3 shows the algorithm of consistency checking in PP. First we compute the set V of variables that appear in the constraint store. Next we check if there is any possible assignment to the variables that satisfies the constraint store S and the constraint P on the symbolic parameters at the same time (line 2). For this purpose, we solve logical formulas with an existential quantifier by quantifier elimination. The resultant constraint P_{tmp} corresponds to conditions of symbolic parameters in which the constraint store is satisfiable. If such a condition does not exist (in the case where $P_{tmp} = \mathit{false}$) this function returns *false* and the original constraints on symbolic parameters P. If such a condition exists, we check if P_{tmp} is equivalent to P. If they are equivalent, the constraint store is consistent in all cases, therefore we return *true* and the original constraint on symbolic parameters P. If they are not equivalent, the consistency of the constraint store depends on the condition of symbolic parameters, therefore the result branches into two cases (line 8).

```
Input: S: constraint store, MS: list of candidate constraint sets,
    E: set of expanded always consequents,
    P: constraint on symbolic parameters,
    T: current time, CheckConsistency: function for consistency checking
Output: constraint store, constraint on symbolic parameters, set of expanded
    always, maximal consistent set, set of not entailed guards, set of entailed
    guards
 1: for $M \in MS$ do
 2:   if $T > 0$ then
 3:     $M := EliminateNotAlways(M)$
 4:   end if
 5:   $(S_{tmp}, E_{tmp}, P, A_-, A_+) :=$
 6:     $CalculateClosure(S, M, P, E, CheckConsistency)$
 7:   if $S_{tmp} \neq false$ then
 8:     return $(S_{tmp}, P, E_{tmp}, M, A_-, A_+)$
 9:   end if
10: end for
11: return $(false, P, E, \emptyset, \emptyset, \emptyset)$
```

Figure 4.2: Algorithm of *MCS*

4.1.3 Consistency Checking in Interval Phase

Figure 4.4 shows the algorithm of *CheckConsistencyIP*. This function is similar to *CheckConsistencyPP* except two differences. The first difference is that it solves ODEs in S symbolically and obtain a constraint store S_t w.r.t. time. The second difference is that it checks satisfiability not on the time point but in the right neighborhood of the time point (line 3). *Inf* means the infimum of a given set.

4.1.4 Calculation of Closure of Constraints

Figure 4.5 shows the algorithm of *CalculateClosure*. This function computes closures of the given constraint store. Such closures can be obtained by repeating

- consistency checking (line 4),
- checking the entailment of guards (lines 11–17), and
- addition of the consequents of guards that are entailed (lines 19–20)

```
Input: $S$: constraint store, $P$: constraint on symbolic parameters
Output: consistency of $S$, new constraint on symbolic parameters
 1: $V := \mathit{GetVariables}(S)$
 2: $P_{tmp} := \exists V(S \land P)$
 3: if $P_{tmp} = \mathit{false}$ then
 4:     return $(\mathit{false}, P)$
 5: else if $P_{tmp} = P$ then
 6:     return $(\mathit{true}, P)$
 7: else
 8:     return $\mathit{GetElement}(\{(\mathit{true}, P_{tmp}), (\mathit{false}, P \land \neg P_{tmp})\})$
 9: end if
```

Figure 4.3: Algorithm of *CheckConsistencyPP*

```
Input: $S$: constraint store, $P$: constraint on symbolic parameters
Output: satisfiability of $S$, new constraint on symbolic parameters
 1: $S_t := \mathit{SolveDifferentialEquation}(S)$
 2: $V := \mathit{GetVariables}(S_t)$
 3: $P_{tmp} := \exists V(\mathit{Inf}\{t \mid \exists_t(S_t \land t > 0)\} = 0 \land P)$
 4: if $P_{tmp} = \mathit{false}$ then
 5:     return $(\mathit{false}, P)$
 6: else if $P_{tmp} = P$ then
 7:     return $(\mathit{true}, P)$
 8: else
 9:     return $\mathit{GetElement}(\{(\mathit{true}, P_{tmp}), (\mathit{false}, P \land \neg P_{tmp})\})$
10: end if
```

Figure 4.4: Algorithm of *CheckConsistencyIP*

until the constraint store reaches a fixed point (line 23, $\neg$*Expanded*). If there is a guard whose entailment cannot be determined, we put the guarded constraint into *BranchedAsk* temporarily (line 15). If the entailment remains undetermined until the loop terminates, we analyze two cases (lines 24–39). In the first case, the guard of *BranchedAsk* is assumed to be entailed (lines 26–27), and in the second case, the guard of *BranchedAsk* is assumed to be not entailed (lines 28–29). If both cases are consistent, the simulation branches into two cases (lines 30–31).

```
Input: $S_{prev}$: constraints on variables in the previous phase, $M$: module set
    whose consistency is to be checked, $P$: constraint on symbolic parame-
    ters, $E$: set of expanded consequents, $CheckConsistency(S)$: function for
    consistency checking
Output: new constraint store, new set of expanded consequents, new con-
    straint on symbolic parameters, set of not entailed guards, set of entailed
    guards
 1: $(A_+, A_-) := CollectAsk(M, E)$;
 2: repeat
 3:   $S := CollectTell(M, A_+, S_{prev})$;
 4:   $(TF, P) := CheckConsistency(S, P)$
 5:   if $TF = false$ then
 6:     return $(false, \emptyset, P, \emptyset, \emptyset)$
 7:   end if
 8:   $Expanded := false$
 9:   $BranchedAsk := \emptyset$
10:   for $(g \Rightarrow c) \in A_-$ do
11:     $(TF, P) := CheckConsistency(S \wedge g, P)$
12:     if $TF \neq false$ then
13:       $(TF, P) := CheckConsistency(S \wedge \neg g, P)$
14:       if $TF \neq false$ then
15:         $BranchedAsk := (g \Rightarrow c)$
16:         continue
17:       end if
18:       $Expanded := true$
19:       $(A_-, A_+, E) :=$
20:       $ExpandAsk(A_-, A_+, E, (g \Rightarrow c))$
21:     end if
22:   end for
23: until $\neg Expanded$
24: if $BranchedAsk \neq \emptyset$ then
25:   $g := GetGuard(BranchedAsk)$
26:   $(S_{tr}, E_{tr}, P_{tr}, A_{-tr}, A_{+tr}) :=$
27:     $CalculateClosure(S \wedge g, M, P, E, CheckConsistency)$
28:   $(S_{fa}, E_{fa}, P_{fa}, A_{-fa}, A_{+fa}) :=$
29:     $CalculateClosure(S \wedge \neg g, M, P, E, CheckConsistency)$
30:   if $S_{tr} \neq false \wedge S_{fa} \neq false$ then
31:     return $GetElement(\{$
          $(S_{tr}, E_{tr}, P_{tr}, A_{-tr}, A_{+tr}), (S_{fa}, E_{fa}, P_{fa}, A_{-fa}, A_{+fa})\})$
32:   else if $S_{tr} \neq false$ then
33:     return $(S_{tr}, E_{tr}, P_{tr}, A_{-tr}, A_{+tr})$
34:   else if $S_{fa} \neq false$ then
35:     return $(S_{fa}, E_{fa}, P_{fa}, A_{-fa}, A_{+fa})$
36:   else
37:     return $(false, \emptyset, P, \emptyset, \emptyset)$
38:   end if
39: end if
40: return $(S, E, P, A_-, A_+)$
```

Figure 4.5: Algorithm of *CalculateClosure*

```
INIT <=> 9 <= y /\ y <= 11 /\ y' = 10.
FALL <=> [](y'' = -10).
BOUNCE <=> [](y- = 15 => y' = -4/5 * y'-).

INIT, (FALL << BOUNCE).
```

Figure 4.6: HydLa program of throwing up a ball

4.1.5 Example of Symbolic Simulation

As an example of the symbolic simulation algorithm shown in Fig. 4.1, we follow the execution of the program shown in Fig. 4.6 with *MaxT* (> 1). *MS*, M_{all} and V are $\{\{\texttt{INIT}, \texttt{FALL}, \texttt{BOUNCE}\}, \{\texttt{INIT}, \texttt{BOUNCE}\}\}$, $\{\texttt{INIT}, \texttt{FALL}, \texttt{BOUNCE}\}$ and $\{\texttt{y}, \texttt{y'}, \texttt{y''}\}$, respectively.

The procedure enters the first PP. *Subst* substitutes the current time T ($= 0$) into the constraint store S. However, S is empty hence it has no effect. In this PP, the maximal consistent set is $\{\texttt{INIT}, \texttt{FALL}, \texttt{BOUNCE}\}$ and we obtain $9 \leq \texttt{y} \leq 11 \wedge \texttt{y'} = 10 \wedge \texttt{y''} = -10$ as the constraint store. Here the value of y has uncertainty, therefore we introduce a symbolic parameter *py* that corresponds to the value of y at the initial time. We add a constraint $9 \leq py \leq 11$ into P and modify S to $\texttt{y} = py \wedge \texttt{y'} = 10 \wedge \texttt{y''} = -10$.

Next, the procedure enters IP. The maximal consistent set in this phase is $\{\texttt{INIT}, \texttt{FALL}, \texttt{BOUNCE}\}$, which is the same as the previous one. Note that INIT and BOUNCE have no effect because INIT is not on the initial time and the guard of BOUNCE is not entailed, respectively. The corresponding constraint store is computed as $\texttt{y(0)} = py \wedge \texttt{y'(0)} = 10 \wedge \texttt{y''(t)} = -10$. *SolveDifferentialEquation* solves this constraint store and obtains $\texttt{y(t)} = py + 10\texttt{t} - 5\texttt{t}^2 \wedge \texttt{y'(t)} = 10 - 10\texttt{t} \wedge \texttt{y''(t)} = -10$. In the computation of *FindMinTime* with S and P, there are two candidates of the time of the collision depending on the condition of the parameter. The first candidate is $1 - \sqrt{py/5 - 2}$ where $10 \leq py \leq 11$ and the second candidate is ∞ (the ball does not reach the roof) where $9 \leq py < 10$. *CompareMinTime* compares these two candidates with $MaxT - T$ ($= MaxT$). If $py \geq 10$ holds, then $1 - \sqrt{py/5 - 2}$ is put into *MinT* because $MaxT - T$ is greater than $1 - \sqrt{py/5 - 2}$, else *MaxT* is put into *MinT*. In this section, we follow the case where $py \geq 10$ holds. T equals $1 - \sqrt{py/5 - 2}$ and it does not reach *MaxT*, therefore the simulation goes into the next PP.

The next PP corresponds to the contact of the ball and the roof. In the computation of *MCS*, the condition of py determines whether `FALL` and `BOUNCE` contradict or not. If py is exactly equal to 10, the ball only touches the roof and the velocity does not change discretely. On the other hand, if $py > 10$ holds, `FALL` and `BOUNCE` contradict and `FALL` is removed from the constraint store because of its weak priority. Here, we follow the case where $py > 10$. The maximal consistent set is $\{\texttt{INIT}, \texttt{BOUNCE}\}$ and the resultant constraint store is $\texttt{y} = 15 \wedge \texttt{y'} = -10\sqrt{py/5 - 2}$.

Afterwards, the simulation enters the next IP. In this example, no further discrete change occurs, hence this IP is the last phase of the simulation. The enabled constraints of this IP is the same as those of the preceding IP except the constraints about the initial value of `y`. As a result, we obtain the constraint store $\texttt{y(t)} = 15 - 10\sqrt{py/5 - 2}\texttt{t} - 5\texttt{t}^2 \wedge \texttt{y'(t)} = 10\sqrt{py/5 - 2} - 10\texttt{t} \wedge \texttt{y''(t)} = -10$.

4.2 Symbolic Simulation with Interval Arithmetic

In this section, we describe the algorithm that integrates interval arithmetic with the symbolic method we described in the previous section. Figure 4.7 shows the algorithm, which is a modified version of Fig. 4.1. There are three differences from the original symbolic algorithm.

The first difference is that we compute enclosures of the ranges of symbolic formulas by *Enclose* at line 15. This computation is performed by affine arithmetic and comes with choices regarding how many noise symbols to preserve in the reduction of noise symbols. We compare the influence of the choice to the performance in Chapter 6. In this algorithm, all symbolic parameters (say p_i's) in P are normalized to satisfy $p_i \in [-1, 1]$, with which an uncertain quantity whose bounds are $\underline{x}$ and $\overline{x}$ can be represented as $(\overline{x} + \underline{x})/2 + (\overline{x} - \underline{x})/2 \times p_i$.

Example 11. *If an uncertain quantity $x = [-5/2, 1/2]$ appears in the simulation, we introduce a new symbolic parameter p_{n+1} and express the quantity by $x = 1 + (3/2)p_{n+1}$.*

By this transformation, we can directly handle such symbolic parameters as noise symbols in affine arithmetic because the range of them are the same.

```
Input: $HydLa$: basic HydLa program,
       $MaxT$: maximum simulation time
 1: $MS := TopologicalSort(HydLa)$ // list of candidate sets of constraints
 2: $V := GetVariables(HydLa)$
 3: $T := 0$     // current time
 4: $S := true$ // current set of constraints
 5: $P := true$ // constraints on symbolic parameters
 6: $G_p := \emptyset$ // guards that caused the previous event
 7: $E := \emptyset$ // expanded always consequents
 8: while $T < MaxT$ do
 9:    // Point Phase (PP)
10:    $S := Subst(S, T)$
11:    $(S, P, E, \_, \_) := MCS(S@G_p, MS, E, P, T, CheckConsistencyPP)$
12:    if $S = false$ then
13:       break
14:    end if
15:    $(S, P) := Enclose(AddParameters(S, P, V))$
16:    // Interval Phase (IP)
17:    $(S, P, E, A_-, A_+) := MCS(S@G_p, MS, E, P, T, CheckConsistencyIP)$
18:    $S := SolveDifferentialEquation(S)$
19:    if $S = false$ then
20:       break
21:    end if
22:    $(MinT, P, G_p) := GetElement(CompareMinTime($
         $(\bigcup_{(g \Rightarrow c) \in A_-} FindMinTimeInterval(Subst(g, S), P, G_p))$
         $\cup\ (\bigcup_{(g \Rightarrow c) \in A_+} FindMinTimeInterval(Subst(\neg g, S), P, G_p))$
         $\cup\ \{(MaxT - T, true)\}))$
23:    $T := MinT + T$
24: end while
```

Figure 4.7: Algorithm of the symbolic simulation with interval arithmetic

The second difference is that we pay attention to guards that cause the discrete event, which are denoted by G_p. We pay attention to G_p at the calls to *MCS* and *FindMinTimeInterval* because the interval solutions are not exact ones and naïve handling of such non-exact solutions leads to redundant case branching in the symbolic simulation (while the original algorithm in Fig. 4.1 assumes that we exploits exact analytic solutions). By the notation "$S@G_p$", we denote a constraint store S with an assumption about G_p (lines 11 and 17), which means that we can use G_p as a necessarily satisfied constraint in the consistency checking of S. We describe how G_p is handled in

```
Input: $G$: the guard represented as a constraint on $t$ using the solution of ODEs,
    $P$: parameter conditions,
    $G_p$: guards that caused the previous event
Output: $t_{min}$: minimum time at which the consistency of the guard changes,
    $g_{list}$: atomic guards whose consistency changes at $t_{min}$
 1: $g_{list}$ := $GetAtomicBoundaryConditions(G)$
 2: $t_{list} := \emptyset$; $Map_g := \emptyset$
 3: for $g_b \in g_{list}$ do
 4:   $(sols, P) := ZeroCrossings(g_b, P, G_p)$
 5:   for $sol \in sols$ do
 6:     $t_{list}.add(sol, g_b)$
 7:   end for
 8:   $Map_g.insert(g_b, ConsistentAtInitialTime(g_b, G_p))$
 9: end for
10: while $t_{list} \neq \emptyset$ do
11:   $(t_{min}, g_{list}) := PopMinTime(t_{list}, P)$
12:   $(Map_g, satisfied) := CheckAndUpdateGuards(Map_g, g_{list}, G)$
13:   if $satisfied = true$ then
14:     $break$
15:   end if
16: end while
```

Figure 4.8: Algorithm of *FindMinTimeInterval*

FindMinTimeInterval later (Section 4.2.1).

The third difference is that we compute the time of discrete changes using the interval Newton method and the mean value theorem in *FindMinTimeInterval.* The specification of *FindMinTimeInterval* can be described as follows.

$$\begin{aligned}&\mathit{FindMinTimeInterval}(C, P, G_p) = \{(t_m, P_{new}) \mid 0 \notin \phi(t_m, P_{new}) \wedge (P_{new} \Rightarrow P)\\&\quad \wedge \exists p \in \mathit{assignments}(P)(\mathit{consistent}(C \wedge t = t_m \wedge V_p = p))\\&\quad \wedge \neg(\exists t_o \exists p \in \mathit{assignments}(P)(t_o < t_m \wedge \mathit{consistent}(C \wedge t = t_o \wedge V_p = p)))\}\end{aligned}$$

wherein V_p is a tuple of symbolic parameters and $\phi(t, P)$ represents a set of possible values of t under P. We describe the procedure of *FindMinTimeInterval* in the next section. Note that *FindMinTimeInterval* does not handle case branching caused by symbolic parameters, while original *FindMinTime* is designed to handle such branching. This is because the interval Newton method used in *FindMinTimeInterval* cannot handle such branching.

4.2.1 Computation of Event Time with Interval Arithmetic

The function *FindMinTimeInterval* computes the time of discrete events with interval arithmetic. In a HydLa program, a discrete change is triggered when the consistency of any guard in the program changes. A guard in a HydLa program is described in the form of a system of equations and inequations. Because we assume that the ODEs have solutions in closed form, a guard can be regarded as a constraint w.r.t time, which is denoted by $G(t)$. Thus, the goal of *FindMinTimeInterval* is to compute the minimum (earliest) time at which the consistency of the given constraint changes (from consistent to inconsistent or vice versa).

Figure 4.8 shows the algorithm of *FindMinTimeInterval*. First, we compute atomic boundary conditions from the given $G(t)$ and substitute it into g_{list} by *GetAtomicBoundaryConditions*. Atomic boundary conditions can be obtained by transposing the right-hand side terms to the left-hand side.

Example 12. *The result of GetAtomicBoundaryConditions*$(t > 0 \wedge t^2 = 1 \wedge t \leq 1)$ *is* $\{t > 0, t^2 - 1 = 0, t - 1 \leq 0\}$.

Now, the time when the consistency of $G(t)$ changes is one of the time points when the elements in g_{list} change their consistency, that is, the zero-crossings of the left-hand sides of the conditions. We compute those zero-crossings of the left-hand side by *ZeroCrossings*. *ZeroCrossings* computes zero-crossings of g symbolically whenever possible. Otherwise, it computes zero-crossings by the method shown in Section 4.2.2 and returns the solutions *sols* and a condition P that includes the conditions on new parameters if any. The zero-crossings are pushed into t_{list} with g. We also create Map_g, a mapping from each guard to the consistency of the guard at the initial time of the current phase (line 8). From line 9 to line 16, we check the consistency of the whole guard at each time interval starting from each element in t_{list}. *PopMinTime* removes the pair whose time is the minimum from t_{list} and returns the minimum time t_{min} and atomic boundary conditions g_{list} that change their consistency at t_{min}. *CheckAndUpdateGuards* returns the consistency of the whole guard in the time interval starting from t_{min} and an updated map Map_g, that is, a mapping for the current time interval.

Figure 4.9 shows the algorithm of *CheckAndUpdateGuards*. From line 3 to line 9, we update the consistency of each atomic condition exactly on

t_{min}. If the relational operator *relop*(g) includes equality (i.e., it is equality or non-strict inequality) it is consistent and otherwise it is inconsistent. At line 10, we check the consistency of the whole guard G. Lines 13 to 24 are a similar process about the open time interval starting from t_{min}. Note that in this case g has already passed the boundary therefore the update process is different from the previous one. If G is satisfied at line 10 or line 24, the start point of the current time interval can be regarded as the time of the event about G.

4.2.2 Computation of Zero-crossings

When *ZeroCrossings* cannot compute zero-crossings symbolically, it computes the solutions using interval techniques. This procedure consists of two steps.

First, we solve the given equation g_b with the interval Newton method [28]. In this method, we narrow the initial interval X_0 step by step by applying

$$X^{(k+1)} = X^{(k)} \cap N(X^{(k)})$$

with the Newton operator

$$N(X) = m(X) - f(m(X))/f'(X),$$

where $m(X)$ is the midpoint of X. The result of the interval Newton method is the fixed point of the operator $N(X)$. It has been proved in [28] that if X_0 includes the exact solution, the width of $X^{(k)}$ converges quadratically. The procedure branches if X_0 includes multiple solutions of g_b, and computes a family of intervals such that each interval is guaranteed [36], thanks to the nice property of the interval Newton method, to include exactly one solution.

One thing to note is that, if g_b belongs to G_p, g_b holds at the start time of the current Interval Phase, and the interval Newton method would persist to an interval including the start time. Since the event at that time point has been already handled in the previous Interval Phase, such a branch should be discarded. This is why *ZeroCrossings* takes G_p as input.

Second, we compute zero-crossings that preserve the first-order terms of parameters using the result of the interval Newton method. Note that in the interval Newton method, the parameters in g_b are replaced by intervals and the resultant zero-crossings discard the dependency between parame-

```
Input: $Map_g$: mapping from each guard to its consistency in the previous time interval,
    $g_{list}$: list of atomic guards whose consistency changes at the current time point,
    $G$: entire guard
Output: $Map_g$: mapping from each guard to its consistency in the current time interval,
    $satisfied$: Consistency of $G$ in the next time interval
 1: $Map_{prev} := Map_g$
 2: $satisfied := false$
 3: for $g \in g_{list}$ do
 4:     if $relop(g) \in \{‘=’, ‘\leq’, ‘\geq’\}$ then
 5:         $Map_g.replace(g, true)$
 6:     else
 7:         $Map_g.replace(g, false)$
 8:     end if
 9: end for
10: if $G.satisfiedBy(Map_g)$ then
11:     $satisfied := true$
12: end if
13: for $g \in g_{list}$ do
14:     if $relop(g) = ‘=’$ then
15:         $Map_g.replace.(g, false)$
16:     else if $relop(g) = ‘\neq’$ then
17:         $Map_g.replace.(g, true)$
18:     else
19:         $Map_g.replace.(g, \neg Map_{prev}.getValue(g))$
20:     end if
21: end for
22: if $G.satisfiedBy(Map_g)$ then
23:     $satisfied := true$
24: end if
```

Figure 4.9: Algorithm of *CheckAndUpdateGuards*

ters, which is why we recompute solutions here. In this step, we compute a symbolic solution that preserves first-order dependency of parameters on the basis of the mean value theorem. This second step has the following specification:

Input: $f(t, \vec{p}) : \mathbb{R} \times [-1, 1]^n \to \mathbb{R}$,
T: time interval computed by the interval Newton method

Output: $T_{result}(\vec{p}) : [-1, 1]^n \to \mathbb{I}$ that encloses the solution $t(\vec{p}) : [-1, 1]^n \to \mathbb{R}$ of the equation $f(t, \vec{p}) = 0$.

For $\vec{P} \in \mathbb{I}^n$, $t \in T$ and $\vec{p} \in \vec{P}$, the multivariate mean value theorem gives

$$\begin{aligned} f(t, \vec{p}) \in f(T_m, \vec{P}_m) + (&\frac{\partial f(T, \vec{P})}{\partial t}, \frac{\partial f(T, \vec{P})}{\partial p_1}, \ldots, \frac{\partial f(T, \vec{P})}{\partial p_n}) \\ &\cdot (t - T_m, p_1 - \vec{P}_{m1}, \ldots, p_n - \vec{P}_{mn}) \end{aligned} \tag{4.1}$$

wherein T_m denotes the midpoint of T and $\vec{P}_m$ denotes the vector whose elements are midpoints of the corresponding elements of $\vec{P}$. The following is a proof of this fact:

Proof. By the mean value theorem, for a continuous and differentiable function $h : \mathbb{R}^n \to \mathbb{R}$ and a closed interval $[a, b] \in \mathbb{I}^n$, there exists $c \in [a, b]$ that satisfies

$$h(b) = h(a) + \nabla h(c) \cdot (b - a).$$

Here, we consider an interval I that satisfies $[a, b] \subseteq I$. Such I satisfies $h(c) \in h([a, b]) \subseteq h(I)$. This gives the condition

$$h(b) \in h(a) + \nabla h(I) \cdot (b - a).$$

By replacing h, b, a and I with f, $(t, \vec{p})$, $(T_m, \vec{P}_m)$ and $(T, \vec{P})$, respectively, we obtain Condition 4.1. □

Because the range of each parameter is normalized to $[-1, 1]$, which means $\vec{P}_m$ is equal to $\vec{0}$, Condition 4.1 can be simplified into

$$\begin{aligned} f(t, \vec{p}) \in &f(T_m, \vec{0}) + (\frac{\partial f(T, \vec{P})}{\partial t}, \frac{\partial f(T, \vec{P})}{\partial p_1}, \ldots, \frac{\partial f(T, \vec{P})}{\partial p_n}) \\ &\cdot (t - T_m, p_1, \ldots, p_n). \end{aligned} \tag{4.2}$$

We define f_{mean} as the right-hand side of Condition 4.2 and solve the equation $f_{mean} = 0$ for t:

$$t = -\frac{(f_{\partial p_1}, \ldots, f_{\partial p_n})}{f_{\partial t}} \cdot \vec{p} + T_m - \frac{f(T_m, \vec{0})}{f_{\partial t}} \tag{4.3}$$

where $f_{\partial t}$ and the $f_{\partial p_i}$'s are new intervals denoting $\partial f(T, \vec{P})/\partial t$ and the $\partial f(T, \vec{P})/\partial p_i$'s, respectively.

If we handle those intervals as symbolic parameters, the costs of computation would grow quickly. To avoid such growth, we introduce only one new interval at each detection of discrete changes. For all $\vec{p} \in \vec{P}$, the following property holds:

$$\begin{aligned}\frac{(f_{\partial p_1}, \ldots, f_{\partial p_n})}{f_{\partial t}} \cdot \vec{p} \in mid(&\frac{f_{\partial p_1}, \ldots, f_{\partial p_n}}{f_{\partial t}}) \cdot \vec{p} \\ &+ [-1, 1] \times \sum_{i=1}^{n} rad(\frac{f_{\partial p_i}}{f_{\partial t}}).\end{aligned} \tag{4.4}$$

Properties (4.3) and (4.4) give the following symbolic solution of the original problem $f(t, \vec{p}) = 0$:

$$\begin{aligned}T_{result}(\vec{p}) = {} & mid(\frac{f_{\partial p_1}, \ldots, f_{\partial p_n}}{f_{\partial t}}) \cdot \vec{p} \\ & + T_m + [-1, 1] \times \sum_{i=1}^{n} rad(\frac{f_{\partial p_i}}{f_{\partial t}}) - \frac{f(T_m, \vec{0})}{f_{\partial t}}\end{aligned} \tag{4.5}$$

which encloses the solution for t parameterized with respect to $\vec{p}$. Here, $[-1, 1] \times \sum_{i=1}^{n} rad(f_{\partial p_i}/f_{\partial t}) - f(T_m, \vec{0})/f_{\partial t}$ is reduced into a single interval by IA. $T_{result}(\vec{p})$ in (4.5) after this parameter reduction is the final output of the procedure, which is in an affine form.

Example 13. *Consider computing the zero-crossing of the input below:*

$$\begin{aligned}&g_b := 2 - exp(t + p_y/1000) + p_x/1000 = 0,\\ &P := -1 \le p_x \le 1 \wedge -1 \le p_y \le 1,\\ &G_p := \emptyset.\end{aligned}$$

In this example, the length of significands is limited to five for simplicity, while the actual implementation is based on the IEEE754 double-precision *floating-point format. First, by the interval Newton method we obtain an interval* $[0.69163, 0.69466]$*. Second, we compute* $T_{result}(\vec{p})$ *as follows:*

$$\begin{aligned}
T_{result}((p_x, p_y)) &= mid(\frac{f_{\partial p_x}}{f_{\partial t}})p_x + mid(\frac{f_{\partial p_y}}{f_{\partial t}})p_y + T_m \\
&\quad + [-1,1] \times (rad(\frac{f_{\partial p_x}}{f_{\partial t}}) + rad(\frac{f_{\partial p_y}}{f_{\partial t}})) - \frac{f(T_m, \vec{0})}{f_{\partial t}} \\
&= 5.0000 \times 10^{-4} \times p_x - 9.9999 \times 10^{-4} \times p_y \\
&\quad +0.69314 + 1.0321 \times 10^{-5} \times [-1,1].
\end{aligned}$$

Chapter 5

HyLaGI: The Implementation

We have implemented the simulation algorithm described in Chapter 4. The implemented simulator is named HyLaGI. We implemented HyLaGI in C++ and used Mathematica as its backend constraint solver. The overview of HyLaGI is shown in Fig. 5.1.

Mathematica is used for the checking consistency of conjunctions of constraints, solving ODEs, solving optimization problems about the time of events and transforming both arithmetic expressions and logical formulas. The class of ODEs that are symbolically solvable with Mathematica is listed in Table 5.1. The use of a general-purpose backend constraint solver is admittedly not advantageous for performance, as shown in Chapter 6, but provided a flexible platform for the combination of various symbolic and interval techniques. The soundness of the current implementation also depends on the soundness of the backend solver that we use as a trusted black box. Further decomposition of the computation process is desired to reduce the granularity of the trusted black box, which is our future work. We use KV library [19] for interval arithmetic and affine arithmetic (including reduction of dummy variables).

The output of HyLaGI is a set of trajectories that satisfy the specification of the given HydLa program. HyLaGI outputs results in the form of human readable text and plot files in JSON. An example of result text is shown in Fig. 5.2, where some parts are omitted for lack of space. This results from the program in Fig. 4.6. The first section with `parameter condition(global)` shows the whole range of the symbolic parameter, that is, $9 <$ `p[y, 0, 1]` < 11. Each symbolic parameter is denoted in the form of `p[`*variable name*, *derivative count*, *phase id*`]`; e.g., `p[y, 0, 1]` denotes

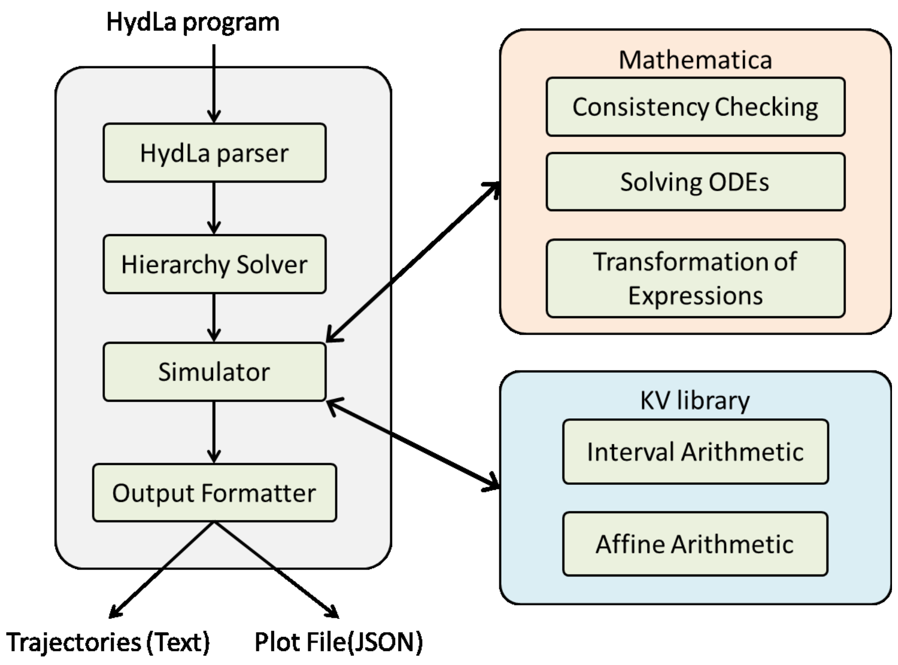


Figure 5.1: Overview of the implementation

a symbolic parameter for the "0"th derivative of `y` (the variable itself) at the "1"st phase. The following lines show the trajectory of the first case. The trajectory of each case goes through Point Phases and Interval Phases alternately. The line starting with `unadpoted modules` shows the set of constraint modules that are not adopted in each phase. The lines starting with `positive` and `negative` show the set of entailed guards and non-entailed guards respectively. The last section of each case shows the local parameter condition for the case and the reason why simulation terminated (listed in Table 5.2).

Plot files can be used by the web frontend (See Section 5.1) and its format is called Hydat. The structure of Hydat is shown in Fig. 5.4. Output trajectories are expressed using symbolic parameters that represent uncertain values of variables at each time point. For the example program of Fig. 3.7, a symbolic parameter for the initial value of `x1` is introduced first. Intervals in interval arithmetic and noise symbols in affine arithmetic are also handled uniformly as symbolic parameters.

```
------ Result of Simulation ------
---------parameter condition(global)---------
p[y, 0, 1]      : (9, 11)
---------Case 1---------
---------PP 1---------
unadopted modules: {}
positive        :
negative        :
t       : 0
y       : p[y, 0, 1]
y'      : 10
y''     : -10
---------IP 2---------
unadopted modules: {}
positive        :
negative        :
t       : 0->Infinity
y       : t*(t+(-2))*(-5)+p[y, 0, 1]
y'      : (t+(-1))*(-10)
y''     : -10
---------parameter condition(Case1)---------
p[y, 0, 1]      : (9, 10)
# time reached limit
(** Case 2 is omitted... **)
---------Case 3---------
(** PP 1 is omitted... **)
---------IP 3---------
unadopted modules: {}
positive        :
negative        :
t       : 0->1+(-1)*(-2+p[y, 0, 1]*1/5)^(1/2)
y       : t*(t+(-2))*(-5)+p[y, 0, 1]
y'      : (t+(-1))*(-10)
y''     : -10
---------PP 5---------
unadopted modules: {FALL}
unsat mod       : {BOUNCE, FALL}
unsat cons      : {y''=-10, y'=-(4/5)*y'-}
positive        : y->=15=>y'=-(4/5)*y'-
negative        :
t       : 1+(-1)*(-2+p[y, 0, 1]*1/5)^(1/2)
y       : 15
y'      : 5^(-1/2)*(-8)*(-10+p[y, 0, 1])^(1/2)
(** Following phases are omitted... **)
---------parameter condition(Case3)---------
p[y, 0, 1]      : (10, 11)
# time reached limit
```

Figure 5.2: Output text by HyLaGI

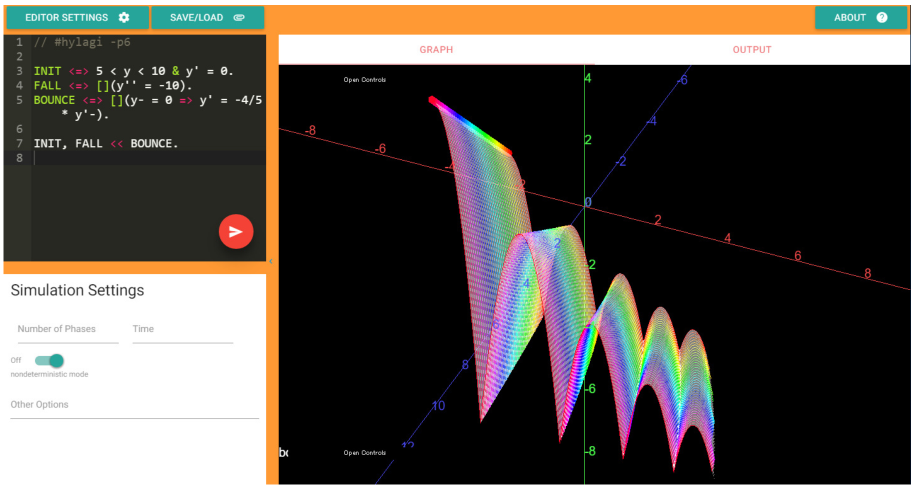
Table 5.1: ODEs handled by the backend solver (cited from [38])

Name	General Form
Separable	$y'(x) = f(x)g(y)$
Homogeneous	$y'(x) = f(\frac{x}{y(x)})$
Linear first-order ODE	$y'(x) + P(x)y(x) = Q(x)$
Bernoulli	$y'(x) + P(x)y(x) = Q(x)y(x)^n$
Ricatti	$y'(x) = f(x) + g(x)y(x) + h(x)y(x)^2$
Exact first-order ODE	$Mdx + Ndy = 0$ with $\frac{\partial M}{\partial y} = \frac{\partial N}{\partial x}$
Clairaut	$y(x) = xy'(x) + f(y'(x))$
Linear with constant coefficients	$y^{(n)}(x) + a_{n-1}y^{(n-1)}(x) + \cdots + a_0y(x) = P(x)$ with constant a_1
Hypergeometric	$x(1 - x)y''(x) + (c - (a + b + 1)x)y'(x) - aby(x) = 0$
Legendre	$(1 - x^2)y''(x) - 2xy'(x) + n(n + 1)y(x) = 0$
Bessel	$x^2y''(x) + xy'(x) + (x^2 - n^2)y(x) = 0$
Mathieu	$y''(x) + (a - 2qcos(2x))y(x) = 0$
Abel	$y'(x) = f(x) + g(x)y(x) + h(x)y(x)^2 + k(x)y(x)^3$
Chini	$y'(x) = f(x) + g(x)y(x) + h(x)y(x)^n$

5.1 Web Frontend

We also implemented a web frontend of HyLaGI, named webHydLa. WebHydLa aims at easy access by users and graphical supports for programming and simulation of HydLa models. It is written in JavaScript and Python. Users can access webHydla at http://webhydla.ueda.info.waseda.ac.jp/ by their web browsers. WebHydLa has several functionalities below.

- A text editor with syntax highlighting and auto completion for HydLa
- Three-dimensional plot of trajectories, which has an advantage that we can use the z-axis as an axis for symbolic parameters
- Animation of trajectories that facilitates the understanding of the model behavior



```
// #hylagi -p6

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

INIT, FALL << BOUNCE.
```

Figure 5.3: Screen shot of webhydla

- Modification of parameter conditions

Figure 5.3 shows a screen shot of webHydLa visualizing a bouncing ball model with uncertainty in its initial position.

5.2 Bounded Model Checking

Since HyLaGI can compute enclosures of all possible trajectories, we can use it on the purpose of bounded model checking of models. Users can specify the condition to be checked by the `ASSERT` statement in input programs. If HyLaGI is given a program with an `ASSERT` statement, it checks whether the condition is satisfied throughout the simulation. When the condition is violated, HyLaGI says "Assertion failed!" and shows values of parameters that violate the condition if necessary.

5.3 Implicit Continuity

The original semantics of HydLa [35] requires that if a constraint C in a given HydLa program refers to the derivative of a variable, the variable should be implicitly continuous. Such continuities have a higher priority than C itself and lower priority than any other constraint whose priority is higher than C.

```
{
    "variables": List<String(Name of Variable)>,
    "parameters": Map<String(Name of Parameter), Range>,
    "first_phases": List<Phase>
}: HyDat

{
    "id": Integer,
    "type": String("PP" or "IP"),
    "time" : Time
    "variable_map": Map<String(Name of Variable), Range>,
    "parameter_map": Map<String(Name of Parameter), Range>,
    "parameter_maps": List<Map<String(Name of Parameter),
                           Range> >,
    "simulation_state": optional String,
    "children": List<Phase>
}: Phase

{
    "time_point": String(Symbolic Formula),
    or
    "start_time": String(Symbolic Formula),
    "end_time": optional String(Symbolic Formula)
} : Time

{
    "unique_value": String(Symbolic Formula),
    or
    "lower_bounds": List<Bound>,
    "upper_bounds": List<Bound>
}: Range

{
    "closed": Bool,
    "value": String(Symbolic Formula)
}: Bound
```

Figure 5.4: Structure of Hydat

Table 5.2: Terminating messages and their meanings

Message	Meaning
`time reached limit`	Specified time limit is exceeded.
`number of phases reached limit`	Specified phase limit is exceeded.
`execution stuck`	Execution cannot be continued due to absence of consistent candidate module sets.
`assertion failed`	The condition in `ASSERT` statement is violated.
`time out`	Execution time exceeds specified limit (different from `time reached limit`)
`some values of variables are not unique in IP`	Some values of variables are not unique in the Interval Phase and simulation fails.
`following phases were not simulated`	The simulation for this case has not been continued. This is mainly caused by assertion failure in another case.
`simulation interrupted`	The simulation is interrupted by users.
`unknown error occurred`	Some error that cannot be handled by the simulator occurred.

However, if we implement this specification directly, the number of candidate module sets increases explosively. Hence in this research we adopt another specification.

In this implementation, the values of variables are considered to be continuous if either of the conditions below holds.

- The constraint store includes a constraint that refers to the derivative.
- In *CheckEntailment*, the consequent of the guard refers to the derivative.

5.4 Guards Referring to the Left-hand Limits of the Initial Time

Constraints in HydLa models constrain the variables from time 0, and they do not go back in time. Therefore, the left-hand limits of variables at time 0 are essentially undefined. We have considered three policies on how we handle guards that refer to such left-hand limits.

1. Consider them to be false — It reflects the intuitive meaning of the program as far as we experimented.

2. Consider them to be true — Many example programs behave badly. For example, in Fig. 4.6 `INIT` and `BOUNCE` conflict at the initial time and there are no solution trajectories.

3. Regard entailment of guards referring to left-hand limits as completely undefined — This is the most exhaustive choice. The simulation branches into 2^n cases (n represents the number of guards that contain backward reference).

In HyLaGI, we adopted the first policy taking the computational cost into account.

5.5 Scalability of HyLaGI

One of the concerns with a symbolic technique is its scalability. A naïve implementation of HydLa would lead to the calculation of all constraints at each phase, but HyLaGI only computes constraints related to each discrete change. This improvement is based on the following three ideas.

The first idea is to analyze the dependencies between constraints. The dependencies can be represented by a bipartite relation graph consisting of *variable nodes* and *constraint nodes*. Edges in the graph correspond to the references to variables from constraints. The dependencies between constraints changes dynamically in the simulation of HyLaGI because (i) a guarded constraint may be switched on and off, (ii) constraint modules not chosen in the current module set have no effect, and (iii) the left-hand limit of a variable in a Point Phase is regarded constant and has no relation with the variable itself. HyLaGI manages the effectiveness of nodes and edges of a dependency graph dynamically and calculates minimal sets of related constraints. We give detailed description about the relation graph in Section 5.5.1 and Section 5.5.2.

The second idea is to exploit the continuity of the values of variables and its derivatives. Variables whose values jump must be referred to in constraints that triggered discrete changes, and HyLaGI keeps other variables evolving continuously without recalculation.

The third idea is to compute candidate subsets of constraint modules dynamically on demand. In HydLa, the number of candidate subsets can increase exponentially with respect to the number of objects in the model. For example, it is 2^n for the program of Fig. 3.9 with n balls. However, the number of subsets to be checked for its maximality is usually small. HyLaGI calculates such subsets on demand by using the information of inconsistent subsets obtained in the process of consistency checking. When a subset is known to be inconsistent, at least one module must be removed from the subset to make it consistent, and HyLaGI removes a low-priority module M and those below M in the constraint hierarchy.

These improvements reduced the time complexity of the calculation of each discrete change. For example, for the program of Fig. 3.9, it is reduced from exponential (without the third idea) or $O(n^3)$ (with the third idea) to $O(n)$.

5.5.1 Construction of Relation Graph

In HyLaGI, a relation graph is constructed for a HydLa program. A relation graph $G = (C, V, E_p, E)$ is a bipartite graph consisting of constraint nodes and variable nodes. C is a set of constraint nodes and defined as $C := \{(c, m) \mid m \in \mathit{Modules}(\mathit{HP}), c \in \mathit{AtomicConstraints}(m)\}$. $\mathit{Modules}(\mathit{HP})$ means a set of all modules in given HydLa program. $\mathit{AtomicConstraints}(\mathrm{m})$ means a set of all atomic constraints without guards. In the construction of relation graphs, we ignore guards and consider them in the invalidation of constraint nodes (Section 5.5.2).

V is a set of variable nodes that corresponds to all variables and those derivatives. E and E_p are disjoint sets of edges between constraints and variables. E_p is a set of edges that means the constraint only refers to left-hand limits of variables (called "prev-edges"). E_p is defined as $E_p := \{(c, v) \mid$

$c \in C, v \in V, \mathit{referPrev}(v, c)\}$. E is a set of edges that means the constraint refers to variables themselves. E is defined as $E := \{(c, v) \mid c \in C, v \in V, \mathit{refer}(c, v)\}$. Note that constraints about time derivatives imply continuity of variables, e.g., constraint $x' = 0$ refers to both x and x'.

Figure 5.5 shows the relation graph of Fig. 3.9 with three balls. In Fig. 5.5, rectangular nodes correspond to constraint nodes and elliptic nodes correspond to variable nodes. For edges, dotted lines correspond to E_p and solid lines correspond to E.

5.5.2 Simulation with Relation Graph

If two constraint nodes on a relation graph are not connected, they share no variables. Such constraints cannot conflict with each other, so we can check consistency of them independently. In other words, we check consistency of constraints for each connected component. If a connected component is judged to be inconsistent, we obtain a set of corresponding modules from constraint nodes. This is why constraint nodes contain information about modules. Obtained module sets are exploited in dynamic computation of candidate module sets in Section 5.5. In addition, when we check consistency of a guard, we only have to check consistency of constraints related to the guard. Such localization significantly reduces the computational costs for models with multiple objects.

However, a statically constructed relation graph itself is not sufficiently localized. For example, all nodes in Fig. 5.5 are connected. As a matter of fact, statically unconnected components do not appear in realistic models because it means that they are completely independent. To solve this problem, we invalidate graph components such as constraint nodes and prev-edges dynamically as simulation goes on so that we can obtain unconnected components.

First, we can invalidate constraint nodes if any of three conditions below holds.

1. The module that the constraint node belongs to is not adopted in the current candidate module set.
   (For example, the constraint `CONST` in Fig. 3.9 is not adopted on collision of balls.)

2. The guard of the atomic constraint corresponding to the constraint

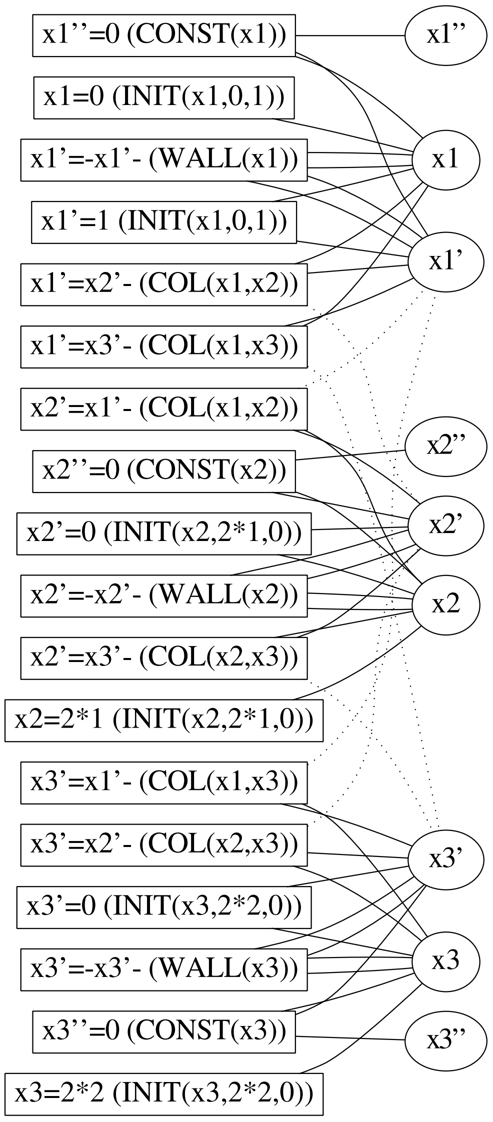


Figure 5.5: Relation graph for one-dimensional billiard

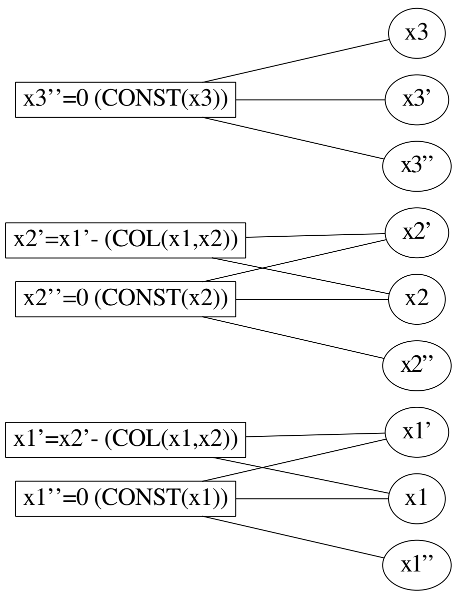


Figure 5.6: Relation graph for one-dimensional billiard on collision of the first ball and the second ball

node is not entailed.
(For example, the guards of `COL` and `WALL` in Fig. 3.9 are not entailed during continuous change.)

3. The constraint node does not have an "□" operator and the current time is not equal to 0.
(The constraint `INIT` is an example of this.)

Prev-edges are invalidated in Point Phases and validated in Interval Phases. This is because left-hand limits of variables in Point Phases are determined from preceding Interval Phases and regarded as constant values.

Example 14. *Figure 5.6 shows the relation graph on a Point Phase. The first ball (x1) and the second ball (x2) collide, and the corresponding constraint nodes are independent. In this Point Phase, {`COL(x1,x2), CONST(x1)`} and {`COL(x1,x2), CONST(x2)`} are inconsistent. We can easily see the dependencies between those constraints on the relation graph.*

Figure 5.7 shows the relation graph on Interval Phases. In Interval Phases, constraints of `COL` have no effect because the guards are not entailed, therefore constraints of `CONST` are the only valid constraints in the Interval Phases.

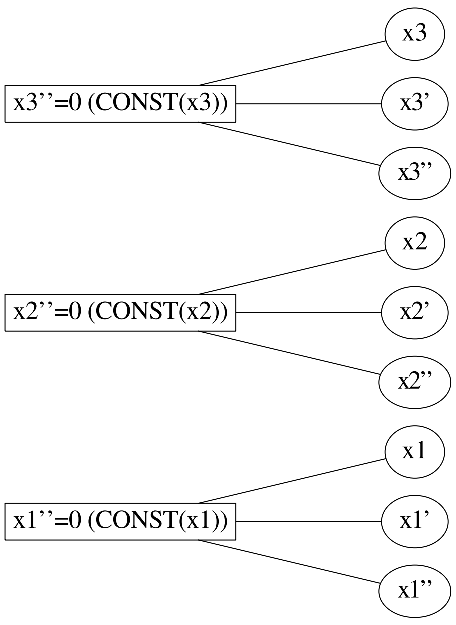


Figure 5.7: Relation graph for one-dimensional billiard on Interval Phases

Chapter 6

Experimental Results

In this chapter, we show experimental results of simulation by our implementation described in Chapter 5. First, we show the results of simulation with the fully symbolic method described in Section 4.1. After that, we show the results of simulation by the symbolic and interval method described in Section 4.2. Note that though plots of trajectories in this chapter show finitely many trajectories, the simulation results themselves represent infinitely many trajectories in the form of expressions with symbolic parameters.

6.1 Fully Symbolic Simulation

6.1.1 Bouncing Particle with a Hole

This example is a two-dimensional model where a ball bounces on the ground with a rectangular hole. Figure 6.2 shows a corresponding HydLa program. The overview of the model is described below.

- A ball is thrown from the point $\mathtt{x} = 0$, $\mathtt{y} = 10$.
- The initial horizontal velocity (x') is positive value and less than 20.
- The initial vertical velocity equals zero.
- The air resistance is ignored and the vertical acceleration y'' by the gravity is -10.
- The ground with a rectangular hole is at the height of $\mathtt{y} = 0$.

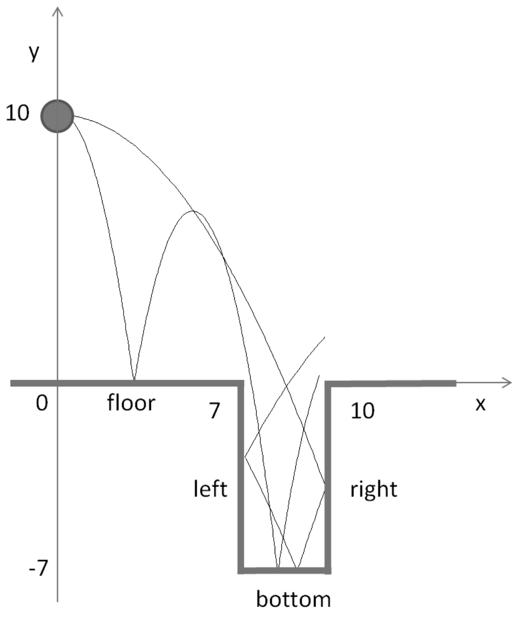


Figure 6.1: Trajectories of the bouncing particle with a hole

- The depth of the hole is 7, the width is 3, and the left edge is located on $x = 7$.
- The coefficient of restitution between the ball and the wall in the ground is 1, and the coefficient between the ball and the ground is 4/5.

The objective of the simulation is to compute the condition of the initial velocity where the ball reaches the target point beyond the hole. In Figure 6.2, `INIT` represents the initial state of the model. `FALL` and `BOUNCE` describe the vertical motion of the ball, while `XCONST` and `XBOUNCE` describe the horizontal one. The condition in `ASSERT` states that the ball never reaches the target point. By this `ASSERT` statement, we can compute the desired condition.

We simulated this program limiting the simulation time to 20 and the number of discrete changes to 6. The simulation branched into 50 different cases. As a result, we obtained the following conditions under which the ball reached the target point.

1. $1250/(405\sqrt{2} + \sqrt{317} + 9\sqrt{1387}) \leq \texttt{x'}_0 \leq 1250/(405\sqrt{2} - \sqrt{317} + 9\sqrt{1387})$
2. $125\sqrt{2}/97 \leq \texttt{x'}_0 \leq 35/(13\sqrt{2})$

```
INIT    <=> y = 10 /\ y' = 0 /\ x = 0 /\ 0 < x' <= 20.
FALL    <=> [](y'' = -10).
XCONST  <=> [](x'' = 0).
XBOUNCE <=> []((x- = 7 \/ x- = 10) /\ y- < 0  => x' = -x'-).
BOUNCE  <=> [](y- = -7 \/ (x- <= 7 \/ x- >= 10) /\ y- = 0
  => y' = -(4/5) * y'-).
ASSERT(!(y >= 0 /\ x >= 10)).

INIT, FALL << BOUNCE, XCONST << XBOUNCE.
```

Figure 6.2: HydLa program of the bouncing ball with a hole

3. $35/(13\sqrt{2}) < \texttt{x'}_0 \leq (1125\sqrt{2} + 225\sqrt{67} + 25\sqrt{197})/(928 + 81\sqrt{134})$
4. $-40\sqrt{197}/(928 + 81\sqrt{134}) + 360(5\sqrt{2} + \sqrt{67})/(928 + 81\sqrt{134}) \leq \texttt{x'}_0$ $< 25\sqrt{2}/13$
5. $25\sqrt{2}/13 \leq \texttt{x'}_0 \leq 7/\sqrt{2}$
6. $(117\sqrt{85} + 13\sqrt{485})/256 < \texttt{x'}_0 < 10\sqrt{5/17}$
7. $10\sqrt{5/17} \leq \texttt{x'}_0 \leq 10\sqrt{5/17}$
8. $10\sqrt{5/17} < \texttt{x'}_0 \leq (9\sqrt{85} + \sqrt{485})/16$
9. $5\sqrt{2} \leq \texttt{x'}_0 \leq 20$

By approximating these formulas into numerical values and adding the order of bouncing, we obtain the following nine cases.

1. $[1.36027, 1.40428]$ (floor, floor, bottom)
2. $[1.82244, 1.90375]$ (floor, floor)
3. $(1.90375, 2.02803]$ (floor, bottom)
4. $[2.64300, 2.71964)$ (floor, right, bottom, left)
5. $[2.71964, 4.94975]$ (floor)
6. $(5.33196, 5.42326)$ (bottom, right, left)
7. $[5.42326, 5.42326]$ (bottom+right, left)

```
INIT   <=> h = 10 /\ h' = 0 /\ timer = 0.
PARAMS <=> 2 <= exT <= 4 /\ 1 <= volume <= 3
  /\ [](exT' = 0 /\ volume' = 0).
TIME   <=> [](timer' = 1).
RESET  <=> [](timer- >= volume + exT => timer = 0).
RISE   <=> [](timer- < volume => h'' = 1).
FALL   <=> [](timer- >= volume => h'' = -2).
ASSERT(h>=0).

INIT, PARAMS, RISE, FALL, TIME<<RESET.
```

Figure 6.3: HydLa program of hot-air balloon

8. $(5.42326, 6.56241]$ (right, bottom, left)
9. $[7.07107, 20]$ ()

6.1.2 Hot-air Balloon

Figure 6.3 shows a HydLa program of a hot-air balloon. The hot-air balloon has multiple fuel tanks and it exchanges those tanks while flying. Each fuel tank can be used for `volume` seconds and exchanging takes `exT` seconds. `INIT` describes the initial state. `PARAMS` describes the behavior of symbolic parameters. `TIME` and `RESET` describe the behavior of timer. `RISE` and `FALL` describe the continuous behavior of the balloon. The balloon rises when the fuel of tanks are burning (`timer- < volume`) and it falls while exchanging (`timer- >= volume`). `ASSERT(h >= 0)` says that the balloon never collides with the ground, which is the safety property of this model. Note that both `volume` and `exT` have uncertainties. Figure 6.4 and Fig. 6.5 show the sample trajectories of the program, where `volume` is fixed to three and `exT` is fixed to two, respectively.

For this model, we performed bounded model checking up to 30 seconds. As a result, we obtained six different cases shown in Fig. 6.6. This figure shows a two-dimensional parameter space for `volume` and `exT`. Each divided subspace corresponds to each case, which differs by the number of phases and whether `ASSERT` is violated or not. "PPn" means that n Point Phases have been computed.

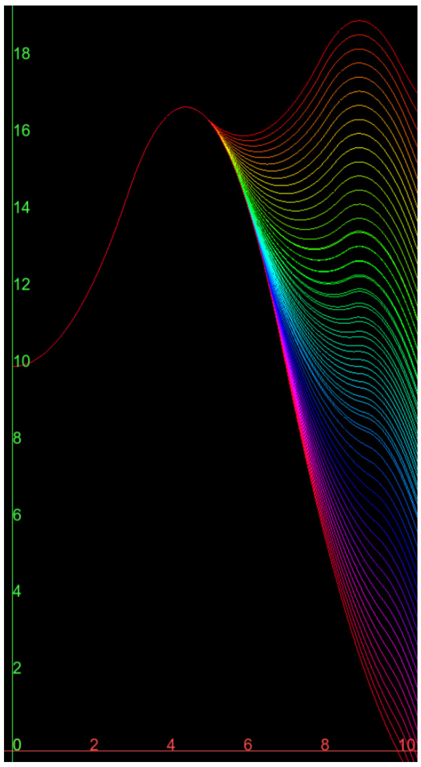


Figure 6.4: Trajectory of balloon (`volume = 3`)

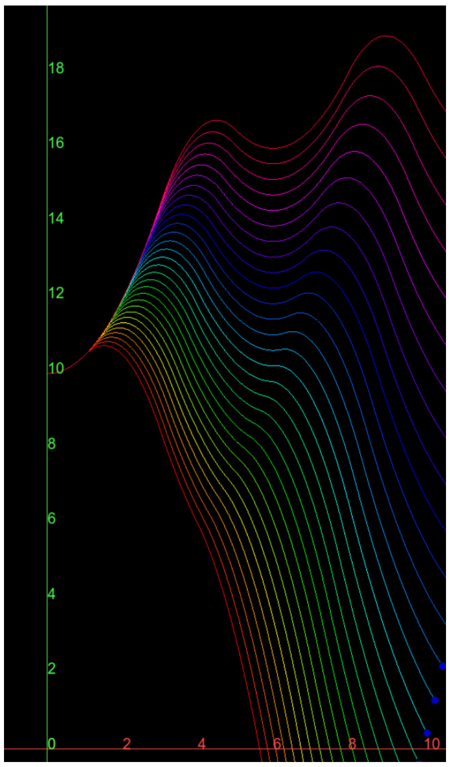


Figure 6.5: Trajectory of balloon (`exT = 2`)

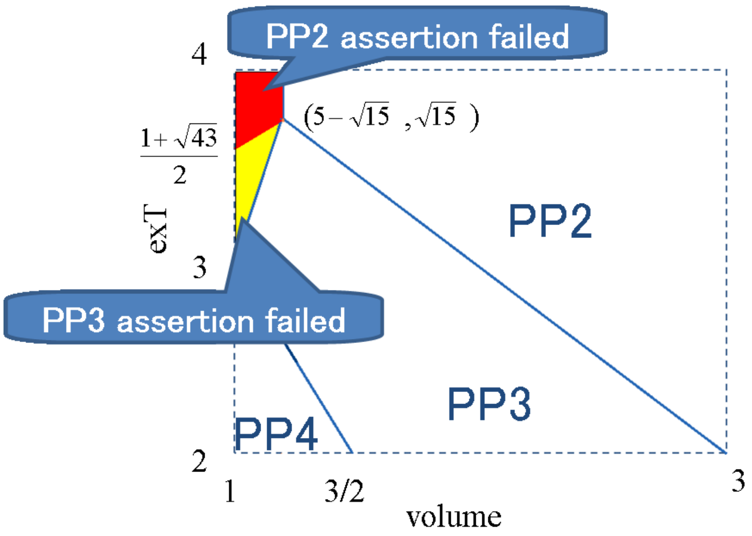


Figure 6.6: Classifying cases in parameter space

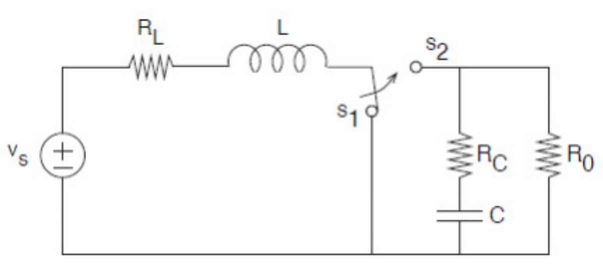


Figure 6.7: Electric circuit (cited from [33])

6.1.3 Electric Circuit

This example model expresses behavior of an electric circuit shown in Fig. 6.7. The circuit has a switch and the switch determines the continuous behavior of the circuit. We focus on the voltage of the capacitor v_C and the current of the inductor i_L. When the switch is directing s_1, the behavior of v_C and i_L is described by the following ODEs.

$$\begin{cases} \frac{d}{dt}i_L = -\frac{R_L}{L}i_L + \frac{1}{L}v_S \\ \\ \frac{d}{dt}v_C = -\frac{1}{C}\frac{1}{R_C + R_0}v_C \end{cases} \quad \text{(on s1)}$$

$$\begin{cases} \frac{d}{dt}i_L = -\frac{1}{L}(R_L + \frac{R_C R_0}{R_C + R_0})i_L - \frac{1}{L}\frac{R_0}{R_C + R_0}v_C + \frac{1}{L}v_S \\ \\ \frac{d}{dt}v_C = -\frac{1}{C}\frac{R_0}{R_C + R_0}i_L - \frac{1}{C}\frac{1}{R_C + R_0}v_C \end{cases} \quad \text{(on s2)}$$

Figure 6.8 shows a corresponding HydLa program with $L = 1, R_L = 1, v_S = 5, R_C = 1, R_0 = 1$ and $C = 1$. The initial value of i_l equals zero and the initial value of v_c has a range from zero to five. In this program, we use a variable `timer` to cause switching at every one second. `INIT` describes the initial state of the model. `TIMER` describes the behavior of `timer`. `SWITCH` describes discrete switching. `STATE1` and `STATE2` describe the continuous behavior of the circuit. The guard `s = 0` describes that the switch is connected to s1 and the guard `s = 1` describes that it is connected to s2.

Figure 6.9 and Fig. 6.10 show plots of the simulation result. In these

```
INIT   <=> 0 <= vc <= 5 /\ il = 0 /\ s = 0 /\ timer=0.
TIMER  <=> [](s' = 0 /\ timer' = 1).
SWITCH <=> [](timer- = 1 => timer = 0 /\ s = 1 - s-).
STATE1 <=> [](s = 0 => il' = -il + 5 /\ vc' = -1/2 * vc).
STATE2 <=> [](s = 1 =>
  il' = -3/2 * il - 1/2 * vc + 5
  /\ vc' = 1/2 * il - 1/2 * vc).

INIT, TIMER << SWITCH, STATE1, STATE2.
```

Figure 6.8: HydLa program of electric circuit

plots, we incremented the initial value of v_c from 0 to 5 by 0.5. In these figures, we can see that the influence of v_c decreases as time goes on and converges into a fixed behavior.

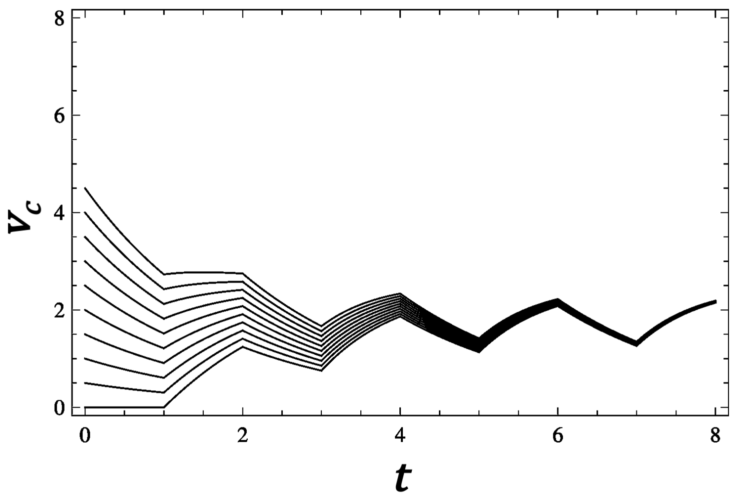


Figure 6.9: Behavior of v_c

6.1.4 Curling Stone

Figure 6.11 shows a HydLa program of a curling stone. There is one curling stone and collision does not happen in this model. The objective of the control is to stop the stone within the target range, which is $9 < \texttt{x} < 11$. This model is one-dimensional; the initial position $x(0)$ is 0, and the initial velocity

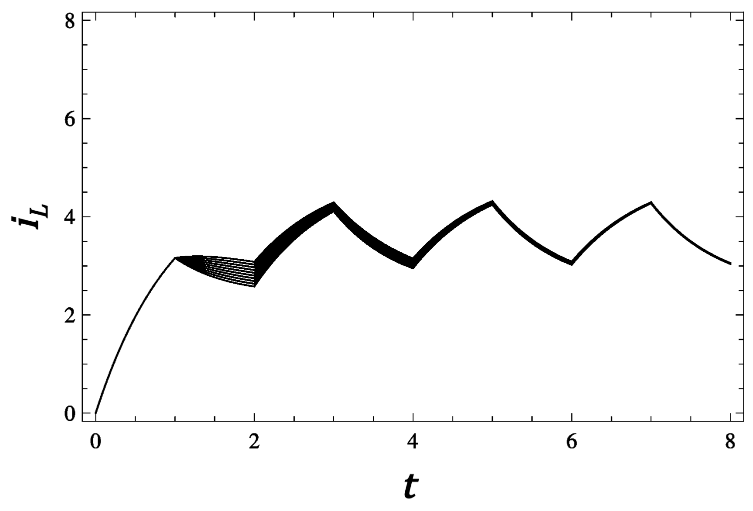


Figure 6.10: Behavior of i_L

```
INIT     <=> x = 0 /\ x' = 1
  /\  [](threshold' = 0) /\ [](fric' = 0)
  /\  0 < threshold < 1 /\ fric = -1/40.
//  /\  threshold = 3/4 /\ -1/20 < fric < -1/100.
FRICTION <=> []((x' > 0 => x'' = -1/10)
  /\ (x' <= 0 => x'' = 0)).
SWEEPING <=> [](x < 9 /\ 0 < x' < threshold
  => x'' = fric).

ASSERT(x' != 0 \/ 9 <= x <= 11).
INIT, FRICTION << SWEEPING.
```

Figure 6.11: HydLa program of curling stone

$x'(0)$ is 1. By default, the stone moves with acceleration $x''(t) = -1/10$. There are sweepers and they can reduce the friction with the ground. The sweepers start sweeping if the velocity of the stone is less than the predefined threshold and the stone has not reached the target area yet. In this model, the threshold or the acceleration while sweeping has an uncertainty. The program of Fig. 6.11 introduces an uncertainty into the threshold, while we can introduce an uncertainty into the acceleration by replacing the third line with the fourth line, which is commented out. The objective of the simulation is to compute the condition to stop the stone within the target range. In Fig. 6.11, `INIT` describes the initial state of the model, `FRICTION` describes the default friction of the ball, and `SWEEPING` describes the start of sweeping and the friction during sweeping. Both `threshold` and `fric` are uncertain parameters, where `threshold` means a parameter for the threshold velocity, and `fric` means the acceleration of the stone during sweeping. The `ASSERT` statement describes the negation of the control objective.

The simulation result of this program is shown in Fig. 6.12 and Fig. 6.13. The simulation time is limited to 40. If the threshold velocity is uncertain, we have to set the range of `threshold` to $2/\sqrt{15} \leq \texttt{threshold} \leq 2/\sqrt{5}$ to stop the stone within the target area. The range is numerically approximated to $0.516 \leq \texttt{threshold} \leq 0.894$. If this condition is satisfied, the stone stops at $\texttt{x} = 8 + 15\texttt{threshold}^2/4$. If $\texttt{threshold} < 2/\sqrt{15}$, the stone stops at the point $\texttt{x} = 5 + 15\texttt{threshold}^2$ and does not reach the target area. If $2/\sqrt{5} < \texttt{threshold}$, sweeping starts too early and the stone passes through the target area and stops at $\texttt{x} = 8 + 15\texttt{threshold}^2/4$.

On the other hand, if `fric` has uncertainty, we have to set the range of `fric` to $-9/218 \leq \texttt{fric} \leq -13/1090$ to stop the stone within the target area. The range is numerically approximated to $-0.041 \leq \texttt{fric} \leq -0.0119266$. If the condition is satisfied, the stone stops at $\texttt{x} = 189/16 + 545\texttt{fric}/8$. If $\texttt{fric} < -9/218$, the stone stops at $\texttt{x} = (70 - 9/\texttt{fric})/32\texttt{fric}$ and does not reach the target area. If $-13/1090 < \texttt{fric}$, the stone passes through the target and stops at $\texttt{x} = 189/16 + 545\texttt{fric}/8$.

6.2 Simulation with Interval Arithmetic

In this section, we show the results of the simulation with interval arithmetic. We compare different choices in the implementation of *Enclose* and *FindMinTime* in Section 4.2. We used three example models for comparison.

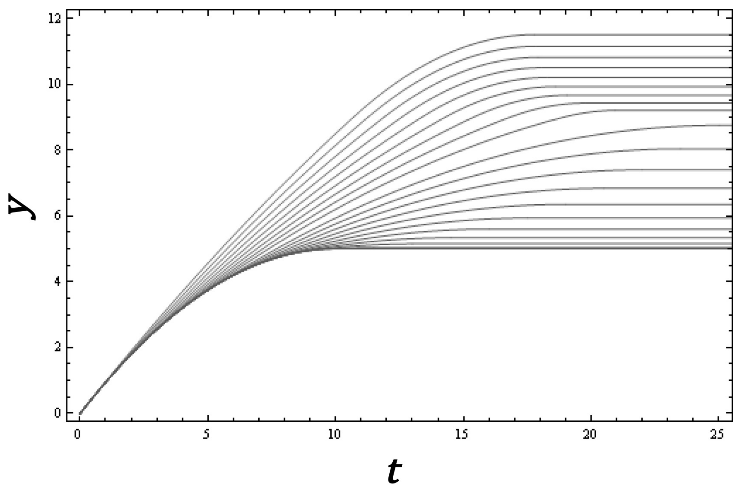


Figure 6.12: The trajectory of the stone with uncertain threshold

6.2.1 Two Water Tanks

This model is described in Chapter 3. The program is shown in Fig. 3.7. One of the result trajectories is shown in Fig. 6.14, where the horizontal axis represents time and the vertical axis represents each water level. We simulated this model with two choices below.

1. Use the result of the interval Newton method directly, skipping the second step in *FindMinTime*, and compute enclosure with interval arithmetic instead of affine arithmetic in *Enclose* (denoted by `Newton&IA`).

2. Use the interval Newton method and the mean value theorem in *FindMinTime*, and compute enclosure with affine arithmetic in *Enclose* as proposed in Section 4.2 (denoted by `Mean&AA`). In the simulation, we preserved six symbolic parameters at the reduction performed at the end of each Point Phase.

Figure 6.15 shows the sum of the width of the interval at each phase and Fig. 6.16 shows the execution time. The horizontal axes represent the number of steps, where a step is a pair of a Point Phase and the following Interval Phase.

In Fig. 6.15, the initial width of the interval is 0.0001, which is the initial width of `x1`. In this program, the ideal behavior of water levels converges despite the uncertainty of the initial value. With `Newton&IA`, the width of

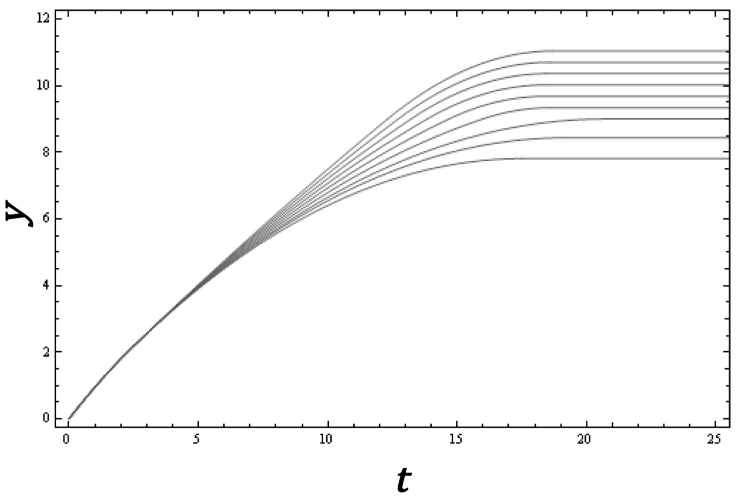


Figure 6.13: The trajectory of the stone with uncertain friction

intervals does not converge because of the dependency problem, and simulation fails after 61 steps. On the other hand, `Mean&AA` successfully handles the dependencies of quantities, reducing the width of the intervals into less than 10^{-7}.

As shown in Fig. 6.16, `Newton&IA` has less computational costs than `Mean&AA`. Both methods have constant time complexity with respect to the number of steps, which is an advantage of using the *Enclose* function. If we did not use *Enclose*, the execution time would grow linearly for this program because the size of symbolic expressions would keep growing.

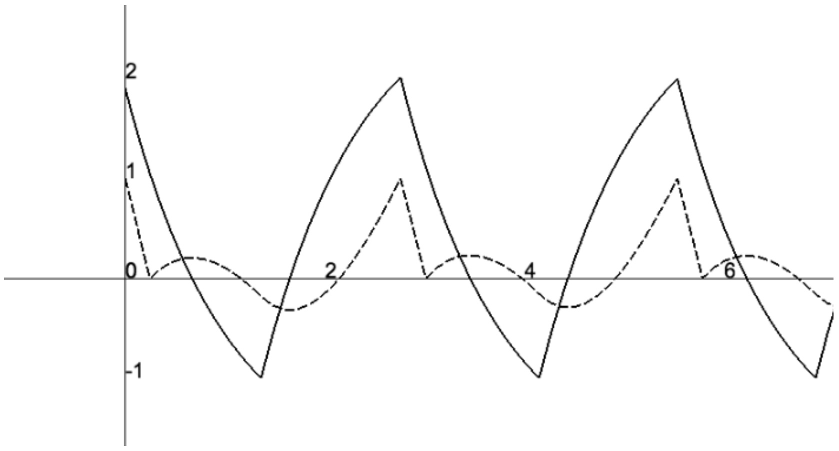


Figure 6.14: Trajectory of two water tanks (solid line: x1, dashed Line: x2)

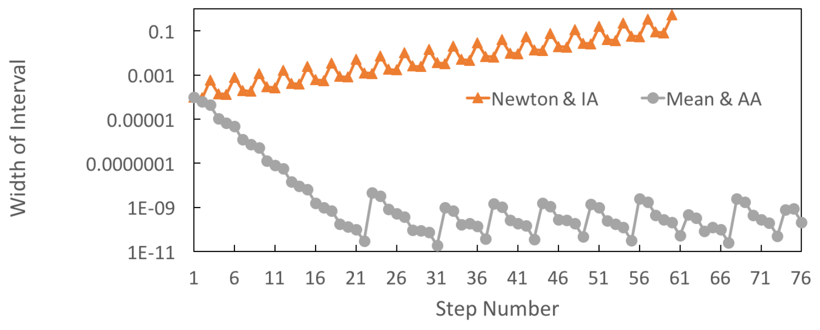


Figure 6.15: Width of interval for two water tanks

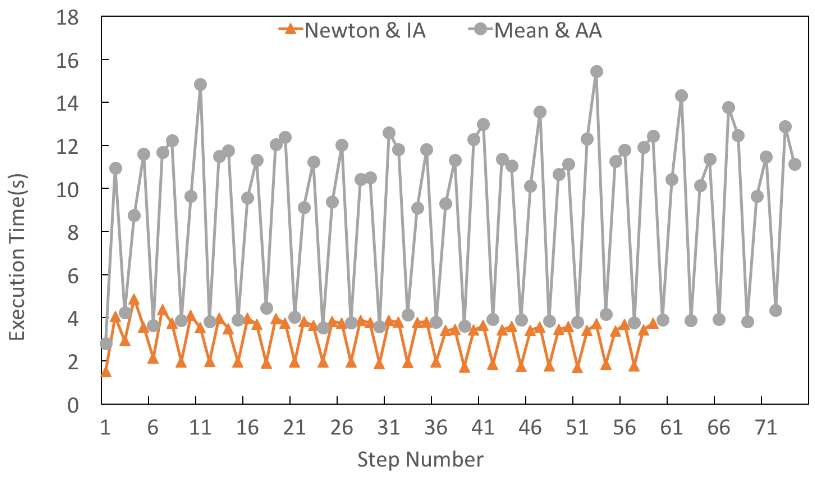


Figure 6.16: Execution time for two water tanks

6.2.2 Bouncing Ball on A Sine Curve

The second model is a variant of a bouncing ball model, which bounces on the floor whose shape is a sine curve in two-dimensional space. Figure 6.17 shows the input program. In this program, `x` and `y` denote the position of the ball, `e` denotes the coefficient of restitution, and `cont`, `s` and `c` are auxiliary variables to describe bouncing. Figure 6.18 shows one of the trajectories of this model.

Figure 6.19 shows the sum of the widths of the intervals for all variables at each phase and Fig. 6.20 shows execution time. This model has no uncertainty in its initial state, therefore the widths of the intervals increases solely

```
INIT   <=>  x = 0 /\ x' = 0 /\ y = 10 /\ y' = 0
  /\ e = 1 /\ [](e' = 0).
FALL   <=> [](cont = 1 => y'' = -10).
CONSTX <=> [](cont = 1 => x'' = 0).
SC     <=> [](s = cos(x-)/(1 + cos(x-)^2)^(1/2)
  /\ c = 1 /(1 + cos(x-)^2)^(1/2)).
BOUNCE <=> []( y- = sin(x-) => cont = 0
  /\ x' = ((-e) * s-^2 + c-^2) * x'- + ((e+1) * s- * c-) * y'-
  /\ y' = ((e+1) * s- * c-) * x'- + (s-^2 + (-e) * c-^2) * y'-).

INIT, SC, FALL, CONSTX,
 [](cont = 1) << BOUNCE.
```

Figure 6.17: HydLa program of bouncing ball on a sine curve

by computational errors. The meanings of `Newton&IA` and `Mean&AA` are the same as in Section 6.2.1. In this experiment, we also changed the number of preserved symbolic parameters to 5, 9, and 13, which is denoted by `d5`, `d9` and `d13`. Note that 5 is the minimum number of the noise symbols because affine arithmetic handles 5 affine quantities for `x`, `x'`, `y`, `y'` and `t`.

As can be seen from Fig. 6.19, the widths of the intervals in all methods increases exponentially. However, we can reduce the speed of the increase by introducing more noise symbols. If the widths increase and reach the amplitude of the sine curve, the computation of *FindMinTime* fails and further simulation cannot be performed.

In Fig. 6.20, as with the previous example, `Newton&IA` has less computational costs than `Mean&AA`. However, an important point is that the execution time of each simulation seems to have some upper bound. Remember that the fully symbolic method suffers from the exponential growths of computational costs. This is because we handle growing symbolic expressions of trajectories. By numerical approximation, we succeeded in reducing such growth.

6.2.3 Bouncing Ball on A Parabola

This model is another variant of a bouncing ball, which bounces on a parabola-shaped floor described by $y = x^2$. Figure 6.21 shows the input program and Fig. 6.22 shows the trajectory. The ball starts to fall with $x(0) = 2, y(0) = 8$

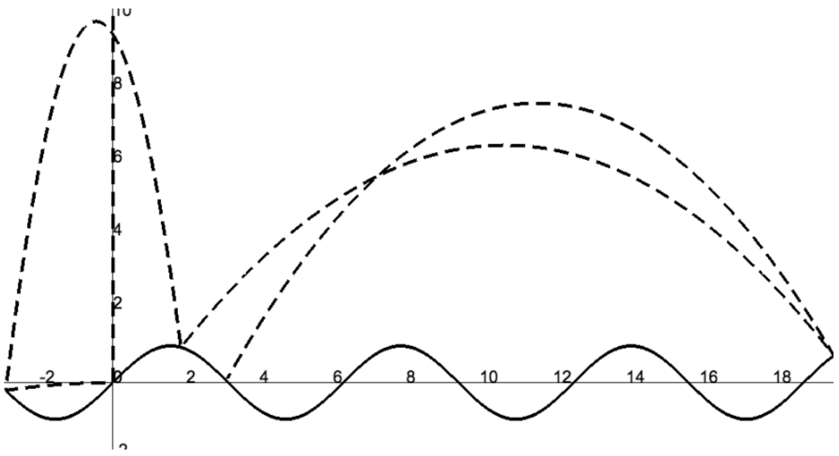


Figure 6.18: Trajectory of bouncing ball (solid line: floor, dashed Line: Ball)

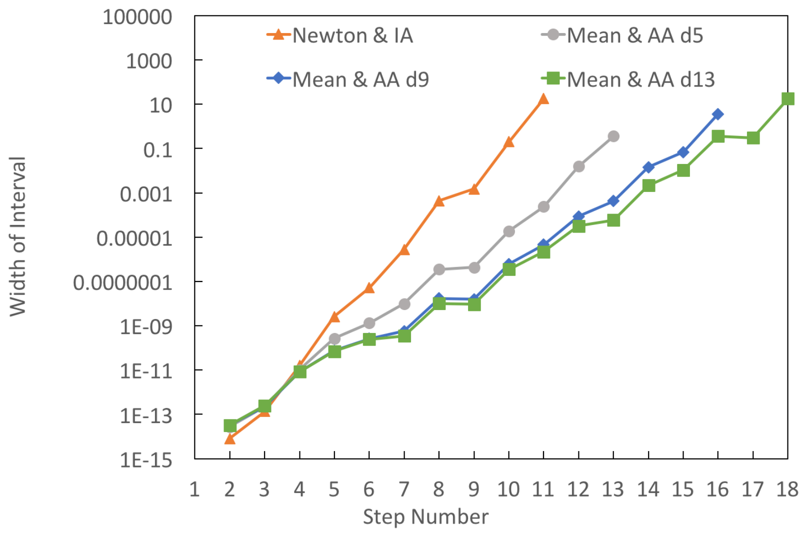


Figure 6.19: Width of interval for bouncing ball

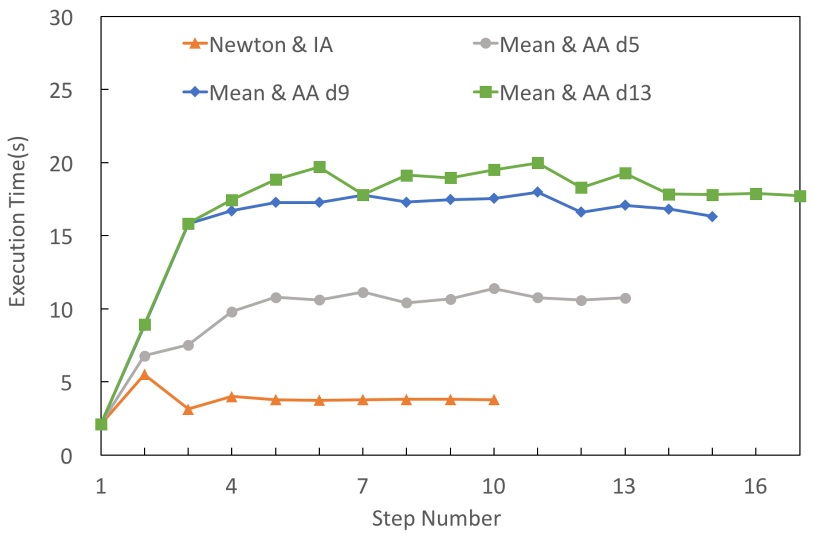


Figure 6.20: Execution time for bouncing ball

```
INIT   <=> x = 2 & y = 8 & x' = 0 & y' = 0.
FALL   <=> [](cont = 1 => x'' = 0 & y'' = -9).
BOUNCE <=> [](y- = (x-)^2 => cont = 0
  /\ x' = x'- - (4*(2*x- * x'- - (y'-))*x-)/(4*(x-)^2 + 1)
  /\ y' = y'- + (2*(2*x- * x'- - (y'-)))/(4*(x-)^2 + 1)).

INIT, FALL, [](cont = 1) << BOUNCE.
```

Figure 6.21: HydLa program of bouncing ball on a parabola

and repeats bouncing on the floor.

Figure 6.23 and Fig. 6.24 show the performance of the simulation. The simulation timeout is set to three minutes. In this experiment, we changed the number of preserved symbolic parameters to 5, 6, 7, 8 and 9. Figure 6.24 also includes the execution time of the fully symbolic method for comparison (denoted by `Symbolic`). The fully symbolic one outperforms the methods with interval and affine arithmetic. For this model, the influence of the number of symbolic parameters seems to be larger than the influence of the complexity of the symbolic expressions.

Next, we evaluate the performance of the parametric version of Fig. 6.21. We parametrize the initial horizontal velocity by replacing the first line with

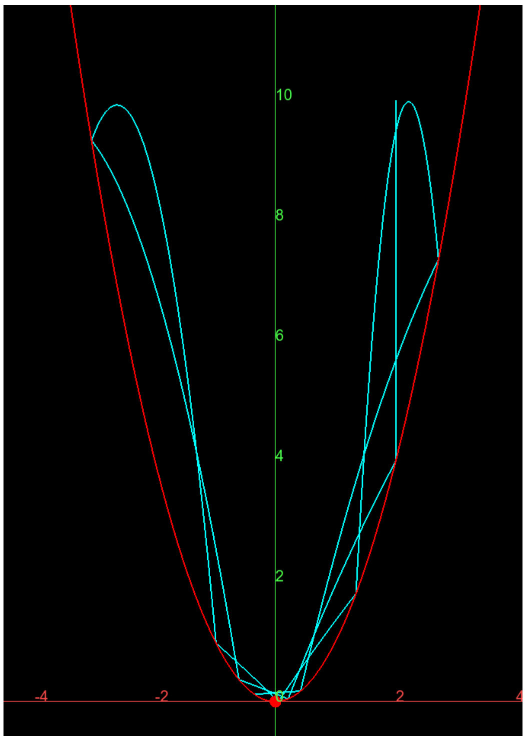


Figure 6.22: Trajectory of a bouncing ball on a parabola

`INIT <=> x = 2 /\ y = 8 /\ 0 <= x' <= 0.00001 /\ y' = 0`. Figure 6.25 and Fig. 6.26 show the performance for the parametric version. The symbolic method no longer performed well; it timed out at the third step. All the integrating methods can simulate until the widths of the intervals reach unacceptable ranges.

6.3 Summary of Experiments

In this chapter, we showed the results of the fully symbolic method and the version integrated with interval arithmetic. By the fully symbolic method, we succeeded in performing case analysis and bounded model checking of parametric hybrid systems with large uncertainties and more than 10 multiple different cases. In the plots of trajectories, we can see how the parameters influence the behaviors of models.

In the integrated version, we succeeded in handling models that could not be handled by symbolic formula manipulation itself. We also succeeded in

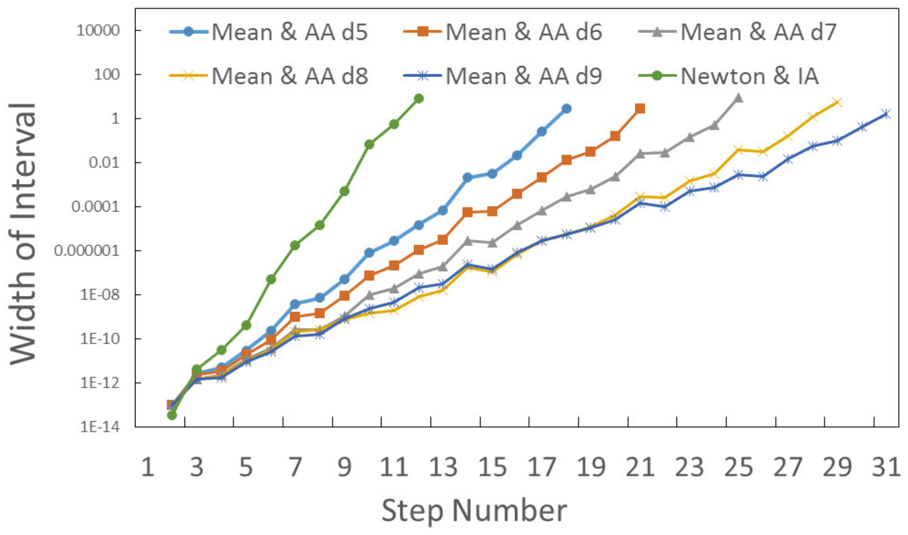


Figure 6.23: Width of interval for bouncing ball on a parabola

suppressing the increase of computational costs caused by complex symbolic formulas of the perturbed bouncing ball model. The accuracy of the proposed method are better than naïve interval arithmetic. This improvement is achieved by the preservation of linear terms of symbolic parameters. There is a trade-off between the accuracy and the execution time depending on the number of preserved symbolic parameters.

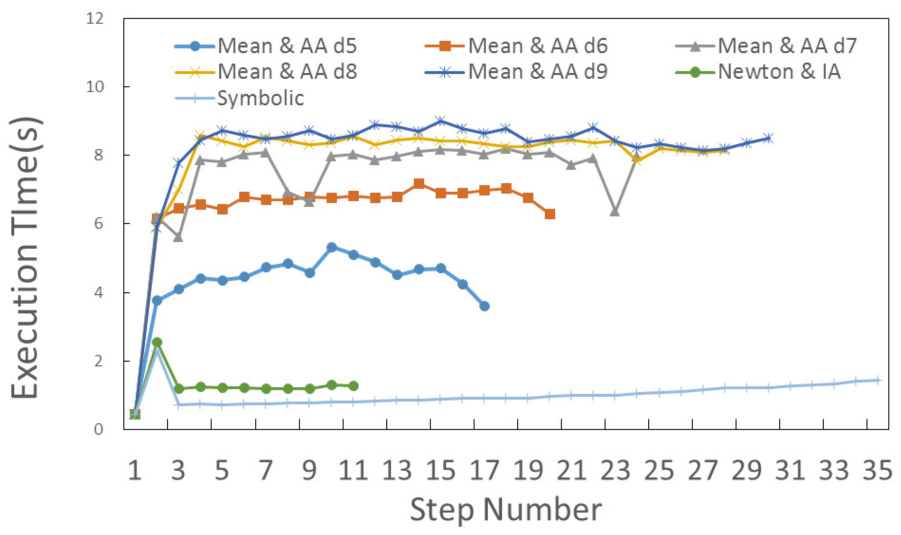


Figure 6.24: Execution time for bouncing ball on a parabola

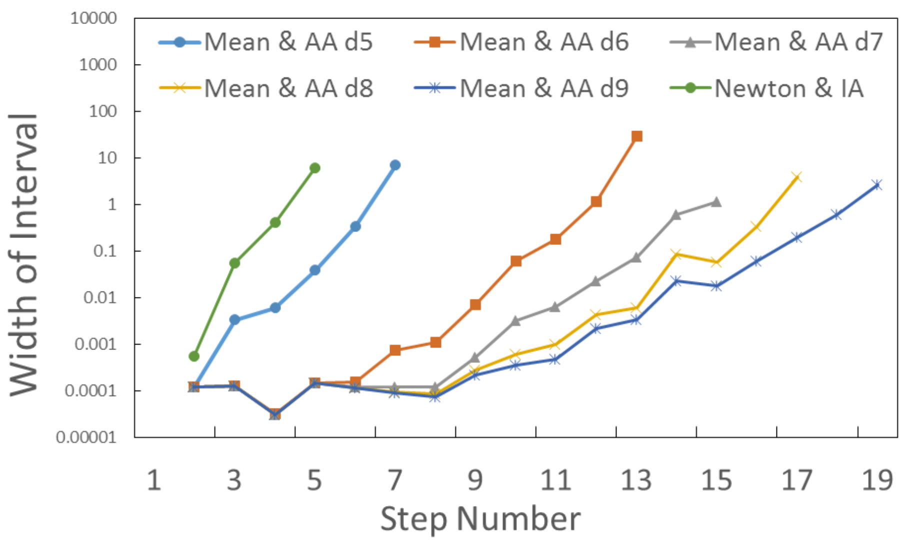


Figure 6.25: Width of interval for parametric bouncing ball on a parabola

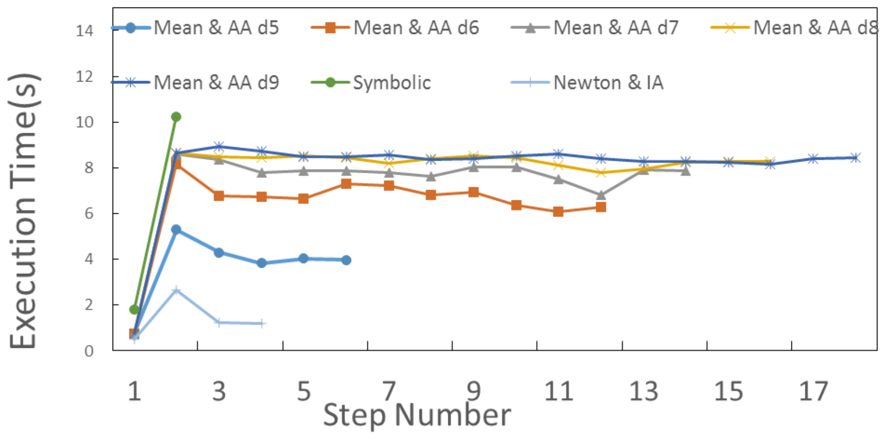


Figure 6.26: Execution time for parametric bouncing ball on a parabola

Chapter 7

Related Work

There are several tools for computing the behavior of hybrid systems rigorously.

Flow* [5][6] is a tool for computing reachable sets of hybrid automata [15]. Flow* encloses the solutions of nonlinear ODEs with flow pipe construction based on the Taylor model. For the computation of discrete changes, Flow* uses *domain contraction*, which is based on a branch and prune algorithm for initial values and time. Flow* can compute the enclosure of trajectories of the example of two tanks in Chapter 6 efficiently, and the width of the computed enclosure converges, but it cannot handle the bouncing ball model on a sine curve because it does not support non-polynomial inequalities as guards and invariants.

Acumen [39] is a hybrid system simulator and takes an original imperative language as its input. In addition to rich graphical features such as drawing three-dimensional objects, it supports rigorous simulation, which can compute an enclosure beyond a Zeno time point [9]. In our experiment, it could handle the example of two water tanks, but the computed enclosure did not converge and the simulation stopped after 12 times of switching of valves.

The tool named dReach [21] is a bounded model checker of hybrid systems based on a satisfiability modulo theory (SMT) solver with ODEs named dReal. It unrolls execution of hybrid automata to bounded length and reduces the problem into SMT formulas. dReach is based on interval constraint propagation and can check the satisfiability of given properties. dReach can compute an enclosure of trajectories in a form of a witness of the satisfiability, but the result is not parametric.

Goldsztejn and Ishii developed a simulation method based on interval arithmetic [12]. This method adopts parallelotopes as enclosing shapes of states and succeeded in reducing the wrapping effect.

KeYmaera [30] and KeYmaera X [11] are theorem provers of differential dynamical logic and are related to our tool as tools employing symbolic computation. They adopt a theorem proving approach, which is powerful but is inherently interactive, while HyLaGI is a simulator of models that computes trajectories automatically. KeYmaera has simulation capability of hybrid programs to assist users to understand models; however, it is not the main purpose of KeYmaera, and the simulation feature does not allow uncertainties in the models. KeYmaera and KeYmaera X are also different from HyLaGI in that they are fully based on symbolic computation.

Ñañez et al. developed a symbolic simulator for hybrid systems based on Matlab [29]. This simulator can simulate systems that are sensitive to perturbation rigorously, but it does not aim at parametric systems.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this research, we have developed a symbolic simulator of parametric hybrid systems that is based on constraint solving techniques. This simulator integrates symbolic and interval techniques internally and can deal with the drawbacks of each technique. The results of the simulation explicitly preserve information about symbolic parameters. It can be used on the purpose of parameter analyses of hybrid systems. We revisit the contributions of this thesis as follows.

- **Symbolic simulation algorithm of parametric hybrid systems (Section 4.1):** This algorithm adopts symbolic formula manipulation as its basic operation. An input model is given in the form of a HydLa program that is based on a constraint-based formalism. All values of variables are represented by symbolic formulas with parameters throughout the simulation. The result of computation expresses all possible trajectories of parametric hybrid systems and involves no errors caused by floating-point arithmetic. If the behavior of an input model may branch depending on the condition of symbolic parameters, this algorithm automatically detects such branching and performs case analysis about parameters. In the experiment, we showed that the method can simulate models with large uncertainties and 50 qualitatively different cases.

- **Extension of symbolic simulation by integrating with interval arithmetic (Section 4.2):** This extension enables simulation of models that cannot be handled analytically. We can also use this method to reduce computational costs of symbolic formula manipulation. In this method, we use affine arithmetic to over-approximate complex symbolic formulas into affine forms, which are simpler but still preserve linear terms of parameters. We use the interval Newton method and the mean value theorem to compute parametric zero-crossings of nonlinear equations. These zero-crossings can be exploited to compute boundaries of systems of inequations. In the experiment, we showed that the method improves the accuracy of the results compared to naïve interval arithmetic. There is a trade-off between the accuracy and the execution time of the simulation. This trade-off can be tuned by the number of symbolic parameters that we preserve in simulations.

- **Implementation of proposed methods (Chapter 5):** We implemented the proposed method with C++ and Mathematica. The implementation is able to compute trajectories with symbolic parameters. It also features bounded model checking as a natural extension of the symbolic simulation. The implementation is publicly available through a web frontend or a command line interface. The web frontend supports three-dimensional plots of resultant parametric trajectories.

8.2 Future Work

In future work, the method in this thesis should be extended to handle models with large uncertainties, which is important for design problems that demand wide ranges of parameters. For such models, the current *FindMinTimeInterval* does not work well because it assumes that the uncertainties are small enough and the solution does not branch depending on the values of parameters. If the range of a parameter is wide and includes a corner case, the interval Newton method fails because both of $f(m(X))$ and $f'(X)$ in the Newton operator $N(X)$ include zero. To deal with this problem, we need to consider the division of parameter space into three subspaces, the first being the space where the solution of $f(X) = 0$ definitely exists, the second where definitely no solution exists, and the third where the existence of solution is unclear (including the corner case). Such subdivision enables us to continue simulation for the first and the second spaces and output the third space as a corner case. Doing this intelligently is among our research agendas.

It is also an important extension to handle models with nonlinear ODEs that cannot be solved analytically. We are now investigating a method to enclose a nonlinear ODE with a parametric linear ODE that can already be handled in our simulation framework. Performance comparison with related tools is another important issue.

References

[1] Berz, M. and Makino, K. : Higher order multivariate automatic differentiation and validated computation of remainder bounds, WSEAS Transactions on Mathematics, 1998, Vol. 3, Issue 1, pp. 37–44.

[2] Borning, A., Freeman-Benson, B. and Wilson, M. : Constraint Hierarchies, Lisp and Symbolic Computation, Vol. 5, No. 3, 1992, pp. 223–270.

[3] Carloni, L., Passerone, R., Pinto, A. and Sangiovanni-Vincentelli, A. L. : Languages and Tools for Hybrid Systems Design, Foundations and Trends in Design Automation, Vol. 1, No. 1, 2006, pp. 1–204.

[4] Carlson, B. and Gupta, V. : Hybrid CC with Interval Constraints, in Proc. ACM International Conference on Hybrid Systems: Computation and Control, LNCS 1386, Springer, 1998, pp. 80–94.

[5] Chen, X., Abraham, E. and Sankaranarayanan, S. : Flow*: An Analyzer for Non-Linear Hybrid Systems, in Proc. International Conference on Computer Aided Verification, 2013, pp. 258–263.

[6] Chen, X. : Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models, PhD thesis, RWTH Aachen University, 2015.

[7] Chen, X., Schupp, S., Makhlouf, I. B., Abraham, E., Frehse, G. and Kowalewski, S. : A Benchmark Suite for Hybrid Systems Reachability Analysis, in Proc. 7th NASA Formal Methods Symposium, Springer, 2015, pp. 408–414.

[8] de Figueiredo, L. H. and Stolfi, J. : Affine Arithmetic: Concepts and Applications, Numerical Algorithms, Vol. 37, Issue 1-4, 2004, pp. 147–158.

[9] Duracz, A., Bartha, F. A. and Taha, W. : Accurate rigorous simulation should be possible for good designs, in Proc. International Workshop on Symbolic and Numerical Methods for Reachability Analysis, 2016, pp. 1–10.

[10] Frehse, G., Guernic, C. L., Donzé, A., Cotton, S., Ray, R., Lebeltel, Olivier., Ripado, R., Girard, A., Dang, T. and Maler, O. : SpaceEx: Scalable Verification of Hybrid Systems, in Proc. International Conference on Computer Aided Verification, LNCS 6806, Springer, 2011, pp. 379–395.

[11] Fulton, N., Mitsch, S., Quesel, J.D., Völp, M. and Platzer, A. : KeYmaera X: An axiomatic tactical theorem prover for hybrid systems, in Proc. International Conference on Automated Deduction, LNCS 9195, Springer, 2015, pp. 527–538.

[12] Goldsztejn, A. and Ishii, D. : A Parallelotope Method for Hybrid System Simulation, Reliable Computing, Springer, Vol. 23, 2016, pp. 163–185.

[13] Gupta, V., Jagadeesan, R., Saraswat, V. and Bobrow, D.G. : Programming in Hybrid Constraint Languages, in Hybrid Systems II, LNCS 999, Springer, 1995, pp. 226–251.

[14] Hansen, E. : A generalized interval arithmetic, in Nickel K. (eds) Interval Mathematics, LNCS 29, Springer, 1975, pp. 7–18.

[15] Henzinger, T. : The Theory of Hybrid Automata, in Proc. Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996, pp. 278–292.

[16] Hickey, T. J. and Wittenberg, D. K. : Rigorous Modeling of Hybrid Systems Using Interval Arithmetic Constraints, in Proc. ACM International Conference on Hybrid Systems: Computation and Control, LNCS 2993, Springer, 2004, pp. 402–416.

[17] IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008, 2008.

[18] Kashiwagi, M. : An algorithm to reduce the number of dummy variables in affine arithmetic, in Proc. 15th GAMM-IMACS International

Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2012.

[19] Kashiwagi, M. : kv library, http://verifiedby.me/kv/ (in Japanese).

[20] Kobayashi, T., Kono, F., Matsumoto, S. and Ueda, K. : Optimization of HydLa Implementation Using Difference Information of Constraints between Phases and Relation between Variables and Constraints, in Proc. 28th Annual Conference of the Japanese Society for Artificial Intelligence, 2014, 4C1-2. (in Japanese)

[21] Kong, S., Gao, S., Chen, W. and Clarke, E. : dReach: δ-reachability analysis for hybrid systems, in Proc. International Conference of Tools and Algorithms for the Construction and Analysis of Systems, LNCS 9035, Springer, 2015, pp. 200–205.

[22] Kono, F., Koboyashi, T., Matsumoto, S. and Ueda, K. : Extensions for Large-Scale Simulation of Hybrid Systems Symbolic Simulator Hyrose, in Proc. 31st Symposium of Japan Society for Software Science and Technology, 2014, general session 7-3. (in Japanese)

[23] Lunze, J. : Handbook of Hybrid Systems Control : Theory, Tools, Applications, Cambridge University Press, 2009.

[24] Matsumoto, S., Kono, F., Kobayashi, T. and Ueda, K. : HyLaGI: Symbolic Implementation of a Hybrid Constraint Language HydLa, Electronic Notes in Theoretical Computer Science, 2015, Vol. 317, pp. 109–115.

[25] Matsumoto, S. and Ueda, K. : Symbolic Simulation of Parametrized Hybrid Systems with Affine Arithmetic, in Proc. 23rd International Symposium on Temporal Representation and Reasoning, 2016, pp. 4–11.

[26] Matsumoto, S. and Ueda, K. : Hyrose: A Symbolic Simulator of the Hybrid Constraint Language HydLa, Computer Software, Japan Society for Software Science and Technology, 2013, Vol. 30, No. 4, pp. 18–35. (in Japanese)

[27] Mimram, S., Bouissou, O. and Chapoutot, A. : HySon: Set-based Simulation of Hybrid Systems, in Proc. IEEE 23rd International Symposium on Rapid System Prototyping, 2012, pp. 79–85.

[28] Moore R. E., Kearfott R. B. and Cloud M. J. : Introduction to Interval Analysis, Society for Industrial and Applied Mathematics, 2009.

[29] Ñañez, P., Risso, N. and Sanfelice, R. G. : A Symbolic Simulator for Hybrid Equations, in Proc. 2014 Summer Simulation Multiconference, 2014, No. 18.

[30] Plätzer, A. and Quesel, J. D. : KeYmaera : A Hybrid Theorem Prover for Hybrid Systems, in Proc. International Joint Conference on Automated Reasoning, LNCS 5195, Springer, 2008, pp. 171–178.

[31] Ramdani, N., Meslem, N. and Candau, Y. : A Hybrid Bounding Method for Computing an Over-Approximation for the Reachable Set of Uncertain Nonlinear Systems, IEEE Transactions on Automatic Control, Vol. 54, No. 10, 2009, pp. 2352–2364.

[32] Rossi, F., Beek, P. V., Walsh, T. : Handbook of Constraint Programming, Elsevier Science, 2006.

[33] Tabuada, P. : Verification and Control of Hybrid Systems, Springer, 2009.

[34] Ueda, K., Matsumoto, S., Takeguchi, A., Hosobe, H. and Ishii, D. : HydLa : A High-Level Language for Hybrid Systems. Second Workshop on Logics for System Analysis, 2012, pp. 3–17.

[35] Ueda, K., Hosobe, H. and Ishii, D. : Declarative Semantics of the Hybrid Constraint Language, Computer Software, Japan Society for Software Science and Technology, Vol. 28, No. 1, 2011, pp. 306–311. (in Japanese)

[36] Wada, T., Matsumoto, S. and Ueda, K. : Simulation with formula manipulation and interval arithmetic by HydLa implementation, in Proc. 29th Annual Conference of the Japanese Society for Artificial Intelligence, 2015, 1E3-3. (in Japanese)

[37] Wolfram Research, Inc. : Mathematica,
http://www.wolfram.co.jp/products/mathematica/index.html.

[38] Wolfram Research, Inc. : Overview of Ordinary Differential Equations,
http://reference.wolfram.com/language
/tutorial/DSolveIntroductionToODEs.html

[39] Zeng, Y., Rose, C., Brauner, P., Taha, W., Masood, J., Philippsen, R., O'Malley, M. and Cartwright, R. : Modeling Basic Aspects of Cyber-Physical Systems, Part II, in Proc. 11th IEEE International Conference on Embedded Software and Systems, 2014, pp. 550–557.