Simulation and Verification of Hybrid Systems Based on Interval Analysis and Constraint Programming

Daisuke Ishii

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Information and Computer Science,
Graduate School of Science and Engineering,
Waseda University

February 2010 (Version 1.1.2)

Keywords: Hybrid Systems, Constraint Programming, Interval Analysis, Simulation, Reachability Analysis.

Abstract

Hybrid systems are systems consisting of discrete changes and continuous changes over time. Various systems in which computers reliably interact with their physical environment are modeled as hybrid systems. Simulation and verification of hybrid systems are done by integrating the computation of continuous dynamics and discrete changes, and by handling the uncertainties and computation errors. However, the computation of hybrid systems is often difficult, and may produce qualitatively wrong results, especially when the systems are described by nonlinear ordinary differential equations (ODEs) and nonlinear algebraic equations. This thesis is intended to provide a framework for nonlinear hybrid systems based on interval analysis and constraint programming.

The detection of discrete changes in hybrid systems plays a significant role in the simulation and verification. We formulate the problem as a *hybrid constraint system* (HCS), which consists of *instantaneous constraints*, *continuous constraints* on trajectories (i.e., continuous functions over time) and *guard constraints* on states causing discrete changes. We implement a technique for solving HCSs by coordinating (i) an interval-based solving technique for nonlinear ODEs, and (ii) a constraint programming technique that reduces the interval enclosures of solutions. The technique generates a set of boxes smaller than a specified size that enclose the theoretical solution. Our technique employs the interval Newton method to accelerate the reduction of interval enclosures while guaranteeing that the enclosure contains a solution, and it reliably solves HCSs with nonlinear constraints.

Next, we present a bounded model checking method for hybrid systems. It translates a reachability problem of a nonlinear hybrid system into a predicate logic formula involving arithmetic constraints, and checks the satisfiability of the formula based on the satisfiability modulo theories (SMT) method. We tightly integrate (i) an incremental propositional satisfiability (SAT) solver to enumerate possible sets of constraints and (ii) an interval-based solver for HCSs to solve the constraints described in the formulas. The HCS solver verifies the occurrence of a discrete change by computing a set of boxes that enclose continuous states that may cause a discrete change. We exploit the existence property of a unique solution in the boxes computed by the HCS solver as (i) a proof of the reachability of a model, and (ii) a guide in the over-approximation refinement procedure. Our implementation, called `hydlogic`, successfully handles several examples including those with nonlinear constraints.

Acknowledgments

I thank my supervisor Prof. Kazunori Ueda for encouraging me to publish this dissertation. Of course, there has been more than encouragement. His helpful comments and suggestions have been essential for completing each chapter of this dissertation. I should also thank Prof. Hiroshi Hosobe for all the support including many valuable comments on the drafts of this work. The author thank the thesis committee members, Prof. Yasuo Matsuyama, Prof. Shin'ishi Oishi, Prof. Toshiharu Sugawara, et al., for their patient support to improve the thesis. I would like to appreciate all the members at Ueda Laboratory in Waseda University for encouraging me. In particular, I should thank the members of the Udraw and HydLa projects for many profitable hours of discussions. I have been fortunate having the opportunity to work with people at LINA, Université de Nantes. I should like to thank Dr. Alexandre Goldsztejn, Dr. Marc Christie, and Dr. Christophe Jermann who provide the basic materials for this study. Finally, I am grateful to my parents for their support and encouragement over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the challenges in building software and hardware components is to design and analyze components so that computer programs reliably interact with their physical environment [74]. Such systems are modeled as *hybrid systems* [73, 54] which consist of discrete and continuous changes over time. Practical examples of hybrid systems include embedded controllers for automobiles [2, 4], electric circuits handling both analog signals and binary switches [38, 41], and models in molecular systems biology [25].

1.1 Modeling of Hybrid Systems

As described by van der Schaft [73] and Carloni et al. [12], various frameworks for modeling hybrid systems have been proposed for different purposes such as simulation, verification, and control.

- *Hybrid automata (HA)* [45] are developed by extending discrete transition systems. Methods for abstracting HA into discrete systems and for model checking about the systems are developed (see [3, 76] for the survey).
- *Constraint-based frameworks* model hybrid systems based on expressions in the form of logical formula involving arithmetic equations. The targets of these frameworks include:
  - the models that involve uncertainties (e.g., equations with parameters) such that the models are described with constraints (generic constraint programming frameworks have been extended for modeling hybrid systems [39, 46]);
  - the models described by the disjunction of several arithmetic equations that are an extention of continuous dynamical systems (e.g., event-flow formulas [73] and differential algebraic equations); and
  - an intermediate expression of models used in the frameworks for rea-

soning hybrid systems by applying logical satisfiability checking methods [4, 27].

1.2 Detection of Discrete Changes in Hybrid Systems

The detection of states that cause discrete changes plays a significant role in the simulation and verification of hybrid systems. The problem is described by the conjunction of an ODE (ordinary differential equation) and the condition for a discrete change (guard constraint). We also consider problems that involve uncertainties: for example, an initial value is given as an interval.

Many techniques for solving the problem (e.g., Reference [24]) involve numerical computation. To solve efficiently, we should reduce the number of iterations to simulate continuous changes, whereas conditions for discrete changes should be examined precisely. Because the techniques may compute unexpected results due to computation errors, various workarounds have been investigated. For example, Park and Barton [64] handled the discontinuity sticking problem, which is the problem of detecting the same discrete event immediately after a discrete change.

A reliable approach for the computation is interval analysis, which guarantees that a set of tight intervals or boxes encloses the solution. Uncertain parameters in problems are described by intervals. Many systems for hybrid systems apply interval-based techniques to the simulation [46, 61] and verification [67]. However, existing methods have difficulties in the efficient computation, especially when problems involve nonlinear constraints. As in the numerical approach, we should reduce the number of intervals to enclose nonlinear continuous changes, which easily increases exponentially, and at the same time, states causing discrete changes should be enclosed by tight intervals.

1.3 Simulation and Verification of Hybrid Systems

Various frameworks have been developed for the simulation and verification of hybrid systems. Most of the existing tools for simulation are based on numerical methods. Carloni et al. [12] have pointed out that the simulators such as MATLAB/Simulink/Stateflow [55] and Dymola [21] are not reliable enough, especially when models involve singular behaviors such as Zeno behaviors and chatterings.

Verification frameworks for hybrid systems are dedicated to the *reachability analysis* of hybrid systems. Many tools primarily analyze the safety properties of various models by translating an analysis of a model into an equivalent problem (e.g., a discrete transition system [3, 76] or a predicate logic formula [4, 27]) consisting of a finite set of discrete states to apply model checking techniques.

When translations are difficult, some reachability computations (e.g., for models with affine constraints) may be abstracted by the *over-approximation* of the constraints in models (e.g., with boxes [67] or polyhedra [29]). In such a case, the properties that are to be verified should be preserved in an over-approximation. Nevertheless, efficient over-approximation of continuous trajectories described by nonlinear constraints is still an active research topic.

1.4 Our Goal and Contributions

This thesis proposes frameworks for the simulation and verification of nonlinear hybrid systems.

1.4.1 Interval-based Solving of Hybrid Constraint Systems

We propose *hybrid constraint systems* (HCSs) to describe the problem of detecting discrete changes by constraints. HCSs allow us to combine intervals and equations of real numbers including ODEs in a clean and natural way [46]. HCSs serve as a key component in the simulation and verification of hybrid systems. An HCS consists of

- *instantaneous constraints* on initial states,
- *continuous constraints* on trajectories over time, and
- *guard constraints* that describe the boundaries in the continuous state space.

To solve an HCS, we translate the original HCS into a *box-consistent* HCS whose domain is an interval enclosure of the solution satisfying the certain accuracy.

We develop a technique for solving HCSs by integrating (i) the interval-based nonlinear ODE solving method by Nedialkov et al. [60, 59] and (ii) the interval-based constraint programming framework by van Hentenryck, Benhamou, Granvilliers et al. [80, 6, 36]. The technique generates a set of boxes smaller than a specified size that enclose the theoretical solution.

The proposed technique employs the interval Newton method for the quadratic convergence of the reduction of boxes, and for guaranteeing that the boxes contain a solution. The method uses an interval Newton operator derived from constraints in HCSs. Experimental results show the efficiency of the method in simulating nonlinear hybrid systems.

1.4.2 Encoding Hybrid Systems into Predicate Logic Formulas

We develop a bounded model checking (BMC) framework for hybrid systems. In this framework, we encode a reachability problem of a nonlinear hybrid system into a predicate logic formula involving arithmetic constraints, and checks the satisfiability of the formula to verify the original system.

As inputs to the encoding scheme that we propose, we consider models described by HA and a constraint-based language called Tiny HCC. The scheme encodes models into predicate logic formulas involving the constraints of HCSs. The encoding is based on the semantics of the languages that defines the bounded *executions* of models described by the languages. We formalize the semantics by translating HA and Tiny HCC programs into an abstract machine called *real-time transition systems* (RTTSs), and then describing executions of models as *hybrid trajectories*.

1.4.3 Reliable Simulation and Verification of Nonlinear Hybrid Systems

Once a model is translated into a predicate logic formula, the satisfiability of the formula is checked by a satisfiability modulo theories (SMT) method. We tightly integrate

- an incremental SAT (propositional satisfiability) solver to enumerate the possible sets of constraints and
- the interval-based solver for HCSs to solve the constraints described in the formulas.

We adopt the existence property of a unique solution in the boxes computed by the HCS solver as (i) a proof of the reachability of a model, and (ii) a guide in the over-approximation refinement procedure. Our implementation called `hydlogic` successfully handled several examples including those with nonlinear constraints.

1.5 Outline of the Thesis

- Chapter 2 introduces some basics of interval analysis. It provides the definitions of real intervals, machine-representable intervals, boxes, interval extensions, and so on. Then, we introduce the interval Newton method and interval-based methods for solving initial value problems for ODEs, which serve as components in the proposed method in Chapter 5.
- Chapter 3 presents some backgrounds of constraint systems whose domain of variables is real numbers. We provide a simple formalization of *real constraint systems* (RCSs) and corresponding notions such as entailment and consistency. Subsequently, we describe methods based on interval analysis for checking the entailment and consistency. The consistency checking method is based on the *box-consistency* notion, which formalizes an interval-based approximation of solutions.
- Chapter 4 describes hybrid systems. We first introduce two simple structures RTTSs and hybrid trajectories to illustrate how hybrid systems behave. Next, we describe two modeling languages for hybrid systems, HA and the Tiny HCC language, by translating them into RTTSs.
- Chapter 5 presents new formulations of *continuous constraint systems* (CCSs) and hybrid constraint systems (HCSs). We first define CCSs and HCSs by extending RCSs, and introduce the box-consistency for HCSs. Next, we describe an interval-based technique for checking the satisfiability of constraints in an HCS. The technique reliably solves HCSs with nonlinear constraints by coordinating interval-based solvers for ODEs and RCSs. We explain how we apply the interval Newton method to solve HCSs.
- Chapter 6 proposes a method for bounded reachability analysis of nonlinear HA. We describe the SMT-based method that encodes models into predicate logic formula, and then checks the satisfiability of the formula using a SAT solver and the HCS solver in Section 5.
- Chapter 7 describes the previous research on hybrid systems especially from the viewpoints of the detection of discrete changes, modeling frameworks, and reachability analysis.
- Finally, in Chapter 8, we conclude and indicate topics for further research.

Table 1.1 Computational environment for experiments.

Processor	1.86GHz Intel Core 2 Duo (with a single core activated)
L2 cache	2MB
RAM	2GB
OS	Linux 2.6.26
C/C++ compiler	gcc 4.3.2

1.6 Prerequisites

This thesis is intended to be self-contained. We assume a basic knowledge of set theory, logic, and programming languages. We use the following notations in the description below:

Definition 1 (basic notation) For a set S of distinguishable objects, $\mathcal{P}(S)$ denotes the powerset of S. For a tuple $V = (v_1, \ldots, v_n)$ of not necessarily distinct objects, $V.i$ denotes the i-th component v_i. For sets $S_1, \ldots, S_n$, and a relation $rel \subseteq S_1 \times \cdots \times S_n$, $\pi_i(rel)$ denotes the i-th projection of rel onto S_i. For a functions $\phi : S_1 \to S_2$, $dom(\phi) = S_1$ and $img(\phi) = S_2$. We represent the set of bounded continuous functions $\{\phi \mid dom(\phi) = S_1 \wedge img(\phi) = S_2\}$ by $\mathcal{M}(S_1, S_2)$. □

For the experiments described in each chapter, we work in the computational environment described in Table 1.1.

Chapter 2

Interval Analysis

Interval analysis [57, 58, 62, 51] provides a rigor in computation with real numbers. It extends *numerical analysis* by replacing machine-representable *floating-point numbers* with machine-representable *intervals*. We first introduce the basic notions such as intervals, boxes, and interval extensions (Section 2.1). We then introduce the interval Newton methods (Section 2.2) and interval-based methods for solving ODEs (Section 2.3).

2.1 Intervals

This section defines intervals and introduces basic interval operations. We define herein closed and open real intervals and the properties of intervals.

Definition 2 (real interval) Let $\mathbb{R}$ denote the set of real numbers, and let $\mathbb{R}^\infty$ denote $\mathbb{R} \cup \{-\infty, +\infty\}$. A *closed real interval* I with the *lower bound* $\underline{I}$ and the *upper bound* $\overline{I}$ is defined as

$$I = [\underline{I}, \overline{I}] = \{r \in \mathbb{R} \mid \underline{I} \leq r \leq \overline{I}\},$$

where $\underline{I}$ and $\overline{I}$ are real numbers ($\underline{I} \leq \overline{I}$). $\mathbb{IR}$ denotes the set of closed real intervals, A closed real interval will be called an *interval*, when no confusion may arise. An interval I where $\underline{I} = \overline{I} = r$ is also denoted by $[r]$. We call an interval I *canonical* if $\underline{I} < \overline{I}$. Given a canonical interval I, an *open real interval* or an *internal* of I is defined as

$$(\underline{I}, \overline{I}) = \mathrm{int}(I) = \{r \in \mathbb{R} \mid \underline{I} < r < \overline{I}\},$$

and *half-open real intervals* $[\underline{I}, \overline{I})$ and $(\underline{I}, \overline{I}]$ are defined as $\{\underline{I}\} \cup (\underline{I}, \overline{I})$ and $(\underline{I}, \overline{I}) \cup \{\overline{I}\}$, respectively. Note that open and half-open real intervals are also called *intervals* when the distinction is not important. The *universal interval* $(-\infty, +\infty)$ is

the set $\mathbb{R}$, and the *empty interval* $\emptyset$ is the empty set (these intervals are elements of $\mathbb{IR}$).

For non-empty intervals I and I',

- $\mathrm{w}(I)$ denotes the *width* $\overline{I} - \underline{I}$,
- $\mathrm{m}(I)$ denotes the *center* $(\overline{I} + \underline{I})/2$,
- $|I|$ denotes the *absolute value* $\max\{|\underline{I}|, |\overline{I}|\}$, and
- $I \uplus I'$ denotes the *hull* $\{r \in \mathbb{R} \mid \min(\underline{I}, \underline{I'}) \leq r \leq \max(\overline{I}, \overline{I'})\}$. □

Example 1 $I = [-1.5, 10]$, $[0] = [0, 0]$, and $\mathrm{int}(I) = (-1.5, 10)$ are intervals. I is canonical. $\underline{I} = -1.5$, $\mathrm{w}(I) = 11.5$, $\mathrm{m}(I) = 4.25$, and $|I| = 10$. We may abbreviate an interval $[3.141, 3.142]$ as $3.14[1, 2]$. □

Definition 3 (box) An n-dimensional *box* or *interval vector* is defined as a tuple of n intervals $B = (I_1, \ldots, I_n)$, where $n \in \mathbb{N}$ and $I_1, \ldots, I_n \in \mathbb{IR}$. The *empty box* $\emptyset$ is a box for which a component is the empty interval $\emptyset$. $\mathbb{IR}^n$ denotes the set of n-dimensional boxes.

For non-empty boxes B and B' in $\mathbb{IR}^n$,

- $\mathrm{w}(B)$ denotes the *width* $\max\{\mathrm{w}(B.i)) \mid i \in \{1, \ldots, n\}\}$ ($B.i$ denotes the i-th component of B), and
- $B \cap B'$ denotes the *intersection* $(B.1 \cap B'.1, \ldots, B.n \cap B'.n)$. □

Example 2 $B = ([2, 3], [-1.5, 10], [0], (-\infty, +\infty))$ is an instance of $\mathbb{IR}^4$. $B.2 = [-1.5, 10]$, and $\mathrm{w}(B) = +\infty$. □

2.1.1 Machine-Representable Intervals

We consider various sets of machine-representable *floating-point numbers* $\mathbb{F} \subset \mathbb{R}$ that conform to the IEEE 754 standard [47].

Definition 4 (floating-point number) Let $b \in \{2, 10\}$ be the radix, $e_{min}, e_{max} \in \mathbb{Z}$ be the bounds for ranging exponents, and $n \in \mathbb{N}$ be the number of digits (precision) in the significand. A system of *floating-point numbers* $\mathbb{F}_{b, e_{min}, e_{max}, n}$ consists of the following objects:

- Triples $\langle s, e, m \rangle$ representing the number

$$(-1)^s \cdot b^e \cdot m,$$

where $s \in \{0, 1\}$, $e \in \mathbb{Z}$ ($e_{min} \leq e \leq e_{max}$), and m is a digit string of the form $d_1.d_2 d_3 \cdots d_n$ ($d_i \in \{0, \ldots, b-1\}$).

- Positive and negative infinities $+\infty$ and $-\infty$.
- Not a number symbol NaN.

Let $\mathbb{F}$ be a floating-point system. For $r \in \mathbb{R}$ and $r' \in \mathbb{F}$, we use the following notations:

$$\begin{aligned}
\nabla r &= \max\{r'' \in \mathbb{F} \mid r'' \leq r\},\\
\triangle r &= \min\{r'' \in \mathbb{F} \mid r'' \geq r\},\\
r'^{-} &= \max\{r'' \in \mathbb{F} \mid r'' < r'\},\\
r'^{+} &= \min\{r'' \in \mathbb{F} \mid r'' > r'\}.
\end{aligned}$$

□

Example 3 3.141592 is an instance of $\mathbb{F}_{10,-95,96,7}$. The single-precision system $\mathbb{F}_s = \mathbb{F}_{2,-126,127,(23+1)}$ and the double-precision system $\mathbb{F}_d = \mathbb{F}_{2,-1022,1023,(52+1)}$ are widely used on computers. □

Definition 5 (machine-representable interval) Let $\mathbb{F}$ be a system of floating-point numbers. A *machine-representable interval* is a real interval I, where $\underline{I}$ and $\overline{I}$ are elements of $\mathbb{F}$. $\mathbb{IF}$ denotes the set of machine-representable intervals based on $\mathbb{F}$. $\mathbb{IF}^n$ denotes the set of n-dimensional boxes based on $\mathbb{F}$. □

We represent a set of intervals by $\mathbb{I}$ when the distinction between the floating-point systems used for bounds is not important.

2.1.2 Interval Enclosures, Approximations, and Extensions

Consider a relation $rel \subseteq \mathbb{R}^n$ and a system $\mathbb{F}$ of floating-point numbers. A box in $\mathbb{IF}^n$ that conservatively approximates *rel* is called an *interval enclosure*.

Definition 6 (interval enclosure) A box $(I_1, \ldots, I_n) \in \mathbb{IF}^n$ is an *interval enclosure* of $rel \subseteq \mathbb{R}^n$ when $rel \subseteq I_1 \times \cdots \times I_n$ holds. □

We have various systems of enclosures for representing relations in $\mathbb{R}^n$ such as $\mathbb{IR}^n$, and $\mathbb{IF}^n$. The smallest (with respect to the set inclusion) interval enclosure in a system of enclosures is called the *interval approximation*.

Definition 7 (interval approximation) Let $\mathbb{F}$ be a system of floating-point numbers. An *interval approximation* (or *hull*) $apx_{\mathbb{IF}^n}(rel) \in \mathbb{IF}^n$ of a relation $rel \subseteq \mathbb{R}^n$ is defined as

$$apx_{\mathbb{IF}^n}(rel) = \bigcap_{B \in BS} B,$$

where $BS = \{(I_1, \ldots, I_n) \in \mathbb{IF}^n \mid \forall (r_1, \ldots, r_n) \in rel\ ((r_1 \ldots, r_n) \in I_1 \times \cdots \times I_n)\}$. □

Another interval-based approximations for functions and relations are defined as *interval extensions*.

Definition 8 (interval extension) Let $\mathbb{IF}$ be a system of floating-point numbers. For a function $f : \mathbb{R}^m \to \mathbb{R}^n$, a function $F : \mathbb{IF}^m \to \mathbb{IF}^n$ is called an *interval extension* of f if and only if

$$\forall I_1 \in \mathbb{IF} \cdots \forall I_m \in \mathbb{IF}\ \forall r_1 \in I_1 \cdots \forall r_m \in I_m\ \forall i \in \{1, \ldots, n\}$$
$$(f(r_1, \ldots, r_m).i \in F(I_1, \ldots, I_m).i).$$

For a relation $rel \subseteq \mathbb{R}^n$, a relation $R \subseteq \mathbb{IF}^n$ on intervals is called an *interval extension* of *rel* if and only if

$$\forall I_1 \in \mathbb{IF} \cdots \forall I_n \in \mathbb{IF}$$
$$(\exists r_1 \in I_1 \cdots \exists r_n \in I_n\ ((r_1, \ldots, r_n) \in rel) \Rightarrow (I_1, \ldots, I_n) \in R).$$

□

Example 4 Let I_1 and I_2 be intervals in $\mathbb{IF}$. The *natural interval extension* of four operators is implemented as follows:

$$I_1 + I_2 = [\nabla(\underline{I_1} + \underline{I_2}), \triangle(\overline{I_1} + \overline{I_2})],$$
$$I_1 - I_2 = [\nabla(\underline{I_1} - \underline{I_2}), \triangle(\overline{I_1} - \overline{I_2})],$$
$$I_1 \cdot I_2 = [\nabla\min\{\underline{I_1} \cdot \underline{I_2}, \underline{I_1} \cdot \overline{I_2}, \overline{I_1} \cdot \underline{I_2}, \overline{I_1} \cdot \overline{I_2}\}, \triangle\max\{\underline{I_1} \cdot \underline{I_2}, \underline{I_1} \cdot \overline{I_2}, \overline{I_1} \cdot \underline{I_2}, \overline{I_1} \cdot \overline{I_2}\}],$$
$$I_1 \,/\, I_2 = I_1 \cdot [\nabla(1/\overline{I_2}), \triangle(1/\underline{I_2})] \qquad \text{if } 0 \notin I_2.$$

The *natural interval extension* of a function $f(x, y) = (x + y)^2/2$ is formed as $F(X, Y) = (X + Y) \cdot (X + Y)/[2]$, where operators $+, \cdot$, and $/$ are the natural interval extensions. Note that interval functions $F(X, Y)$ and $(X \cdot X + [2] \cdot X \cdot Y + Y \cdot Y)/[2]$ are not equivalent. □

2.1.3 Implementations

Implementations of intervals are available for various platforms. For example, `Gaol` [33], `PROFIL/BIAS` [52], and Boost Interval Arithmetic Library [56] are implementations in C/C++, and INTLAB [69] is an implementation built on top of the MATLAB system.

Table 2.1 Computation results from `Gaol` and `PROFIL/BIAS`.

library	problem	result	cycles
`Gaol`	prism	$1.5867066805824[69, 72]$	14553
	g&p	$[-56254330, 94177270]$	9823
`PROFIL/BIAS`	prism	$1.5867066805824[687, 719]$	8521
	g&p	$[-56254330, 94177270]$	7226

Example 5 `Gaol` and `PROFIL/BIAS` provide the object classes of intervals in $\mathbb{IF}_d$, interval extensions of four operators and elementary functions, operators for accessing the various properties of intervals, and so on. `Gaol` also provides relational functions such as $acos\_rel(I_2, I_1) = apx_{\mathbb{IF}^n}(\{r_1 \in I_1 \mid \exists r_2 \in I_2\ (r_2 = \cos(r_1))\})$.

We computed interval extensions of the following functions to compare the performance and the accuracy of the interval extensions.

- The following function represents the index of refraction of a prism taken from Reference [58]:

$$f(\alpha, \delta) = \frac{\sin((\alpha + \delta)/2)}{\sin(\alpha/2)},$$

  where $\alpha \in [\pi/3]$ is the angle, and $\delta \in [\pi/4]$ is the minimum deviation angle.
- A global optimization test function by Goldstein and Price (g&p) taken from the `Gaol` package is

$$f(x, y) = (1 + (x + y)^2 \cdot (19 - 14 \cdot x + 3 \cdot x^2 - 14 \cdot y + 6 \cdot x \cdot y + 3 \cdot y^2))$$
$$\cdot(30 + (2 \cdot x - 3 \cdot y)^2 \cdot (18 - 32 \cdot x + 12 \cdot x^2 + 48 \cdot y - 36 \cdot x \cdot y + 27 \cdot y^2)),$$

  for $x \in [-2, 2]$ and $y \in [-2, 2]$.

The results and the CPU clock cycles of the computation are illustrated in Table 2.1. `PROFIL/BIAS` computed somewhat efficiently than `Gaol`. The accuracy of the results are almost the same. □

2.2 Interval Newton Method

The operator provided below gives a basis for *interval Newton methods*.

Theorem 1 (univariate interval Newton operator) Given an equation $h(t) = 0$, where $h : \mathbb{R} \to \mathbb{R}$ is a differentiable function, a solution of the equation

in an interval I is also included in an interval obtained by the following *interval Newton operator*

$$N_{H,\dot{H}}(I) = I \cap \left([\mathrm{m}(I)] - \frac{H([\mathrm{m}(I)])}{\dot{H}(I)} \right),$$

where H and $\dot{H}$ are interval extensions of h and its derivative. □

Proof. See Theorem 8.1 of Reference [58]. ■

The operator $N_{H,\dot{H}}(I)$ is defined if and only if $0 \notin \dot{H}(I)$ holds. The (univariate) *interval Newton method* iteratively refines an interval enclosure by using the operator above. Successive applications of $N_{H,\dot{H}}(I)$ will converge because the operator takes an intersection in the operator. The fixpoint is denoted by $N^*_{H,\dot{H}}(I)$.

By taking a sufficiently small enclosure I of a solution, an application of $N_{H,\dot{H}}(I)$ will reduce I unless it has reached a fixpoint. An important sub-product of the interval Newton method is that it guarantees the existence of the unique solution within an interval.

Theorem 2 (existence of unique solution) If the condition $N_{H,\dot{H}}(I) \subseteq \mathrm{int}(I)$ holds, a unique solution of the equation $h(t) = 0$ exists within $N_{H,\dot{H}}(I)$. □

Proof. See Theorem 8.4 of Reference [58]. ■

2.3 Interval-based Solving of ODEs

This section describes interval-based techniques for solving initial value problems for ODEs.

Definition 9 (initial value problem for ODE) A *continuous trajectory* ϕ is a vector-valued differentiable function over time $\mathbb{R}_{\geq 0} \to \mathbb{R}^n$ ($t_0 \in \mathbb{R}_{\geq 0}$). Such ϕ is specified by an *initial value problem* for an *ordinary differential equation* (IVP-ODE) formed by the conjunction of equations

$$\dot{\phi}(\tau) = f(\phi(\tau)) \wedge \phi(t_0) = \phi_0,$$

where $\dot{\phi}(\tau)$ denotes $d\phi(\tau)/d\tau$, $\phi_0 \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^n$ (assuming Lipschitz continuity).

Given an IVP-ODE, a *solution* denoted by ϕ_{t_0,ϕ_0} is a continuous trajectory that satisfies the IVP-ODE. □

When considering the interval-based techniques for IVP-ODEs, we parameterize the initial values for an ODE by specifying the initial time points t_0 and initial values ϕ_0 by a box. The techniques then compute interval extensions of all the possible solutions.

Definition 10 (interval extension of solutions for IVP-ODE) Given an ODE and an initial domain $(T_0, \Phi_0) \in \mathbb{I}^{n+1}$, where $\underline{T_0} \geq 0$, an *interval extension* Φ_{T_0,Φ_0} of the set of solutions ϕ_{t_0,ϕ_0} for IVP-ODE satisfies the following condition

$$\forall t_0 \in T_0\, \forall \phi_0 \in \Phi_0\, \forall t \in T\, \forall i \in \{1, \ldots, n\}\ (\phi_{t_0,\phi_0}(t).i \in \Phi_{T_0,\Phi_0}(T).i),$$

where T is a time interval such that $\underline{T} \geq \overline{T_0}$. It is denoted by $\Phi_{T_0,\Phi_0} : \mathbb{I} \to \mathbb{I}^n$. □

We employ the existing method VNODE proposed by Nedialkov et al. [60, 59] to solve IVP-ODEs based on interval analysis. For an ODE and an initial domain (T_0, Φ_0), the solving process of VNODE computes the interval extensions of solutions for the IVP-ODE $\Phi_{T_0,\Phi_0} : \mathbb{I} \to \mathbb{I}^n$. Given a time interval $T \in \mathbb{I}$, VNODE computes a box $\Phi_{T_0,\Phi_0}(T)$ that encloses every trajectory over T. As a by-product of the computation, VNODE computes an enclosure $\Phi_{T_0,\Phi_0}([\underline{T_0}, \overline{T}])$, because the computation is done iteratively from the initial value.

In the method proposed in Chapter 5, we also need an interval extension of the derivative of the solution $\dot{\Phi}_{T_0,\Phi_0} : \mathbb{I} \to \mathbb{I}^n$, which encloses the derivative of the solution. Let $T_1 \in \mathbb{I}$ be a time interval. Then $\dot{\Phi}_{T_0,\Phi_0}(T_1)$ will be implemented as the computation of a function $F(\Phi_{T_0,\Phi_0}(T_1))$, which is an interval extension of $f(\phi(t))$ in the ODE. Here, $\Phi_{T_0,\Phi_0}(T_1)$ is computed by VNODE beforehand.

The computation of VNODE is based on a Taylor series coefficients generated by automatic differentiation. Each integration step of VNODE consists of the following procedures ($i \in \{1, \ldots, m\}$):

1. Compute a time interval T_i and an *a priori* enclosure $\Phi_{T_{i-1},\Phi_{i-1}}(T_i)$ of solutions. This enclosure guarantees the existence of a unique solution within it.
2. Using $\Phi_{T_{i-1},\Phi_{i-1}}(T_i)$, compute a *tight* enclosure $\Phi_i = \Phi_{T_0,\Phi_0}([\overline{T_i}])$.

Step 1 is applied to obtain a rough enclosure of solutions using a method based on high-order Taylor series expansion. The method computes T_i based on the parameters below:

- The maximum order $k \in \mathbb{N}$ to compute Taylor expansion.

- The minimum step width $h_{min} \in \mathbb{R}_{>0}$.
- The tolerance values *atol* and *rtol* in $\mathbb{R}_{>0}$.

The computation in Step 2 is based on the interval Hermite-Obreschkoff method. In our methods, we utilize a set of a priori enclosures computed in Step 1.

Example 6 We solved the following IVP-ODEs by using the `VNODE-LP` solver [59] that implements VNODE. We set `Gaol` or `PROFIL/BIAS` as an underlying interval library in the computation. The parameters were set as $k = 20, h_{min} = 10^{-11}$, $atol = 10^{-20}$, and $rtol = 10^{-20}$.

- A problem describing a particle falling down by the gravity acceleration and the air resistance:

$$\begin{aligned} \dot{\phi_p}(\tau) &= \phi_v(\tau), \\ \dot{\phi_v}(\tau) &= -G + K \cdot \phi_v(\tau)^2, \\ (\phi_p(0), \phi_v(0)) &\in [1, 1.1] \times [-4.1], \end{aligned}$$

  where ϕ_p and ϕ_v represent the position and velocity of the particle, $G = [9.8]$ and $K = [10^{-3}]$.
- The Lorenz equation:

$$\begin{aligned} \dot{\phi_1}(\tau) &= 10 \cdot (\phi_2(\tau) - \phi_1(\tau)), \\ \dot{\phi_2}(\tau) &= \phi_1(\tau) \cdot (28 - \phi_3(\tau)) - \phi_2(\tau), \\ \dot{\phi_3}(\tau) &= \phi_1(\tau) \cdot \phi_2(\tau) - 8/3 \cdot \phi_3(\tau), \\ (\phi_1(0), \phi_2(0), \phi_3(0)) &= (15, 15, 36). \end{aligned}$$

The results are illustrated in Table 2.2, and Figures 2.1 and 2.2. In the third column of Table 2.2, each row indicates the enclosures of $\phi_p(10^4)$ and $\phi_1(24)$, respectively. Boxes in the figures are the a priori enclosures of solutions. □

Table 2.2 Computation results by `VNODE-LP` with `Gaol` and `PROFIL/BIAS`.

library	problem	result	time (ms)
`Gaol`	falling	$-989295.[828, 928]$	70
	lorenz	$4.2[146, 386]$	927
`PROFIL/BIAS`	falling	$-989295.[828, 928]$	75
	lorenz	$4.2[183, 349]$	970

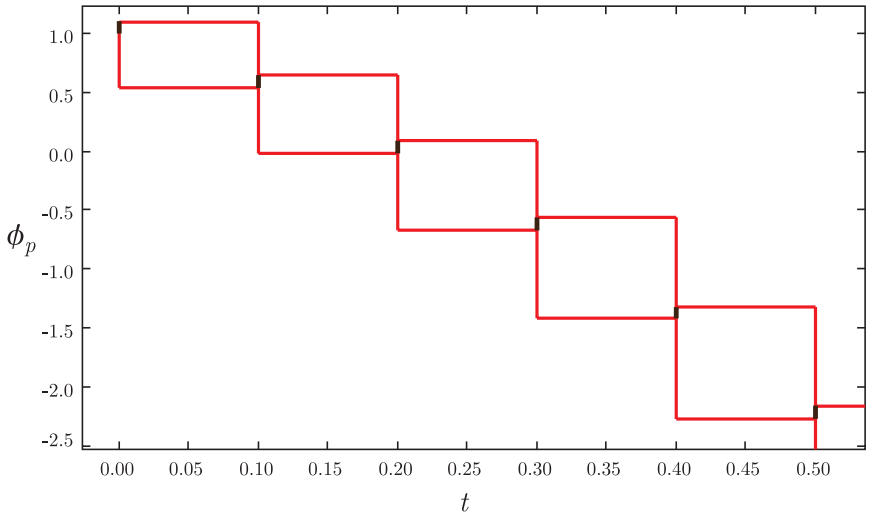

Figure 2.1 Computation result of falling particle by `VNODE-LP`.

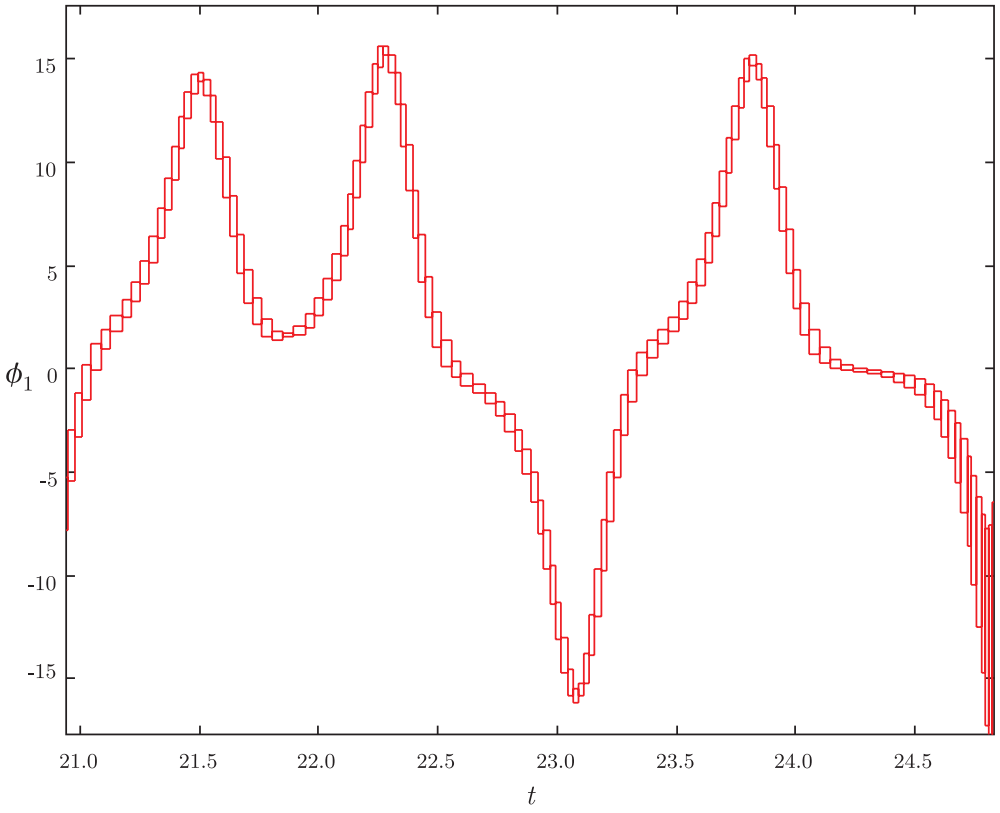

Figure 2.2 Computation result of Lorenz equation by `VNODE-LP`.

Chapter 3

Real Constraint Systems

In this chapter, we introduce the basics of *constraint systems* (or *constraint satisfaction problems*) [20, 68]. Constraint systems formalize a framework for computing with partial information. Elements of data are *constraints* that hold the possible values for a set of variables. We consider various properties on constraints such as *entailment* to check whether a constraint can be derived from another constraint, and *satisfiability* to check whether there exists a valuation of variables with which a constraint can be interpreted as *true*. In this thesis, we work on constraint systems with numerical domains. In general, these constraint systems are not machine-representable because they have infinite numbers of values within the domains. Thus, we approximate the checking of entailment or consistency for those that can be computed incompletely yet efficiently.

We begin with the definition of generic constraint systems.

Definition 11 (constraint system) A simple *constraint system* is a structure $\langle x, \mathcal{D}, \mathcal{C} \rangle$, where $x = (x_1, \ldots, x_n)$ is a tuple of n variables, $\mathcal{D} = D_1 \times \cdots \times D_n$ is a *domain*, which is the Cartesian product of n collections of objects, and $\mathcal{C}$ is a (multi-)set of *constraints* $\mathcal{C} = \{c_1, \ldots, c_p\}$. A constraint c_i $(i \in \{1, \ldots, p\})$ defined on x is a relation $c_i \subseteq \mathcal{D}$.

A *valuation* is a map of the form $x \mapsto v$, where $v = (v_1, \ldots, v_n)$ is an n vector of objects in $\mathcal{D}$. When v belongs to a constraint c, the valuation $x \mapsto v$ *satisfies* c, and $v \models_{\mathcal{D}} c$ denotes this relationship. A *solution* is a valuation $x \mapsto v$ that satisfies every constraint in the constraint system, i.e., $\forall c \in \mathcal{C}\ (v \models_{\mathcal{D}} c)$. Let c and c' be constraints. c *entails* c' when $c \subseteq c'$ holds, and $c \vdash c'$ denotes this entailment relation. $c \nvdash c'$ denotes that $c \vdash c'$ does not hold. $\models_{\mathcal{D}} c$ denotes that $\mathcal{D} \subseteq c$ holds for a constraint c. □

We use two special constraints, that are not contained in $\mathcal{C}$, the *valid* constraint $\mathbf{true} = \mathcal{D}$, and the *contradiction* $\mathbf{false} = \emptyset$.

Remark 1 Constraint systems are defined more generally and differently in [71]. In general, the entailment relation is not only defined by inclusion but is also defined by a transitive property between two relationships. The constraints $c \subseteq \mathcal{D}$ in the above definition are called *tokens*, and constraints are defined as closures of tokens, i.e., $\{D \subseteq \mathcal{D} \mid c \vdash D\}$. □

In the following, we provide an instance of constraint systems called *real constraint systems* (RCSs) where the domain of variables is the set of real numbers (Section 3.1). Next, we describe how we can represent the RCSs based on intervals and handle constraints of the RCSs (Section 3.2 and 3.3). Our approach is to use techniques based on interval analysis for enclosing solutions by intervals with rounding errors. We describe several algorithms for computing entailment and consistency in this approach.

3.1 Real Constraint Systems

In this section, we define real constraint systems (RCSs) involving a real vector-valued variable consisting of n real-valued variables $x = (x_1, \ldots, x_n)$. A *real constraint* on x is a subset of the domain $\mathbb{R}^n$ of x.

Definition 12 (real constraint system) A *real constraint system* (RCS) is a constraint system $\langle x, \mathcal{D}, \mathcal{C} \rangle$ consisting of an n-dimensional vector of variables $x = (x_1, \ldots, x_n)$, a domain $\mathcal{D} \subseteq \mathbb{R}^n$, and a set of *real constraints* $\mathcal{C} = \{c_1, \ldots, c_p\}$, where $c_i \subseteq \mathcal{D}$. □

We describe a real constraint by a *real formula* in the conjunction form of nonlinear equations and inequalities.

Definition 13 (real formula) The syntax of *real formulas*, i.e., the formulas in *nonlinear arithmetic*, is defined by the following rules:

$$F ::= F \wedge F \mid A \mid \textbf{true} \mid \textbf{false}$$
$$A ::= T \;\; op \;\; T$$
$$T ::= id \mid constant \mid -T \mid T + T \mid T \cdot T \mid 1/T \mid T^T \mid elem\_func(T)$$

In the rules, $op \in \{=, <, >, \leq, \geq\}$, *id* denotes the identifiers for variables, and $elem\_func \in \{\exp, \log, \sin, \cos, \ldots\}$. Formulas of the form A are called *atomic constraints*. □

Each formula *fml* represents a constraint $c \subseteq \mathbb{R}^n$ on the variables. When a variable $x_i = x.i$ does not appear in a formula, the corresponding constraint

c does not restrict the values of a variable x_i, i.e., $\pi_i(c) = \pi_i(\mathcal{D})$. Note that, to describe examples, we may use formulas in the form of $T_1 \ op \ T_2 \ op \ T_3 \equiv (T_1 \ op \ T_2) \wedge (T_2 \ op \ T_3)$ and $T_1 \in [T_2, T_3] \equiv T_2 \leq T_1 \leq T_3$.

The underlying relation $R \subseteq \mathbb{R}^n$ of a constraint *fml* corresponds to the mathematical *interpretation* of the formula *fml*. A solution $x \mapsto d$ of *fml* is determined by checking whether $d \in R$ holds. An interpretation may result in *undefined* (e.g., $x = 1/0$), in which case the constraint is represented as **false**.

Example 7 Consider an RCS $\langle x, \mathcal{D}, \mathcal{C} \rangle$, where x is a two-dimensional variable $x = (x_1, x_2)$ over the domain $\mathcal{D} = \mathbb{R}^2$. Let $c \in \mathcal{C}$ be a constraint described by the formula $(x_2 < \sin(x_1)) \wedge (0 \leq x_1 \leq 2 \cdot \pi)$. The valuation $x \mapsto (1/2, 1/2)$ satisfies c. When $\mathcal{C}$ only contains c, it is a solution of the RCS. Clearly the constraint $\mathbf{true} = \mathcal{D}$ is entailed by the constraint c, but the constraint $x_2 < 1$ is also entailed by c, but the constraint $x = (1/2, 1/2)$ is not entailed by c. If we consider another domain $\mathcal{D}' = \{(v_1, v_2) \mid v_2 < 0 \wedge 0 \leq v_1 \leq \pi\}$, then $\models_{\mathcal{D}'} c$ holds. □

3.2 Interval-based Entailment Checking

The RCSs discussed in the previous section are generally not representable on computers because we should handle infinite sets of values and valuations in a constraint. For example, real numbers are not machine-representable, and relations described by nonlinear inequalities contain infinite points and are not computable. In this section, we describe methods for computing RCSs by expressing tokens by their box enclosures.

We begin by recalling interval approximations and extensions of relations.

Remark 2 Consider a real constraint $c \in \mathcal{C}$ that ranges over $\mathbb{R}^n$. An interval approximation $apx_{\mathbb{I}^n}(c)$ of c is the smallest box in $\mathbb{I}^n$ satisfying $c \subseteq apx_{\mathbb{I}^n}(c)$. For a box $C = (I_1, \ldots, I_n) \in \mathbb{I}^n$ that is an interval extension of c, $\exists (v_1, \ldots, v_n) \in I_1 \times \cdots \times I_n \ ((v_1, \ldots, v_n) \models_{\mathcal{D}} c)$ holds. □

We can check the entailment on real constraints using the interval approximations and set operations in some cases.

Lemma 1 For real constraints c and c' that range over $\mathbb{R}^n$, the following properties hold:

$$(apx_{\mathbb{I}^n}(c) \cap apx_{\mathbb{I}^n}(c') = \emptyset) \Rightarrow ((c \not\vdash c') \wedge (c' \not\vdash c)), \tag{3.1}$$

$$(apx_{\mathbb{I}^n}(c) \cap apx_{\mathbb{I}^n}(\mathbb{R}^n \setminus c') = \emptyset) \Rightarrow (c \vdash c'). \tag{3.2}$$

□

Proof. (Equation (3.1)) Because $apx_{\mathbb{I}^n}(c)$ and c do not intersect with $apx_{\mathbb{I}^n}(c')$ nor c' from the antecedent, $c \not\vdash c'$ and $c \not\vdash c'$ hold.

(Equation (3.2)) For $rel, rel' \subseteq \mathbb{R}^n$, $(rel \subseteq rel') \Leftrightarrow (rel \cap (\mathbb{R}^n \setminus rel') = \emptyset) \Leftarrow ((apx_{\mathbb{I}^n}(rel) \cap apx_{\mathbb{I}^n}((\mathbb{R}^n \setminus rel')) = \emptyset)$ holds. Thus, $c \subseteq c'$ and $c \vdash c'$ hold. ■

3.3 Interval-based Consistency Techniques

Consistency enforcing techniques refine a constraint system into an equivalent one whose domain has the same set of solutions as before, and is *consistent*. An ideal consistent system is a solved system.

Remark 3 An RCS $\langle x, \mathcal{D}, \mathcal{C} \rangle$ is *solved* when, for every constraint $c \in \mathcal{C}$, $\models_{\mathcal{D}} c$ holds. □

In general, it is difficult to transform an RCS into a solved system without losing a solution. Moreover, the computation of solved RCSs is untractable in general because we should enumerate infinite number of valuations to prove the consistency of a domain. Accordingly, we introduce below the *locally consistent* RCSs that partially satisfy constraints, i.e., only subsets of the domain satisfy the projections of constraints.

Local consistencies for RCSs have been proposed based on interval analysis (see Reference [7] for an introduction). Methods such as Newton [80] and CLP(BNR) [9] computes *hull-* and *box-consistent* RCSs.

Definition 14 (hull- and box-consistencies [5, 8, 6]) Consider an RCS $\langle x, \mathcal{D}, \mathcal{C} \rangle$ with n real variables $x = (x_1, \ldots, x_n)$, a domain $\mathcal{D} = I_1 \times \cdots \times I_n \subseteq \mathbb{R}^n$, and a set of constraints $\mathcal{C}$. We can evaluate the the following consistencies of the RCS:

- $\langle x, \mathcal{D}, \mathcal{C} \rangle$ is *hull-consistent* [5] if and only if it satisfies the following condition:
$$(I_1, \ldots, I_n) = apx_{\mathbb{I}^n}(\{v \in \mathcal{D} \mid \forall c \in \mathcal{C}\ (v \models_{\mathcal{D}} c)\}). \tag{3.3}$$
- $\langle x, \mathcal{D}, \mathcal{C} \rangle$ is *box-consistent* [8] if and only if every I_i satisfies the following condition ($i \in \{1, \ldots, n\}$):
$$I_i = apx_{\mathbb{I}}(\{v \in I_i \mid \forall c \in \mathcal{C}\ ((I_1, \ldots, I_{i-1}, apx_{\mathbb{I}}(\{v\}), I_{i+1}, \ldots, I_n) \in C)\}). \tag{3.4}$$

□

Hull- and box-consistencies relax the validity check (i.e., $\models_{\mathcal{D}} c$) of constraints. The right-hand side of Equation 3.3 represents an interval approximation $apx_{\mathbb{I}^n}(\mathcal{D}')$, where $\mathcal{D}' \subseteq \mathcal{D}$ is a domain such that an RCS $\langle x, \mathcal{D}', \mathcal{C} \rangle$ is solved.

Lemma 2 An interval $I_i \in \mathbb{I}$ specified in Equation 3.4 satisfies the following condition for each constraint $c \in \mathcal{C}$

$$((I_1, \ldots, I_{i-1}, [\underline{I_i}, \underline{I_i}^+), I_{i+1}, \ldots, I_n) \in C) \wedge \\ ((I_1, \ldots, I_{i-1}, (\overline{I_i}^-, \overline{I_i}], I_{i+1}, \ldots, I_n) \in C).$$

This condition states that, taking the smallest half-open interval at each bound of I_i, an interval extension C of the constraint holds. □

In the following, we overview the basic framework of interval-based consistency techniques for RCSs.

Definition 15 (interval-based consistency technique) Given a constraint system $\langle x, \mathcal{D}, \mathcal{C} \rangle$, *interval-based consistency techniques* compute a constraint system $\langle x, \mathcal{D}', \mathcal{C} \rangle$, where $\mathcal{D}' \subseteq \mathcal{D}$ is hull- or box-consistent. □

3.3.1 Branch-and-Prune Framework

The generic computation in the consistency-based methods is described by the BranchAndPrune algorithm in Figure 3.1. It is the basic algorithm used in Newton and `Numerica` (called SolveConstraints in Reference [81]). Let $\mathbb{F}$ be a system of floating-point numbers, and let $\langle x, \mathcal{D}, \mathcal{C} \rangle$ be an RCS consisting of an n-dimensional variable x, a domain $\mathcal{D} = I_1 \times \cdots \times I_n$, where $(I_1, \ldots, I_n)$ forms a box in $\mathbb{IF}^n$, and a set $\mathcal{C}$ of constraints. The inputs to the algorithm are a set of constraints $\mathcal{C}$, a box $B = (I_1, \ldots, I_n)$, and a maximal box width $w_{max} \in \mathbb{F}_{>0}$. The algorithm repeatedly refines and divides the box B and computes a set BS of boxes where $\langle x, I'_1 \times \cdots \times I'_n, \mathcal{C} \rangle$ is box-consistent for each $B' = (I'_1, \ldots, I'_n) \in BS$. The width of each box B' is narrower than w_{max}. In the algorithm, the two procedures Prune (line 1) and Branch (line 6) that refine and divide a box are left open. If the result of a pruning is empty, the algorithm returns empty (line 9). When the result is not empty and it is small enough, the algorithm returns it as a result (line 4). Otherwise, the algorithm divides the box (line 6), and calls itself recursively for each box (line 7).

A pseudo-algorithm of Branch is shown in Figure 3.2. It divides an element I_i of an input box B at the midpoint of the interval, into two intervals $I_{i,1}$ and $I_{i,2}$

Input: set of constraint $\mathcal{C}$, box B, box width w_{max}
Output: set of consistent boxes $\{B_i\}_{i\in\{1,\ldots,n\}}$
1: $B := \textsc{Prune}(\mathcal{C}, B)$
2: **if** $B \neq \emptyset$ **then**
3: **if** $\mathrm{w}(B) \leq w_{max}$ **then**
4: return $\{B\}$
5: **else**
6: $(B_1, B_2) := \textsc{Branch}(B)$
7: return $\textsc{BranchAndPrune}(\mathcal{C}, B_1) \cup \textsc{BranchAndPrune}(\mathcal{C}, B_2)$
8: **else**
9: return $\emptyset$

Figure 3.1 BranchAndPrune algorithm.

Input: box $B = (I_1, \ldots, I_n)$
Output: pair of boxes (B_1, B_2)
1: $I_i := \textsc{Select}(B)$
2: $(I_{i,1}, I_{i,2}) := ([\underline{I_i}, \mathrm{m}(I_i)], [\mathrm{m}(I_i), \overline{I_i}])$
3: return $((I_1, \ldots, I_{i-1}, I_{i,1}, I_{i+1}, \ldots, I_n), (I_1, \ldots, I_{i-1}, I_{i,2}, I_{i+1}, \ldots, I_n))$

Figure 3.2 Branch algorithm.

(line 2). To select an element (line 1), we typically use the round-robin method. At line 3, a pair of boxes containing the divided intervals is returned.

The generic implementation of Prune is illustrated in Figure 3.3. The inputs to the algorithm are a set of p constraints and a box B. The algorithm returns a refined box $B' = (I'_1, \ldots, I'_n)$, where $\langle x, I'_1 \times \cdots \times I'_n, \mathcal{C}\rangle$ is consistent. The algorithm continues refining B until the fixpoint is reached. At line 4, the Revise procedure refines B with respect to an atomic formula. Two instantiations HC4Revise and BC3Revise of Revise are shown in the last part of this section. Atomic formulas are managed by the tuple $(\mathcal{C}, \mathcal{C}')$ of sets of constraints. Each time a refinement succeeds, the corresponding constraint is moved from $\mathcal{C}$ to $\mathcal{C}'$ (line 10). Because the monotonicity is not held in this process, a constraint in $\mathcal{C}'$ will be put back in $\mathcal{C}$ after another refinement succeeds (line 7).

In the following, we describe the existing two instantiations of Revise to filter input boxes by enforcing the consistencies.

Input: set of constraints $\{c_1, \ldots, c_p\}$, box B
Output: consistent box B

1: $(\mathcal{C}, \mathcal{C}') := (\{c_1, \ldots, c_p\}, \emptyset)$
2: **while** $\mathcal{C} \neq \emptyset \wedge B \neq \emptyset$ **do**
3: $c_i := \textsc{Select}(\mathcal{C})$
4: $B' := \textsc{Revise}(c_i, B)$
5: **if** $B' \neq B$ **then**
6: $\mathcal{C}'' := \{c_j \in \mathcal{C}' \mid \exists x_k \in var(c_j)\ (B.k \neq B'.k)\}$
7: $(\mathcal{C}, \mathcal{C}') := (\mathcal{C} \cup \mathcal{C}'', \mathcal{C}' \setminus \mathcal{C}'')$
8: $B := B'$
9: **else**
10: $(\mathcal{C}, \mathcal{C}') := (\mathcal{C} \setminus \{c_i\}, \mathcal{C}' \cup \{c_i\})$
11: **endwhile**
12: return B

Figure 3.3 Prune algorithm.

Input: constraint $c(t_1, \ldots, t_q)$, box B
Output: hull-consistent box B

1: **for** $i \in \{1, \ldots, q\}$ **do**
2: $B := \textsc{ForwardEval}(t_i, B)$
3: $B := \textsc{BackwardPropag}(c(t_1, \ldots, t_q), B)$
4: return B

Figure 3.4 HC4Revise algorithm.

3.3.2 Hull-Consistency-based Revise Algorithm

The first instantiation HC4Revise [6] (Figure 3.4) provides a method for computing hull-consistent RCSs. The computation is based on the *attribute tree* of the underlying formula of the input constraint. We take the leaf nodes $t_1, \ldots, t_q$ of the tree, which represent the constants and variables in the formula. Other nodes in the tree represent the unary or binary operations on real values defined in Definition 13. At line 2, the algorithm evaluates the possible domain of the formula by traversing the tree from the bottom-up. The algorithm then propagates the refined domain to the leaves from the top down (line 3).

Input: constraint $c(x_1, \ldots, x_n)$, box $(I_1, \ldots, I_n)$
Output: box-consistent box B
1: **for** $i \in \{1, \ldots, n\}$ **do**
2: $C' := C(I_1, \ldots, I_{i-1}, \bullet, I_{i+1}, \ldots, I_n)$
3: $I_i := \text{BC3REVISEL}(C', I_i) \uplus \text{BC3REVISEU}(C', I_i)$
4: return $(I_1, \ldots, I_n)$

Figure 3.5 BC3REVISE algorithm.

3.3.3 Box-Consistency-based REVISE Algorithms

The second instantiation of REVISE consists of the algorithms called BC3REVISE, BC3REVISEL and BC3REVISEU (Figures 3.5 and 3.6). These algorithms are a modification of the algorithms described in Reference [6], which generalizes the variants such as BOXPRUNE in Newton and `Numerica`. The output of BC3REVISE is box-consistent boxes. At line 2, the BC3REVISE algorithm computes an interval extension C of the input constraint c, and then computes the i-th projection of C. The algorithm then applies the sub-filters BC3REVISEL and BC3REVISEU, which revise the lower and upper interval bounds for every component (indicated by $\bullet$) of the input box (line 3).

Figure 3.6 illustrates the BC3REVISEL algorithm. At line 8, the algorithm checks whether a solution of the (uni-variate) constraint is contained in the input interval I, and returns the empty interval if the check fails. For example, consider a constraint of the form of $f(x) = 0$, where f is a function $\mathbb{R} \to \mathbb{R}$. This check can be done by checking whether $F(I) \ni 0$ holds. Next, the algorithm checks whether the interval I is the smallest canonical interval with respect to the system of floating-point numbers (line 3) and returns I if the check succeeds. This check corresponds to the definition of box-consistency.

When both checks are failed, the algorithm optionally applies the NARROW_C operator to reduce the interval I with respect to the constraint C (line 6). For example, the interval Newton operator is used as the operator [80]. At lines 7–12, the algorithm splits I in two, and applies itself to each result.

3.3.4 Implementation of Branch-and-Prune

The consistency-based technique just described has been implemented in various systems such as `Numerica` [81], `ILOG Solver`, and `RealPaver` [35, 34].

Here, we introduce a state-of-the-art implementation called `Elisa` [36], which consists of about 25,000 lines of C++ code, is built as a library, and is available as

Input: constraint C, interval I
Output: interval I

```
1:  if I ∈ C then
2:    return ∅
3:  if Ī = I̲⁺ then
4:    return I
5:  else
6:    I := NARROW_C(I)
7:    (I_l, I_u) := ([I̲, m(I)], [m(I), Ī])
8:    I_l := BC3REVISEL(C, I_l)
9:    if I_l ≠ ∅ then
10:     return I_l
11:   else
12:     return BC3REVISEL(C, I_u)
```

Figure 3.6 BC3ReviseL algorithm.

open-source software. The main classes of `Elisa` are illustrated in Table 3.1. The upper half of the table shows the constructs for representing data. `Elisa` supports real numbers represented as intervals, integers, and floating-point numbers as the domain of constraints. Each constraint is represented by the derived class of `RealConstraint`. The bottom half of the table shows the classes that implement the algorithms introduced in this section. We can extend the solver by overriding the appropriate classes.

3.3.5 Examples

We solved two examples, which are taken from the `RealPaver` package [34], using the `Elisa` framework. As the Revise algorithms, we used HC4Revise, BC3Revise, and BC5Revise implemented in `Elisa`. BC5Revise (a successor of BC4Revise [6]) is an algorithm that combines the HC4Revise and BC3Revise algorithms. The maximal box width w_{max} input to BranchAndPrune was set to 10^{-12}. Other parameters were not changed. The computational results are illustrated in Table 3.2. Each column represents:

- The number of resulting boxes.
- The number of calls to Branch.
- The number of calls to Revise (calls to HC4Revise and BC3Revise are shown separately in the lower rows).
- The CPU clock cycles for the computation.

Table 3.1 Main classes in `Elisa`.

object type	class
values	`Interval`, `IntSet`
constaints	`IntervalConstant`, `IntConstant`, `RealConstant`
variables	`RealVariable`, `IntVariable`
constraints	`RealConstraint`
models	`Model`
algorithm	**class**
BranchAndPrune	`IntervalSolver`
Branch	`NeighborhoodFunction`
Prune	`FixedPointReduction`
Revise	`Reduction`
HC4Revise	`HullReduction`
BC3Revise	`BoxReduction`
Select	`SplitChooser`, `MoveStrategy`

Example 8 The first example is a simple problem to compute the positive seventh root of 12 (shown in Table 3.2 as "abc"). The constraint is described with a real variable x as

$$x^7 = 12.$$

The initial domain was set to $[0, 10^8]$. We could compute a precise interval by a single application of HC4Revise, whereas we should apply BC3Revise for 26 times while applying Branch for 12 times. BC5Revise computed in the same way as HC4Revise because BC5Revise first applied HC4Revise when a variable occurred only once in constraints. □

Example 9 The second and third examples model chemical equilibrium systems (shown in Table 3.2 as "c1" and "c2"). In the first model, we used a three-dimensional variable (x_1, x_2, x_3), and described the following constraints

$$\begin{aligned} 14 \cdot x_1^2 + 6 \cdot x_1 \cdot x_2 + 5 \cdot x_1 - 72 \cdot x_2^2 - 18 \cdot x_2 &= 850 \cdot x_3 - 2.0 \cdot 10^{-9}, \\ 0.5 \cdot x_1 \cdot x_2^2 + 0.01 \cdot x_1 \cdot x_2 + 0.13 \cdot x_2^2 + 0.04 \cdot x_2 &= 4.0 \cdot 10^4, \\ 0.03 \cdot x_1 \cdot x_3 + 0.04 \cdot x_3 &= 850. \end{aligned}$$

The initial domain was set to $[-10^3, 10^3]^3$.

Table 3.2 Computation results by `Elisa`.

	HC4Revise				BC3Revise			
prob.	boxes	br.	revise	cycles	boxes	br.	revise	cycles
abc	1	0	1	$120{\cdot}10^3$	1	12	26	$440{\cdot}10^3$
c1	2	1	476	$3074{\cdot}10^3$	13	8289	80785	$617{\cdot}10^6$
c2	–	–	–	–	–	–	–	–
	BC5Revise							
abc	1	0	1/0	$127{\cdot}10^3$				
c1	2	1	41/101	$10.9{\cdot}10^6$				
c2	8	70	10737/27871	$1135{\cdot}10^6$				

In the second model, we used a five-dimensional variable $(x_1, x_2, x_3, x_4, x_5)$, and the following constraints

$$\begin{aligned}
x_1 \cdot (x_2 + 1) &= 3 \cdot x_5, \\
x_3 \cdot (x_2 \cdot (2 \cdot x_3 + a_7) + 2 \cdot a_5 \cdot x_3 + a_6) &= 8 \cdot x_5, \\
x_4 \cdot (a_9 \cdot x_2 + 2 \cdot x_4) &= 4 \cdot a \cdot x_5, \\
x_1 + x_2 \cdot (2 \cdot x_1 + x_3 \cdot (x_3 + a_7) + a_8 + 2 \cdot a_{10} \cdot x_2 + a_9 \cdot x_4) &= a \cdot x_5, \\
x_1 + x_3 \cdot (a_5 \cdot x_3 + a_6) + x_4^2 + & \\
x_2 \cdot (x_1 + a_{10} \cdot x_2 + x_3 \cdot (x_3 + a_7) + a_8 + a_9 \cdot x_4) &= 1,
\end{aligned}$$

where $a = 10$, $a_5 = 0.193$, $a_6 = 0.002597/\sqrt{40}$, $a_7 = 0.003448/\sqrt{40}$, $a_8 = 0.00001799/40$, $a_9 = 0.0002155/\sqrt{40}$, $a_{10} = 0.00003846/40$. The initial domain was set to $[0, 10^8]^5$.

In Table 3.2, we can see that BC5Revise solved both the models most efficiently by applying both HC4Revise and BC3Revise. Computation to solve the second model using only HC4Revise or BC3Revise was failed because of a heap overflow. □

Chapter 4

Hybrid Systems

Hybrid systems are systems consisting of discrete changes and continuous changes over time. See References [73, 54] for an introduction. In an execution of a hybrid system, each phase of continuous changes is governed by a (continuous) dynamics (or vector fields on manifolds), and the behavior of each phase is formed as a continuous function over time. A discrete change, which is also referred to as a (discrete) event, corresponds to a switching between different dynamics.

Example 10 An example of a hybrid system is a particle that bounces off a sinusoidal ground surface. The position and velocity of the particle in the two-dimensional space are described by four real functions over time $p_x, p_y, v_x, v_y : \mathbb{R}_{\geq 0} \to \mathbb{R}$. A possible particle behavior is illustrated in Figure 4.1. The dynamics of the particle is described by the IVP-ODE

$$\begin{aligned}(\dot{p_x}(\tau), \dot{p_y}(\tau), \dot{v_x}(\tau), \dot{v_y}(\tau)) &= (v_x(\tau), v_y(\tau), 0, -g + k \cdot v_y^2(\tau)) \\ \wedge\, (p_x(t_0), p_y(t_0), v_x(t_0), v_y(t_0)) &= (p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}),\end{aligned} \tag{4.1}$$

where $g = 9.8$ and $k = 10^{-3}$ are constants representing the acceleration due to gravity and the air resistance, respectively, $t_0 \in \mathbb{R}_{\geq 0}$ is an initial time (e.g., $t_0 = 0$), and $(p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}) \in \mathbb{R}^4$ is an initial state (e.g., $(p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}) = (0, 1.1, 4.1, -0.4)$). We assume that a contact of the particle with the ground surface at a given time t is detected by

$$\sin(2 \cdot p_x(t)) - p_y(t) = 0. \tag{4.2}$$

The above formula holds when the particle contacts the surface at time t, and the velocity of the particle is changed from $(v_x(t_-), v_y(t_-)) = \lim_{\tau \uparrow t}(v_x(\tau), v_y(\tau))$ to $(v_x(t), v_y(t))$ by the force of repulsion according to

$$(v_x(t), v_y(t)) = (v_x(t_-), v_y(t_-)) - (e+1) \cdot n \cdot dot(n, (v_x(t_-), v_y(t_-))), \tag{4.3}$$

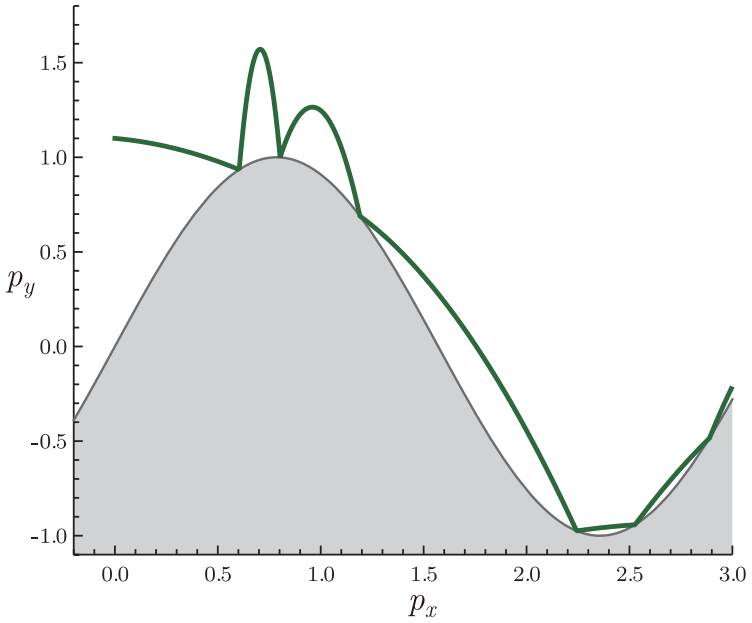

Figure 4.1 Behavior of bouncing particle.

where $e = 0.3$ is the coefficient of restitution, $dot(\cdot, \cdot)$ is the dot product operator, and n is the normal vector of the surface and is given as $n = m/\|m\|$, where m is a real vector $(-\cos(p_x(t_-)), 1)$. □

There have been various modeling frameworks for hybrid systems that originate from several research communities including computer science, control, and biology. Various formalizations of the frameworks have been developed [73, 42]. Relations between different frameworks (e.g., translation [18] and bisimulation [63, 72]), and notions such as hybrid trajectories are important in the modeling, simulation, and verification of hybrid systems.

To develop methods for the simulation and verification of hybrid systems, we should determine the class of arithmetic expressions (e.g., rectangular, affine, or nonlinear) that can be described within models. For example, many of recent model checkers restrict equations to linear equations [76, 12]. Users of these tools have to translate a nonlinear problem into a piecewise linear model by hand [44]. Many of the verification problems on nonlinear models still remain unsolved. We should also consider whether expressions in models allow us to express uncertainties (e.g., interval-valued coefficients). In our experiments with existing frameworks, we have encountered some difficulties in the simulation of models involving uncertainties (see Section 4.3). We attribute the difficulties to

the formalization of discrete changes by the framework. Another variation of classes is identified by whether discrete changes are given externally or occur internally through interactions between continuous changes and boundaries in the state space. In this study, we only consider the latter.

We employ two modeling frameworks, *hybrid automata* (HA), and a *hybrid concurrent constraint programming* language called *Tiny HCC*. We first introduce simple structures called *real-time transition systems* (RTTSs) and *hybrid trajectories* to illustrate the behavior of hybrid systems (Section 4.1). Next, in Sections 4.2 and 4.3, we describe HA and Tiny HCC. We explain the semantics of HA and Tiny HCC by translating them into RTTSs. We also explain notions of *execution* (i.e., hybrid trajectories) and *reachability* that correspond to the models.

4.1 Real-Time Transition Systems and Hybrid Trajectories

Real-time transition systems (RTTSs) are transition systems in which transitions are labeled by positive real numbers that represent the duration of the transitions.

Definition 16 (real-time transition system [18]) A *real-time transition system* (RTTS) is a triple $\langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$, where $\mathcal{S}$ is a state space, $\mathcal{T} \subseteq \mathcal{S} \times \mathbb{R}_{\geq 0} \times \mathcal{S}$ is the time transition relation, and $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states. A transition $(s, t, s') \in \mathcal{T}$ is also denoted by $s \xrightarrow{t} s'$. The relation satisfies the following conditions:

- Every transition $s \xrightarrow{t} s'$ with $t > 0$ can be split into transitions $s \xrightarrow{t'} s''$ and $s'' \xrightarrow{t''} s'$, where $t = t' + t''$ and $t', t'' > 0$.
- For every two transitions $s \xrightarrow{t} s'$ and $s' \xrightarrow{t'} s''$ with $t, t' > 0$, there exists a transition $s \xrightarrow{t+t'} s''$. □

In an RTTS, we describe a discrete change from a state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ by $s \xrightarrow{0} s'$, and we describe a continuous change from s to s' with the duration of $t \in \mathbb{R}_{>0}$ by $s \xrightarrow{t} s'$.

Example 11 To model the bouncing particle in Example 10, we can construct an RTTS $\langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$, where

- $\mathcal{S} = \mathbb{R}^4$,
- $\mathcal{S}_0 = [0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4]$ (we assume that $\sin(2 \cdot p_{x,0}) - p_{y,0} \geq 0$ holds for every $(p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}) \in \mathcal{S}_0$),

- a continuous change from a state $(p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}) \in \mathcal{S}$ at time $t_0 \in \mathbb{R}_{\geq 0}$ with the duration $t \in \mathbb{R}_{>0}$ is specified as a transition in $\mathcal{T}$

$$((p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}),\ t,\ \lim_{\tau \uparrow (t_0+t)} (p_x(\tau), p_y(\tau), v_x(\tau), v_y(\tau))),$$

  where trajectories $p_x, p_y, v_x, v_y : [t_0, t_0+t) \to \mathbb{R}$ are described by IVP-ODE (4.1), and, for every $t'' \in (t, t+t')$, $\sin(2 \cdot p_x(t'')) - p_y(t'') = 0$ (Equation (4.2)) does not hold, and
- a discrete change at time $t \in \mathbb{R}_{>0}$ is described as a transition in $\mathcal{T}$

$$((p_x(t_-), p_y(t_-), v_x(t_-), v_y(t_-)),\ 0,\ (p_x(t), p_y(t), v_x(t), v_y(t))),$$

  where $(p_x(t_-), p_y(t_-), v_x(t_-), v_y(t_-))$ denotes $\lim_{\tau \uparrow t} (p_x(\tau), p_y(\tau), v_x(\tau), v_y(\tau))$, $\sin(2 \cdot p_x(t_-)) - p_y(t_-) = 0$ holds, $p_x(t) = p_x(t_-)$, $p_y(t) = p_y(t_-)$, and $v_x(t)$ and $v_y(t)$ are computed from $v_x(t_-)$ and $v_y(t_-)$ by Equation (4.3). □

Executions or *behaviors* of RTTSs are described as *hybrid trajectories*, which are continuous-state-valued functions over a *hybrid time set*.

Definition 17 (hybrid trajectory) A k-step *hybrid trajectory* ($k \in \mathbb{N}$) is a triple $\langle \mathcal{S}, \boldsymbol{T}, \Phi \rangle$ consisting of:

- A state space $\mathcal{S}$.
- A *hybrid time set* $\boldsymbol{T}$ that is a (possibly infinite) indexed family of left-closed, right-open time intervals $\{I^i\}_{i \in \{1,\ldots,k\}}$, where
  - $\mathrm{w}(I^i) > 0$,
  - $\underline{I^0} \geq 0$, and
  - $\overline{I^i} = \underline{I^{i+1}}$.
- A family Φ of functions $\{\phi^i\}_{i \in \{1,\ldots,k\}}$, where $\phi^i : I^i \to \mathcal{S}$. □

An execution of an RTTS is formalized as a hybrid trajectory.

Definition 18 (execution of RTTS) A k-step *execution* of an RTTS $\langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$ is a hybrid trajectory $\langle \mathcal{S}', \boldsymbol{T}, \Phi \rangle$ that satisfies the following:

- $\mathcal{S} = \mathcal{S}'$.
- $\boldsymbol{T}$ is formed as below:
  1. Consider a sequence of transitions in $\mathcal{T}$

$$s^0 \xrightarrow{t^1} s^1_- \xrightarrow{0} s^1 \xrightarrow{t^2} \cdots \xrightarrow{t^k} s^k_- \xrightarrow{0} s^k,$$

  where $s_0 \in \mathcal{S}_0$, $s^1, \ldots, s^k \in \mathcal{S}$, $s^1_-, \ldots, s^k_- \in \mathcal{S}$, and $t^1, \ldots, t^k \in \mathbb{R}_{>0}$.

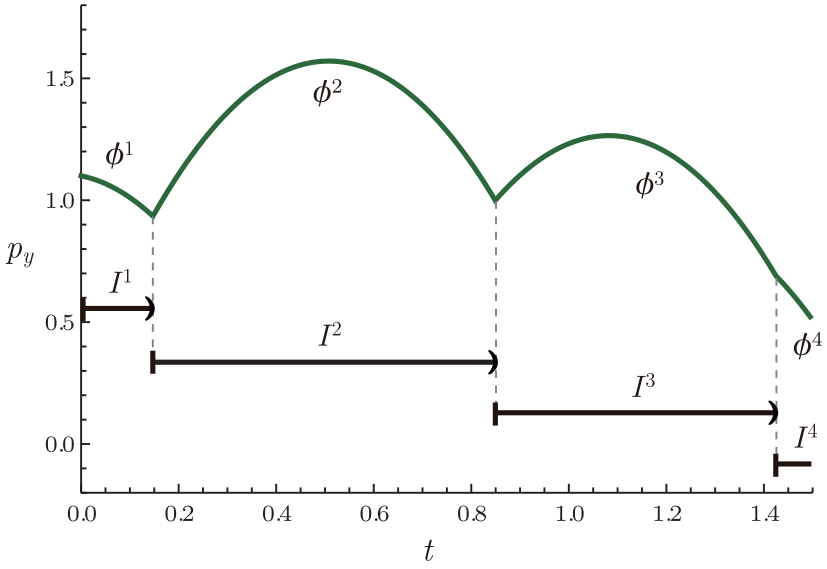

Figure 4.2 Hybrid trajectory of bouncing particle.

2. Construct the set $\boldsymbol{T}$ as

$$\{[0, t^1)^1, [t^1, t^1 + t^2)^2, \ldots, [\sum_{i=1}^{k-1} t^i, \sum_{i=1}^{k} t^i)^k\}.$$

- For $i \in \{1, \ldots, k\}$, $\phi^i \in \Phi$ is defined as the function $I^i \to \mathcal{S}$, where $\phi^i(t) = s$ such that $(s^{i-1}, t - \underline{I^i}, s) \in \mathcal{T}$. □

Example 12 We sketch an execution of the RTTS in Example 11 by assuming the initial state of the particle is $(p_{x,0}, p_{x,0}, v_{x,0}, v_{y,0}) = (0, 1.1, 4.1, -0.4)$ at time $t^0 = 0$. The hybrid trajectory of the particle is illustrated in Figures 4.1 and 4.2. The continuous trajectory ϕ^1 of the particle over the time interval I^1 is determined by IVP-ODE (4.1). The particle contacts the ground at time $t^1 = \overline{I_1} = \underline{I_2} = 0.147\cdots$ because the position $(p_x(t^1_-), p_y(t^1_-)) = \lim_{\tau \uparrow t^1}(p_x(\tau), p_y(\tau)) = (0.603\cdots, 0.934\cdots)$ of the particle satisfies Equation (4.2) to detect a contact. The contact of the particle with the ground changes the velocity from $(v_x(t^1_-), v_y(t^1_-))$ to $(v_x(t^1), v_y(t^1))$ subject to Equation (4.3). After the contact at time t_1, the particle again flies in the air following IVP-ODE (4.1) with the initial state $(p_x(t^1_-), p_y(t^1_-), v_x(t^1), v_y(t^1))$. □

4.2 Hybrid Automata

Hybrid automata [1, 42] are a mathematical model of hybrid systems. The discrete aspect of a hybrid system is described by a (finite) automaton, where each state corresponds to a phase of continuous change and each transition corresponds to a discrete change. Continuous dynamics is described by an ODE indexed by states of the automaton.

Definition 19 (hybrid automaton) A *hybrid automaton* (HA) is a tuple $\langle \mathcal{Q}, \mathbb{R}^n, \mathcal{E}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{R}, \mathit{Init} \rangle$ that consists of the following components:

- A finite set $\mathcal{Q}$ of *discrete states*.
- A set $\mathbb{R}^n$ of *continuous states*.
- A finite set $\mathcal{E} \subseteq \mathcal{Q} \times \mathcal{Q}$ of *discrete state transitions*.
- A family $\mathcal{F} = \{f_q\}_{q \in \mathcal{Q}}$ of *vector fields* $f_q : \mathbb{R}^n \to \mathbb{R}^n$. We assume that f_q is a Lipschitz continuous function.
- A family $\mathcal{I} = \{\mathit{Inv}_q(x)\}_{q \in \mathcal{Q}}$ of *invariants*, where x is a variable over $\mathbb{R}^n$, and $\mathit{Inv}_q \subseteq \mathbb{R}^n$.
- A family $\mathcal{G} = \{\mathit{grd}_e(x) = 0\}_{e \in \mathcal{E}}$ of *guard constraints*, where x is a variable over $\mathbb{R}^n$, and grd_e is a function $\mathbb{R}^n \to \mathbb{R}$.
- A family $\mathcal{R} = \{\mathit{rst}_e\}_{e \in \mathcal{E}}$ of *reset functions* $\mathit{rst}_e : \mathbb{R}^n \to \mathbb{R}^n$.
- A set of *initial states* $\mathit{Init} = (q_0, X_0)$, where $q_0 \in \mathcal{Q}$, and $X_0 \subseteq \mathbb{R}^n$. □

Example 13 Figure 4.3 illustrates an HA $\langle \mathcal{Q}, \mathbb{R}^4, \mathcal{E}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{R}, \mathit{Init} \rangle$ that models the bouncing particle in Example 10. Each component of the HA is specified as follows:

$$\mathcal{Q} = \{\mathit{falling}\}, \quad \mathcal{E} = \{\mathit{bounce} \equiv (\mathit{falling}, \mathit{falling})\},$$
$$\mathcal{F} = \{f_{\mathit{falling}}(p_x, p_y, v_x, v_y) = (v_x, v_y, 0, -g + k \cdot v_y^2)\},$$
$$\mathcal{I} = \{\mathit{Inv}_{\mathit{falling}}(p_x, p_y, v_x, v_y) \equiv (\sin(2 \cdot p_x) - p_y \geq 0)\},$$
$$\mathcal{G} = \{\mathit{grd}_{\mathit{bounce}}(p_x, p_y, v_x, v_y) = \sin(2 \cdot p_x) - p_y = 0\},$$
$$\mathcal{R} = \{\mathit{rst}_{\mathit{bounce}}(p_x, p_y, v_x, v_y) = (p_x, p_y,$$
$$\frac{(1 - e \cdot 4 \cdot \cos(2 \cdot p_x)^2) \cdot v_x + (1 + e) \cdot 2 \cdot \cos(2 \cdot p_x) \cdot v_y}{1 + 4 \cdot \cos(2 \cdot p_x)^2},$$
$$\frac{(1 + e) \cdot 2 \cdot \cos(2 \cdot p_x) \cdot v_x + (-e + 4 \cdot \cos(2 \cdot p_x)^2) \cdot v_y}{1 + 4 \cdot \cos(2 \cdot p_x)^2})\},$$
$$\mathit{Init} = (\mathit{falling}, [0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4]),$$

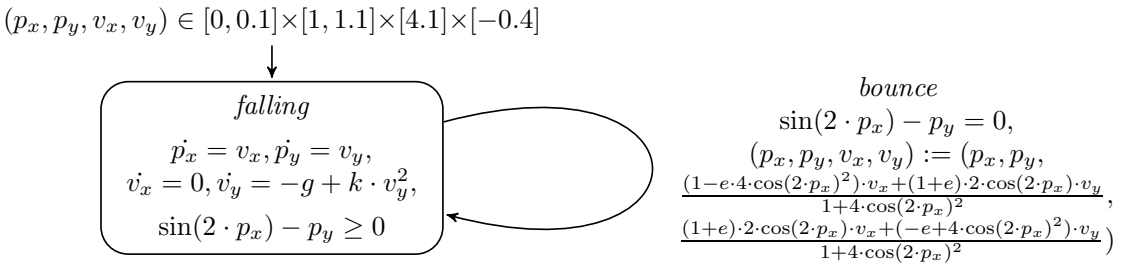

Figure 4.3 Model of bouncing particle in HA.

where $g = 9.8$ and $k = 10^{-3}$ as in Example 10. In Figure 4.3, the discrete state *falling* and the discrete state transition *bounce* are illustrated as the node and edge of the graph, respectively. □

4.2.1 Operational Semantics of HA

We describe the operational semantics of HAs by presenting a translation scheme from HA into RTTSs. The formal semantics of HA is also found in Reference [42].

Definition 20 (translation from HA into RTTS) An HA $\langle \mathcal{Q}, \mathbb{R}^n, \mathcal{E}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{R}, \mathit{Init} \rangle$ is translated into an RTTS $\langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$ as follows. We represent a state in $\mathcal{S}$ by a pair $\langle q, v \rangle$, where $q \in \mathcal{Q}$ is a discrete state and $v \in \mathbb{R}^n$ is a continuous state at an instant of the execution. An initial state in $\langle q^0, v^0 \rangle \in \mathcal{S}_0$ satisfies $(q^0, X_0) = \mathit{Init}$ and $v^0 \in X_0$. Transitions in $\mathcal{T}$ are defined by the rules in Figure 4.4.

- Rule (4.4) describes a continuous evolution from a state $\langle q, v \rangle$ to another state $\langle q, \phi(t) \rangle$ with the duration $t \in \mathbb{R}_{>0}$. In the first premise, we obtain a continuous trajectory ϕ that satisfies an IVP-ODE given by a discrete state q in the HA. The fourth premise checks the invariant condition of q over the internal of the duration $(0, t)$.
- Rule (4.5) describes that any state $\langle q, v \rangle$ can stay within the discrete state q when the invariant condition of q holds.
- Rule (4.6) represents discrete changes corresponding to $(q, q') \in \mathcal{E}$. The guard constraint $\mathit{grd}_{(q,q')}(v) = 0$ should hold before a continuous state v jumps to another state $\mathit{rst}_{(q,q')}(v)$, where $\mathit{rst}_{(q,q')}$ is the reset function. The third premise checks the invariant condition in the destination. □

$$\frac{\begin{array}{c}\phi \models (\dot{y}(\tau) = f_q(y(\tau)) \wedge y(0) = v), \\ t \in \mathbb{R}_{>0}, \quad dom(\phi) = [0,t], \quad \forall t' \in (0,t)\ (\phi(t') \in Inv_q)\end{array}}{\langle q, v\rangle \xrightarrow{t} \langle q, \phi(t)\rangle} \tag{4.4}$$

$$\frac{v \in Inv_q}{\langle q, v\rangle \xrightarrow{0} \langle q, v\rangle} \tag{4.5}$$

$$\frac{(q,q') \in \mathcal{E}, \quad grd_{(q,q')}(v) = 0, \quad rst_{(q,q')}(v) \in Inv_{q'}}{\langle q, v\rangle \xrightarrow{0} \langle q', rst_{(q,q')}(v)\rangle} \tag{4.6}$$

Figure 4.4 Operational semantics of HA.

Note that Rule (4.5) describes states that do not cause a discrete change with respect to Rule (4.6) even if the premises hold.

Once an HA is translated into the associated RTTS, the executions of HA are easily described by hybrid trajectories.

Lemma 3 (execution of HA) A k-step *execution* of an HA forms a hybrid trajectory $\langle \mathcal{Q} \times \mathbb{R}^n, \boldsymbol{T}, \Phi\rangle$ that satisfies the following conditions ($i \in \{1, \ldots, k\}$):

- $\langle q^0, v^0\rangle$ is implied by $(q^0, X') = Init$ and $v^0 \in X'$.
- $(q^i, q^{i+1}) \in \mathcal{E}$.
- $grd_{(q^i,q^{i+1})}(\phi^i(\overline{I^i})) = 0$.
- $\phi^{i+1}(\underline{I^{i+1}}) = rst_{(q^i,q^{i+1})}(\phi^i(\overline{I^i}))$.
- ϕ^i is a solution of ODE: $\dot{y}(\tau) = f_{q^i}(y(\tau))$.
- For all $t \in I^i$, $\phi^i(t) \in Inv_{q^i}$ holds. □

Proof. Straightforward from Definition 20. ■

Example 14 Figures 4.1 and 4.2 illustrate an execution of the bouncing particle in Example 13. □

An execution of an HA is sketched by a *trace*.

Definition 21 (trace of HA) Consider a k-step execution of an HA of the form

$$\langle q^0, x^0\rangle \xrightarrow{t^1} \langle q^0, x^1_-\rangle \xrightarrow{0} \langle q^1, x^1\rangle \xrightarrow{t^2} \cdots \xrightarrow{t^k} \langle q^{k-1}, x^k_-\rangle \xrightarrow{0} \langle q^k, x^k\rangle,$$

where $q^0, q^i \in \mathcal{Q}$, $x^0, x^i, x^i_- \in \mathbb{R}^n$, and $t^i \in \mathbb{R}_{>0}$ ($i \in \{1, \ldots, k\}$). A k-step *trace* of an HA with respect to the above execution is an indexed family $\{q^i\}_{i\in\{0,\ldots,k\}}$ of discrete states in $\mathcal{Q}$. □

4.2.2 HA with Unsafe Regions

We extend the HA to indicate that some discrete states in $\mathcal{Q}$ are unsafe. Next, we consider a reachability problem to decide whether the executions of a hybrid system may reach (or will never reach) the states.

Definition 22 (HA with unsafe regions) We extend HA following Definition 19 to *HA with unsafe regions* $\langle \mathcal{Q}, \mathbb{R}^n, \mathcal{E}, \mathcal{F}, \mathit{Init}, \mathcal{I}, \mathcal{G}, \mathcal{R}, \mathit{US}\rangle$, where $\mathit{US} \subseteq \mathcal{Q}$ is a finite set of *unsafe states* to falsify the safety property of the model. □

Example 15 Consider a controller that steers a car along a straight road near a canal [15]. Figure 4.5 shows the controller modeled as an HA. The model consists of 7 discrete states corresponding to each node, three-dimensional continuous states $(p, \gamma, c) \in \mathbb{R}^3$, and 10 discrete state transitions corresponding to each edge. Let p, γ, and c represent the horizontal position of the car, the heading angle, and the internal timer, respectively. A discrete state labeled `go_ahead` has a vector field $(-r \cdot \sin(\gamma), 0, 0)$ and an invariant constraint $(p, \gamma, c) \in [-1, 1] \times (-\infty, +\infty) \times (-\infty, +\infty)$. The set of initial states is $(\texttt{go\_ahead}, [-1, 1] \times [-\pi/4, \pi/4] \times [0])$. A transition e from `go_ahead` to `left_border` has a guard constraint $\mathit{grd}_e(x) = p+1 = 0$ and a reset function $\mathit{rst}_e(x) = (p, \gamma, 0)$ (x is a variable over $\mathbb{R}^3$). An edge entering `go_ahead` represents the initial constraint. In an execution, reaching the state labeled `in_canal` signals the unsafety of the model. We have a redundant loop on the `in_canal` node to conform to the method described in Chapter 6. Figure 4.6 illustrates possible executions of the model. □

4.2.3 Reachability

We consider a reachability problem that decides whether the execution of a hybrid system may reach (or never reach) a state that is marked unsafe.

Definition 23 (reachability and unsafety) For a hybrid system, a state $q \in \mathcal{Q}$ is *reachable* within k steps if and only if there exists a k-step execution, where $\exists i \in \{0, \ldots, k\}$ $(q_i = q)$. A hybrid system is *unsafe* (respectively *safe*) within k steps if and only if there exists (respectively does not exist) a state $us \in \mathit{US}$ reachable within k steps. □

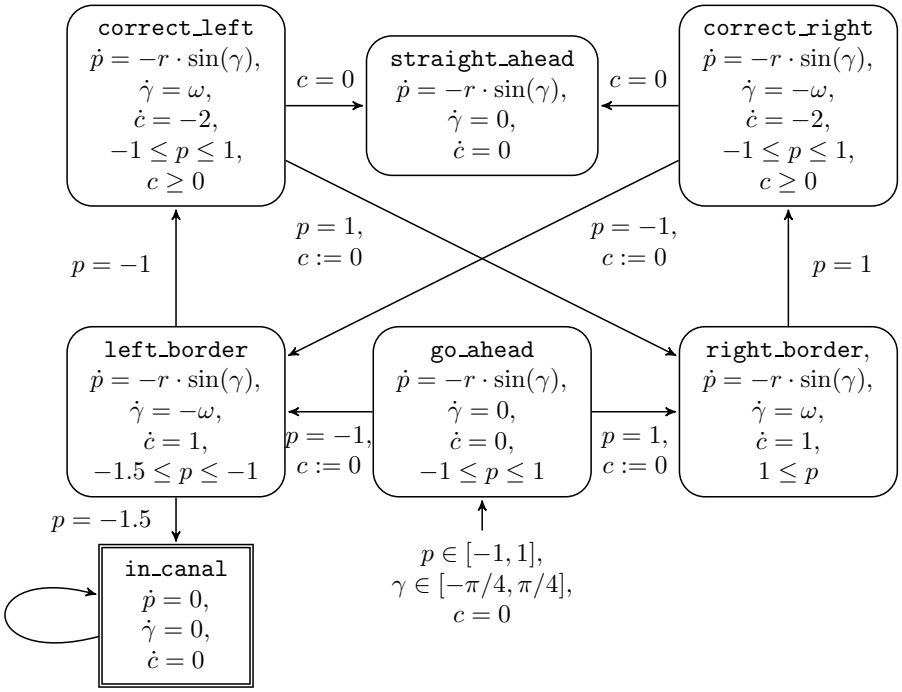

Figure 4.5 Model of car steering problem in HA.

4.3 Hybrid Concurrent Constraint Programming

Gupta et al. [40, 39] proposed a programming framework for hybrid systems called *hybrid concurrent constraint programming* (HCC). HCC languages describe hybrid systems in a declarative style using constraints (i.e., partial information). Computation in HCC is performed by (concurrent) processes that communicate with each other in terms of constraints intermediated by a *constraint store*. Computation proceeds by accumulating constraints into the store, and by checking whether constraints are entailed by the constraints in the store. A description in HCC can be regarded as a logic formula involving arithmetic constraints on continuous states that evolve over time. Since HCC is designed as a simple computational model with generic constraint systems, it would be a basis for various high-level features. For example, constructs for describing *default constraints* are proposed [40] to model complex systems.

In this thesis, we consider the *Tiny HCC* language, a small subset of the full HCC constructs, to develop a basis of a reliable implementation of constraint-based languages. In this section, we present the syntax of Tiny HCC in Section

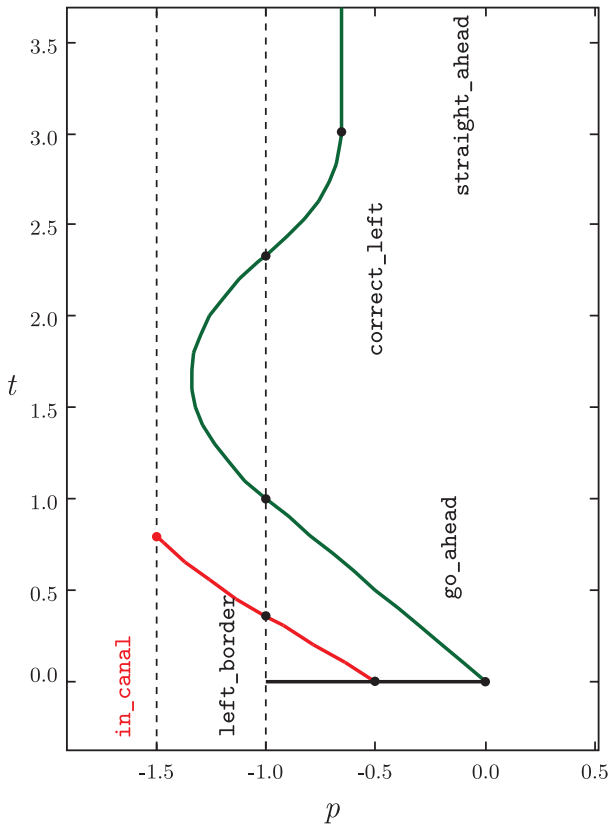

Figure 4.6 Executions of car steering problem.

4.3.1, describe the operational semantics of the language in Section 4.3.2, and illustrate an execution of a Tiny HCC program in Section 4.3.3.

Example 16 Below is a model of the saw tooth wave described in Tiny HCC.

```
1: (x, l) ∈ [0, 1]×[0.9, 1],
2: hence {
3:     l̇ = 0, cont(l),
4:     if x < l then ẋ = 1,
5:     if x_ = l then x = 0
6: }
```

The variables x and l represent the saw tooth function, and the upper limit of the value of x, respectively. At line 1, we state the initial domain of the variables, which bounds the value of the variables at time $t = 0$. The `hence` P construct

activates the inner sub-program P over the time line for every $t > 0$. At line 3, we describe constraints to force the value of l unchanged over time (i.e., enforcing the derivative of l to 0), and to keep the value of l unchanged even when the value of x causes a discrete change. These constraints are added in the constraint store at every instant in an execution. Constructs of the form `if` C `then` P at lines 4–5 check whether the constraint C is entailed by the constraint store, and when C is entailed, interpret the sub-process P. □

4.3.1 The Tiny HCC Language

The following is the syntax of Tiny HCC:

(process)	$P ::= C \mid \texttt{if}\ C\ \texttt{then}\ P \mid P, P \mid \texttt{hence}\ P \mid \epsilon$
(constraint)	$C ::= A \mid \mathit{cont}(V)$
(expression)	$A ::=$ an arithmetic formula (Definition 13) involving V
(variable)	$V ::= \mathit{id} \mid \dot{\mathit{id}} \mid \mathit{id}\_$

A Tiny HCC *process* P represents one of the following processes:

- A process of the form C, called a *tell* process, adds the constraint C to a constraint store. A constraint store is a conjunction of all the constraints given by the tell processes. All variables in an expression A range over real-valued functions over time $\mathbb{R}_{\geq 0} \to \mathbb{R}$. A variable is represented by an identifier id such as x (i.e., an abbreviation for $x(\tau)$). For a temporal description, we express the derivative $dx(\tau)/d\tau$ by $\dot{x}$, and the left-hand limit $\lim_{t \uparrow \tau} x(t)$ of $x(\tau)$ by $x\_$. The constraint C held by the tell process is one of the following kinds:
  - *instantaneous constraints* of the form A not involving variables of the form $\dot{\mathit{id}}$;
  - *continuous constraints* of the form A involving only pure identifiers and $\dot{\mathit{id}}$ as variables; or
  - constraints of the form $\mathit{cont}(V)$ for propagating the value of $x\_(t)$ into the variable $x(t)$ in an execution (this constraint was introduced in the existing implementation of HCC [13]).
- An *ask* process, `if` C `then` P, contains a constraint C, called a *guard constraint*, of the form A involving only pure identifiers, or only $\mathit{id}\_$ as variables (an instance is denoted hereafter by c_g). When the entailment relation between the constraint store and the guard constraint c_g held by an ask process is changed, a discrete change is triggered. In this study, we consider the ask processes `if` c_g `then` P formed as follows:

  - c_g contains only variables of the form $id\_$, and P contains only instantaneous constraints; or
  - c_g contains only variables described by pure identifiers, and P contains only tell processes that maintain continuous constraints.
- A process of the form P_1 , P_2 is the *parallel composition* of sub-processes P_1 and P_2.
- A *hence* process `hence` P activates a process P along the time line for all $t > t_0$, where t_0 is the time at which the hence process is interpreted.
- We express an empty process by ϵ.

We consider Tiny HCC as a language scheme, thus constraint stores are parameterized in the language, and we do not specify a class of expressions that describe constraints.

4.3.2 Operational Semantics of Tiny HCC

We formalize the operational semantics of Tiny HCC by means of translation into RTTS as in Section 4.2.1. Gupta and others described the formal denotational and operational semantics of (default) HCC [39]. The main difficulty in the formalization of HCC is in describing the evolution of a constraint store that manages the uncertain values of variables along with the time line. Our semantics formalizes a hybrid trajectory that satisfies the constraint store for every instant over time. The evolution of the constraint store is observed as the whole set of such trajectories.

The operational semantics is described by associating the Tiny HCC programs with RTTSs.

Definition 24 (translation from Tiny HCC into RTTS) A program P in Tiny HCC is translated into an RTTS $\langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$ as follows. We express the tuple of all pure identifiers of variables appearing in a program by $x = (x_1, \ldots, x_n)$, and the tuple of variables of the form $id\_$ corresponding to x by $x_- = (x_{1-}, \ldots, x_{n-})$. We use the following tuples to express states in $\mathcal{S}$ of the RTTS and states internally used in transitions:

- Pairs $\langle P, v \rangle$ consisting of a Tiny HCC program P and a value v in the continuous state space $\mathbb{R}^n$. $\mathcal{S}$ is a set of these pairs.
- Quadruples $\langle P, c, H, m \rangle$ consisting of a Tiny HCC program P, a constraint store c, a Tiny HCC program H to be interpreted in the next phase, and a flag for switching between two modes $\{\mathbf{c}, \mathbf{d}\}$.

Before defining the transition relation $\mathcal{T}$, we define a transition relation $\langle P, c, H, m \rangle \rightsquigarrow \langle P', c', H', m \rangle$ by Rules (4.7)–(4.12) in Figure 4.7 corresponding

$$\langle c, c', H, m\rangle \rightsquigarrow \langle \epsilon, c' \wedge c, H, m\rangle \tag{4.7}$$

$$\frac{c \vdash c_g}{\langle(\texttt{if}\ \ c_g\ \ \texttt{then}\ \ P), c, H, m\rangle \rightsquigarrow \langle P, c, H, m\rangle} \tag{4.8}$$

$$\frac{\langle P_1, c, H, m\rangle \rightsquigarrow \langle P_1', c', H', m\rangle}{\langle(P_1\,\texttt{,}P_2), c, H, m\rangle \rightsquigarrow \langle(P_1'\,\texttt{,}P_2), c', H', m\rangle} \tag{4.9}$$

$$\langle(\texttt{hence}\ \ P), c, H, \mathbf{c}\rangle \rightsquigarrow \langle P, c, (H\,\texttt{,hence}\ \ P), \mathbf{c}\rangle \tag{4.10}$$

$$\langle(\texttt{hence}\ \ P), c, H, \mathbf{d}\rangle \rightsquigarrow \langle \epsilon, c, (H\,\texttt{,hence}\ \ P), \mathbf{d}\rangle \tag{4.11}$$

$$\langle cont(x_i), c, H, m\rangle \rightsquigarrow \langle cont(x_i), c \wedge cont(x_i), H, m\rangle \tag{4.12}$$

$$\frac{\begin{array}{c}\langle H, (x{=}v), \epsilon, \mathbf{c}\rangle \overset{*}{\rightsquigarrow} \langle P, c, H', \mathbf{c}\rangle, \quad \phi \models c, \quad t \in \mathbb{R}_{>0}, \\ \forall t' \in (0,t)\ (\langle P, (x{=}\phi(t')\ \wedge\ x_-{=}\phi(t')), \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P, c', \epsilon, \mathbf{d}\rangle)\end{array}}{\langle H, v\rangle \xrightarrow{t} \langle(P\,\texttt{,}H'), \phi(t)\rangle} \tag{4.13}$$

$$\frac{\langle P, (x_-{=}v_-), \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P', c, H, \mathbf{d}\rangle, \quad v \models c}{\langle P, v_-\rangle \xrightarrow{0} \langle H, v\rangle} \tag{4.14}$$

Figure 4.7 Operational semantics of Tiny HCC.

to each construct of Tiny HCC. When these transition rules are examined in the premises of Rules 4.13 and 4.14, the transitions are applied until no further transitions can take place.

- Rule (4.7) describes the tell process that conjuncts an atomic constraint c with the store c'.
- Rule (4.8) describes the ask processes. The guard constraint c_g should be entailed by the store c to take place the transition. Accordingly, the process P is taken out.
- Rule (4.9) represents the parallel composition of processes P_1 and P_2. We also consider the reduction of P_2 in the same way.
- Rules (4.10) and (4.11) describe the `hence` P process that expands the process P over the time line. It takes out P to apply the reduction in a

continuous change phase, and just disappears in a discrete change phase. In each phase, the process is copied into the third element to continue the process in the next phase.

- Rule (4.12) describes the tell process of the constraint $cont(x_i)$, where x_i is an element of x. The process is interpreted as in Rule (4.7), except for that it remains after the transition.

Now, we define the transitions in $\mathcal{T}$ by Rules (4.13) and (4.14) in Figure 4.7.

- Rule (4.13) describes continuous changes expressed as the transition $\langle H, v\rangle \xrightarrow{t} \langle (P, H'), \phi(t)\rangle$. In the first premise, the program H is interpreted with the store $x = v$. In the interpretation a hence process `hence` P' is reduced to the sub-processes P', and all ask processes in H remain in P. We then have a store c that can be regarded as an IVP-ODE, and we can obtain a trajectory ϕ satisfying the IVP-ODE (the second premise). In the fourth premise, we check that the set of ask processes in P does not change over the time interval $(0, t)$. Notice that an ask process disappears when the state (i.e., entailment relation) of the guard constraint on x or x_- changed.
- Rule (4.14) represents two kinds of transitions. First, it determines the set of initial states $\mathcal{S}_0$ of the RTTS. Assume we have $\langle P, \bullet\rangle$, where P is an input program and $\bullet$ is a special value denoting the undefined value. In the premise, we compute $\langle P, \mathbf{true}, \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P', c, H, \mathbf{d}\rangle$, where P' is a residual process not needed in this case, c stores the constraints added by the tell processes in P, and H is a composition of hence processes in P. An initial state in $\mathcal{S}_0$ is obtained as $\langle H, v\rangle$, where v is a value of x satisfying the constraint c.
- Second, Rule (4.14) describes the computation of discrete changes. In the premise, we compute $\langle P, (x_- = v_-), \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P', c, H, \mathbf{d}\rangle$, and obtain a constraint store c and a composed process H of hence processes in P. We then compute a value v that satisfies the constraint c, and an initial state $\langle H, v\rangle$ for the next continuous change. □

A difference between the executions of Tiny HCC and HA is that, in an execution of Tiny HCC, a discrete change occurs at the earliest state satisfying a guard constraint, i.e., an instantaneous transition should take place whenever the premises of Rule (4.14) hold.

4.3.3 Example of an Execution

In this section, we describe an execution of a Tiny HCC program.

Example 17 Below is a model in Tiny HCC of the bouncing particle described in the previous examples.

```
1: (p_x, p_y, v_x, v_y) ∈ [0, 0.1]×[1, 1.1]×[4.1]×[−0.4],
2: hence {
3:     cont(p_x), cont(p_y),
4:     if sin(p_x) − p_y > 0 then
5:       (ṗ_x, ṗ_y, v̇_x, v̇_y) = (v_x, v_y, 0, −g + k · v_y^2),
6:     if sin(p_x−) − p_y− = 0 then {
7:       m = (− cos(p_x−), 1), n = m/‖m‖,
8:       (v_x, v_y) = (v_x−, v_y−) − (e + 1) · n · dot(n, (v_x−, v_y−))
9: }}
```

We use the four-dimensional variable (p_x, p_y, v_x, v_y). The first line states the initial condition. The ask process at lines 4–5 describes the continuous movement of the particle in the air, and the ask process at lines 6–8 describes bounces of the particle at the ground.

In the following, we describe the execution of the above Tiny HCC program, henceforth referred to as P_{in}. We abbreviate sub-processes in P_{in} to P_a and P_h as follows:

- P_a :

  ```
  1: cont(p_x), cont(p_y),
  2: if sin(p_x−) − p_y− = 0 then {
  3:     m = (− cos(p_x−), 1), n = m/‖m‖,
  4:     (v_x, v_y) = (v_x−, v_y−) − (e + 1) · n · dot(n, (v_x−, v_y−))
  5: }
  ```

- P_h :

  ```
  1: hence {
  2:     P_a,
  3:     if sin(p_x) − p_y > 0 then
  4:       (ṗ_x, ṗ_y, v̇_x, v̇_y) = (v_x, v_y, 0, −g + k · v_y^2)
  5: }
  ```

We also abbreviate the following constraint to c_c:

$$cont(p_x) \wedge cont(p_y) \wedge (\dot{p_x}, \dot{p_y}, \dot{v_x}, \dot{v_y}) = (v_x, v_y, 0, -g + k \cdot v_y^2).$$

The execution of the program proceeds as follows:

1. We first obtain an initial state at time 0 (i.e., a state $s^0 \in \mathcal{S}_0$ of the associated RTTS). We apply Rule (4.14) to a state $\langle P_{in}, \bullet \rangle$, where $\bullet$ denotes

the undefined value, as follows:

$$\frac{\langle P_{in}, \mathbf{true}, \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle \epsilon, c_0, P_h, \mathbf{d}\rangle,\ v_0 \models c_0}{\langle P_{in}, \bullet\rangle \overset{0}{\rightarrow} \langle P_h, v_0\rangle} \quad (4.14)$$

The first premise is proved by applying the rules $\rightsquigarrow$ in Figure 4.7 as many times as possible. As a result, we obtain a quadruple $\langle \epsilon, c_0, P_h, \mathbf{d}\rangle$, where

$$c_0 \equiv (p_x, p_y, v_x, v_y) \in [0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4],$$

and P_h is the hence process in P_{in}. In the second premise, v_0 denotes a value for the variable (p_x, p_y, v_x, v_y) that satisfies the constraint c_0. Assume that we take the value $(0, 1.1, 4.1, -0.4)$.

2. The resulting state evolves continuously over time $t > 0$ with respect to Rule (4.13). Here, we consider the continuous transition $\overset{\delta}{\rightarrow}$, where $\delta \in \mathbb{R}_{>0}$ is a duration for an evolution (we assume a sufficiently small δ that causes no discrete change). The transition is formalized as follows:

$$\frac{\begin{array}{ll} & \forall t' \in (0, \delta) \\ \langle P_h, (x = v_0), \epsilon, \mathbf{c}\rangle \overset{*}{\rightsquigarrow} \langle P_a, c_0, P_h, \mathbf{c}\rangle, & (\langle P_a, (x = \phi_0(t') \wedge x_- = \phi_0(t')), \epsilon, \mathbf{d}\rangle \\ \phi_0 \models c_0,\ \delta \in \mathbb{R}_{>0}, & \overset{*}{\rightsquigarrow} \langle P_a, c_{t'}, \epsilon, \mathbf{d}\rangle) \end{array}}{\langle P_h, v_0\rangle \overset{\delta}{\rightarrow} \langle (P_a\,, P_h), \phi_0(\delta)\rangle} \quad (4.13)$$

The antecedent of $\overset{\delta}{\rightarrow}$ is the tuple obtained in Step 1, and the constraint store c_0 in the premise is set as

$$c_0 \equiv c_c \wedge (p_x, p_y, v_x, v_y) = (0, 1.1, 4.1, -0.4).$$

In the second premise $\phi_0 \models c_0$, a continuous trajectory ϕ_0 is obtained with respect to the IVP-ODE stored in c_0. In the last premise of Rule (4.13), we check that the closure of transitions $\overset{*}{\rightsquigarrow}$ does not change the program P_a for every $t' \in (0, \delta)$. For example, at time $t' = 0.1$, we have a state $\langle P_a, \phi_0(0.1)\rangle$, where $\phi_0(0.1) = (0.41, 1.01 \cdots, 4.1, 1.37 \cdots)$. As the result of transitions $\overset{*}{\rightsquigarrow}$, we have the quadruple $\langle P_a, c_{t'}, \epsilon, \mathbf{d}\rangle$, where

$$c_{t'} \equiv cont(p_x) \wedge cont(p_y) \wedge x_- = \phi_0(0.1),$$

and we can confirm that the first element is unchanged from the antecedent.

3. We describe another successive continuous transition $\xrightarrow{\delta'}$, after the transition in Step 2. We again assume a sufficiently small $\delta' \in \mathbb{R}_{>0}$. The transition is formalized as follows:

$$\frac{\begin{array}{ll}\langle (P_a \,,P_h), (x{=}\phi_0(\delta)), \epsilon, \mathbf{c}\rangle & \forall t' \in (0,\delta') \\ \quad \overset{*}{\rightsquigarrow} \langle P_a, c_\delta, P_h, \mathbf{c}\rangle, & (\langle P_a, (x{=}\phi_\delta(t') \wedge x_- {=} \phi_\delta(t')), \epsilon, \mathbf{d}\rangle \\ \phi_\delta \models c_\delta,\ \delta' \in \mathbb{R}_{>0}, & \quad \overset{*}{\rightsquigarrow} \langle P_a, c_{t'}, \epsilon, \mathbf{d}\rangle)\end{array}}{\langle (P_a \,,P_h), \phi_0(\delta)\rangle \xrightarrow{\delta'} \langle (P_a \,,P_h), \phi_\delta(\delta')\rangle} \quad (4.13)$$

   The reductions are computed as in Step 2. In the first premise, we compute the constraint store c_δ, where δ is the duration evolved in Step 2. When $\delta = 0.1$,

$$c_\delta \equiv c_c \wedge (p_x, p_y, v_x, v_y) = (0.41, 1.01\cdots, 4.1, 1.37\cdots).$$

   The continuous trajectory ϕ_δ is equivalent to ϕ_0 in Step 2 except that the initial time is shifted for $\delta = 0.1$.
4. Assume the duration $\delta = 0.147\cdots$, and the value $\phi_0(\delta) = (0.603\cdots, 0.934\cdots, 4.1, -1.84\cdots)$ of the trajectory ϕ_0 obtained in Step 2. Note that the constraint $\sin(\phi_0(\delta).1) - \phi_0(\delta).2 = 0$ holds. We can consider the continuous transition $\xrightarrow{\delta}$ from the initial state at time 0 as in Step 2. The time set $(0, 0.147\cdots)$ examined in the fourth premise of Rule (4.13) is the maximal set in this phase of continuous transitions because a discrete change causes within the time set $(0, \delta'')$, where $\delta'' > 0.147\cdots$.
5. Next, we assume the duration $\delta'' > 0.417\cdots$, and consider again the continuous transition $\xrightarrow{\delta''}$ from the initial state at time 0. The fourth premise in Rule (4.13) is checked by computing Rule (4.14) for $t' \in (0, \delta'')$. For $t' = 0.147\cdots$, the transition is applied as

$$\langle P_a, (x{=}\phi_0(t') \wedge x_- {=} \phi_0(t')), \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P'_a, c_{t'}, \epsilon, \mathbf{d}\rangle$$

   The process P_a in the antecedent is changed into P'_a:

```
1: cont(p_x) , cont(p_y)
```

   An ask process in P_a disappears by Rule (4.8) because the constraint $x_- = \phi_0(t')$ entails the guard constraint $\sin(p_{x-}) - p_{y-} = 0$. Accordingly, the transition $\xrightarrow{\delta''}$ from the initial state is not possible.
6. We adopt the instantaneous transition at time $t^1 = 0.147\cdots$ as follows:

$$\frac{\langle (P_a \,,P_h), (x_- = \phi_0(t^1)), \epsilon, \mathbf{d}\rangle \overset{*}{\rightsquigarrow} \langle P'_a, c_{t^1}, P_h, \mathbf{d}\rangle,\ v_{t^1} \models c_{t^1}}{\langle (P_a \,,P_h), \phi_0(t^1)\rangle \xrightarrow{0} \langle P_h, v_{t^1}\rangle} \quad (4.14)$$

The resulting constraint store is as follows:

$$c_{t^1} \equiv \mathit{cont}(p_x) \wedge \mathit{cont}(p_y) \wedge (v_x, v_y) = (0.283\cdots, 3.53\cdots),$$

and the value v_{t^1} is evaluated by the premise $v_{t^1} \models c_{t^1}$ as follows:

$$v_{t^1} = (0.603\cdots, 0.934\cdots, 0.283\cdots, 3.53\cdots),$$

where $\mathit{cont}(p_x)$ and $\mathit{cont}(p_y)$ assign the value in the previous state (i.e., $\phi_0(t^1)$) to the variables p_x and p_y.

7. The resulting state in Step 6 evolves continuously over time $t^1 + \delta$ ($\delta \in \mathbb{R}_{>0}$) as in Step 2.

$$\frac{\begin{array}{ll} & \forall t' \in (0,\delta) \\ \langle P_h, (x = v_{t^1}), \epsilon, \mathbf{c}\rangle \overset{*}{\rightsquigarrow} \langle P_a, c_{t^1}, P_h, \mathbf{c}\rangle, & (\langle P_a, (x = \phi_{t^1}(t') \wedge x_- = \phi_{t^1}(t')), \epsilon, \mathbf{d}\rangle \\ \phi_{t^1} \models c_{t^1},\ \delta \in \mathbb{R}_{>0}, & \overset{*}{\rightsquigarrow} \langle P_a, c_{t'}, \epsilon, \mathbf{d}\rangle) \end{array}}{\langle P_h, v_{t^1}\rangle \xrightarrow{\delta} \langle (P_a \,,P_h), \phi_{t^1}(\delta)\rangle} \quad (4.13)$$

The constraint store is computed as

$$c_{t^1} \equiv c_c \wedge (p_x, p_y, v_x, v_y) = (0.603\cdots, 0.934\cdots, 0.283\cdots, 3.53\cdots).$$

□

Chapter 5

Hybrid Constraint Systems

Hybrid systems are modeled by *constraints* whose domain is the space-time continuum [39, 46], and which are equations of real numbers, functions, and intervals (inequalities). Reliable simulation are done by integrating the computation of continuous dynamics and discrete changes, and by handling the uncertainties and computation errors. It is not obvious to reliably compute hybrid systems described by nonlinear ODEs and nonlinear conditions for discrete changes. This chapter presents a framework for such nonlinear problems.

- We propose *hybrid constraint systems* (HCSs) to describe the problem of detecting discrete changes by constraints (Section 5.2). We later describe how an HCS serves as a key component in the simulation (Section 5.6.1) and verification (Section 6) of hybrid systems. HCSs are defined based on *continuous constraint systems* (Section 5.1). A constraint in HCSs is either an *instantaneous constraint*, a *continuous constraint* on trajectories, or a *guard constraint* on continuous states. HCSs are considered as a class of *algebraic differential equations* [64]. We formulate the *box-consistency* for HCSs to develop a local consistency technique as in Section 3.3.
- We then develop a consistency technique for solving HCSs on top of an interval-based method for nonlinear ODEs (Section 2.3) and an interval-based constraint programming framework (Section 3.3). The technique generates a set of boxes smaller than a specified size that encloses theoretical solutions.
- The proposed technique employs the interval Newton method to achieve the quadratic convergence in the reduction of boxes (Section 5.4.1) and to guarantee that a box contain a solution (Section 5.4.3). The method uses an interval Newton operator derived from constraints. Experimental results indicate that the method is efficient for solving HCSs with nonlinear constraints (Section 5.6).

5.1 Continuous Constraint Systems

For *continuous constraint systems* (CCSs), we describe constraints on continuous trajectories.

Definition 25 (continuous constraint system) A *continuous constraint system* (CCS) is a constraint system $\langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \mathcal{C} \rangle$ consisting of:

- A variable y representing an n-dimensional continuous trajectory.
- A domain $\mathcal{M}(\mathcal{D}_t, \mathcal{D})$ of the variable that is a whole set of continuous functions over time $\mathcal{D}_t \to \mathcal{D}$, where $\mathcal{D}_t \subseteq \mathbb{R}_{\geq 0}$ and $\mathcal{D} \subseteq \mathbb{R}^n$.
- A set $\mathcal{C}$ of constraints $c \subseteq \mathcal{M}(\mathcal{D}_t, \mathcal{D})$.

A valuation in a CCS is a map of the form $y \mapsto \phi$, where $\phi : \mathcal{D}_t \to \mathcal{D}$. A solution is a valuation satisfying every constraint in $\mathcal{C}$. □

In the following, we consider CCSs described in the form of IVP-ODEs. We handle two kinds of constraints in a CCS corresponding to the initial condition and ODE in an IVP-ODE. The first is an *instantaneous constraint* or an *initial constraint* and describes the value of a trajectory at a certain time point (e.g., an initial condition for a trajectory). The constraints are generalized to describe a region in the space-time continuum with parameterized values and time (e.g., time intervals).

Definition 26 (instantaneous constraint) An *instantaneous constraint* $c_i \in \mathcal{C}$ of a CCS is described by a formula of the form

$$\exists (\tau_0, v_0) \in c \ (y(\tau_0) = v_0),$$

where c is a real constraint ranging over the domain $\mathcal{D}_t \times \mathcal{D}$. □

The second constraint is called a *continuous constraint*, and describes an ODE, which is a relation between the values and derivatives of a trajectory over time.

Definition 27 (continuous constraint) A *continuous constraint* $c_c \in \mathcal{C}$ of a CCS is described by a formula of the form

$$\forall \tau \in \mathcal{D}_t \ ((y(\tau), \dot{y}(\tau)) \in c),$$

where c is a real constraint ranging over the domain $\mathcal{D} \times \mathbb{R}^n$. □

In the following, we consider CCSs of the form $\langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \mathcal{C}_i \cup \mathcal{C}_c \rangle$, where $\mathcal{C}_i$ is a set of instantaneous constraints and $\mathcal{C}_c$ is a set of continuous constraints. A solution is a valuation that satisfies an IVP-ODE described by $\mathcal{C}_i$ and $\mathcal{C}_c$. The entailment relation between instantaneous constraints or between continuous constraints are derived from the entailment relation between the real constraints c used in the description of each constraint.

Example 18 A falling particle is modeled by a CCS $\langle y, \mathcal{M}(\mathbb{R}_{\geq 0}, \mathbb{R}^4), \{c_i, c_c\} \rangle$, where

$$\begin{aligned} y &= (y_{p_x}, y_{p_y}, y_{v_x}, y_{v_y}), \\ c_i &\equiv \exists (\tau_0, v_0) \in [0] \times [0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4] \ (y(\tau_0) = v_0), \\ c_c &\equiv \forall \tau \in \mathbb{R}_{\geq 0} \ (\dot{y}(\tau) = (y_{v_x}(\tau), y_{v_y}(\tau), 0, -g + k \cdot y_{v_y}(\tau)^2)). \end{aligned}$$

The variable y represents a function $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^4$. Given a time $t \in \mathbb{R}_{\geq 0}$, $(y_{p_x}(t), y_{p_y}(t))$ and $(y_{v_x}(t), y_{v_y}(t))$ represent the position and velocity of the particle, respectively. the constants g and k represent the gravity acceleration and air resistance, respectively. The instantaneous constraint c_i specifies the initial state of the particle at time $[0]$ as the region $[0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4]$. The continuous constraint c_c describes the movement of the particle by a constraint on the state $y(\tau)$ and derivative $\dot{y}(\tau)$. Figure 5.1 illustrates the set Φ of solution trajectories of the particle with parameters set to $g = 9.8$ and $k = 10^{-3}$. □

5.2 Hybrid Constraint Systems

We have seen that continuous trajectories over time $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n$ are described by CCSs. *Hybrid constraint systems* (HCSs) describe the crossing points of the trajectories with time-invariant boundaries in the state space $\mathbb{R}^n$. Figure 5.2 illustrates an example of an HCS consisting of the trajectory in the previous example and a sine-curved surface. An HCS is constructed by extending a CCS to model a trajectory, which is done by adding to the CCS a *guard constraint*.

Definition 28 (hybrid constraint system) Consider a CCS $\langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \mathcal{C}_i \cup \mathcal{C}_c \rangle$, where y represents an n-dimensional trajectory, $\mathcal{D}_t \subseteq \mathbb{R}_{\geq 0}$, $\mathcal{D} \subseteq \mathbb{R}^n$, and $\mathcal{C}_i \cup \mathcal{C}_c$ is a set of instantaneous and continuous constraints.

A *hybrid constraint system* (HCS) is a tuple $\langle \tilde{x}, \mathcal{D}_t \times \mathcal{D}, \mathcal{C}_g, \langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \mathcal{C}_i \cup \mathcal{C}_c \rangle \rangle$, where the inner tuple is the CCS described above, and the other elements consist of the following:

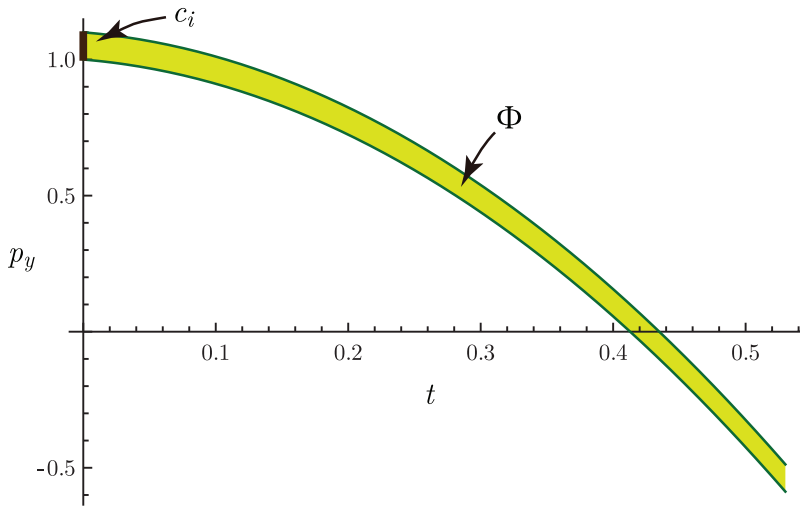

Figure 5.1 Example of CCSs.

- An $(n+1)$ vector variable $\tilde{x} = (t, x_1, \ldots, x_n)$ ranging over the space-time $\mathbb{R}_{\geq 0} \times \mathbb{R}^n$.
- A domain $\mathcal{D}_t \times \mathcal{D} \subseteq \mathbb{R}_{\geq 0} \times \mathbb{R}^n$.
- A set $\mathcal{C}_g$ of *guard constraints* in which a constraint $c_g \in \mathcal{C}_g$ is described by the following formula

$$y(t) = (x_1, \ldots, x_n) \wedge (x_1, \ldots x_n) \in c,$$

where c is a real constraint ranging over the domain $\mathcal{D}$.

The domain $\mathcal{D}_t$ that corresponds to the variable t is called the *time domain*.

A *valuation* of an HCS is a map of the form $\tilde{x} \mapsto v$, where $v \in \mathcal{D}_t \times \mathcal{D}$. A *solution* of an HCS is a valuation $\tilde{x} \mapsto v$ where the value v satisfies every guard constraint in $\mathcal{C}_g$ with respect to the every solution $y \mapsto \phi$ of the inner CCS. Accordingly, for a solution $\tilde{x} \mapsto v$ and a guard constraint c_g of an HCS, the value v satisfies

$$\exists \phi \in \Phi \ (\phi(v.1) = (v.2, \ldots, v.(n+1)) \wedge (v.2, \ldots, v.(n+1)) \in c),$$

where Φ is the set of all solutions of the inner CCS, and c is a real constraint described in c_g. □

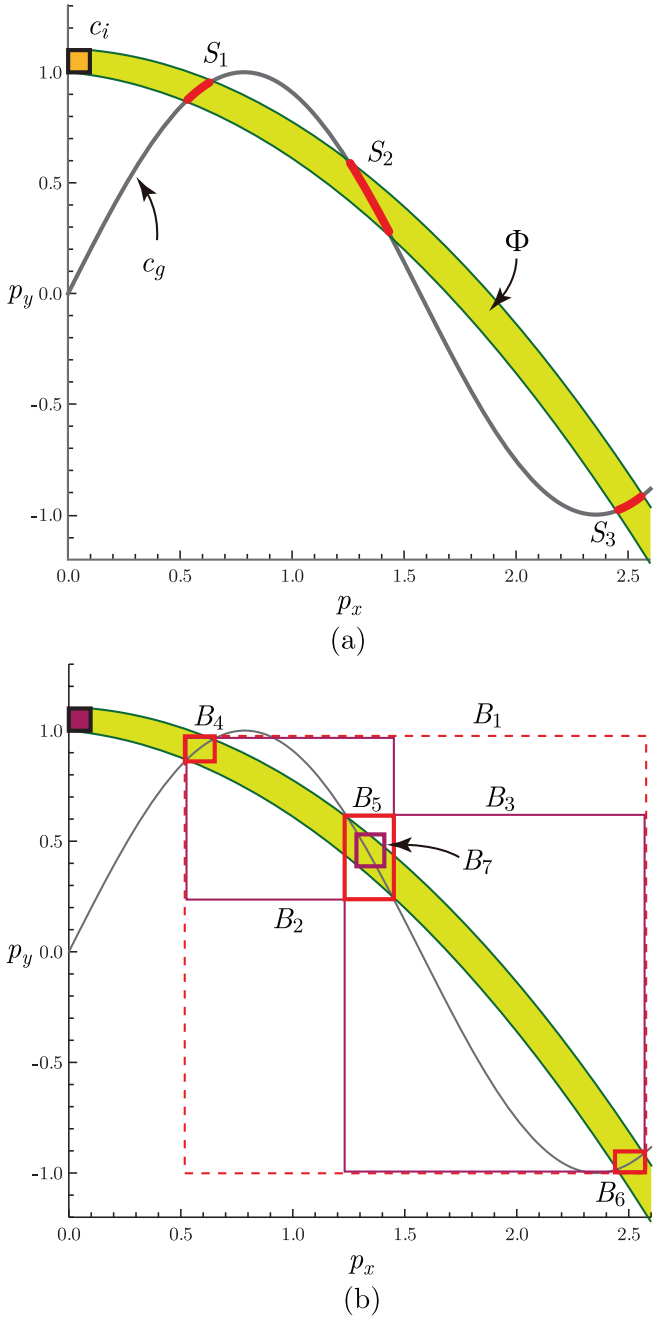

Figure 5.2 Example of HCSs.

Example 19 A particle that bounces off a sinusoidal surface is modeled by an

HCS $\langle \tilde{x}, \mathbb{R}_{\geq 0} \times \mathbb{R}^4, \{c_g\}, \langle y, \mathcal{M}(\mathbb{R}_{\geq 0}, \mathbb{R}^4), \{c_i, c_c\}\rangle\rangle$, where

$$\begin{aligned}
\tilde{x} &= (t, x) = (t, p_x, p_y, v_x, v_y), \\
y &= (y_{p_x}, y_{p_y}, y_{v_x}, y_{v_y}), \\
c_g &\equiv (y(t) = (p_x, p_y, v_x, v_y) \wedge \sin(2 \cdot p_x) = p_y), \\
c_i &\equiv \exists (\tau_0, v_0) \in [0] \times [0, 0.1] \times [1, 1.1] \times [4.1] \times [-0.4]\ (y(\tau_0) = v_0), \\
c_c &\equiv \forall \tau \in \mathbb{R}_{\geq 0}\ (\dot{y}(\tau) = (y_{v_x}(\tau), y_{v_y}(\tau), 0, -g + k \cdot y_{v_y}(\tau)^2)).
\end{aligned}$$

The variable t represents time, and variables (p_x, p_y) and (v_x, v_y) represent the position and velocity of the particle, respectively. The CCS $\langle y, \mathcal{M}(\mathbb{R}_{\geq 0}, \mathbb{R}^4), \{c_i, c_c\}\rangle$ is equivalent to the CCS in Example 18 that models the continuous trajectory $\phi : \mathbb{R}_{\geq 0} \to \mathbb{R}^4$ of the particle. In c_g, the value of x is associated with the value of the trajectory ϕ at time t. Contact of the particle with the surface is detected by checking whether the condition $\sin(2 \cdot p_x) = p_y$ holds. Figure 5.2 (a) illustrates the trajectory of the particle with parameters set to $g = 9.8$ and $k = 10^{-3}$. Solutions are illustrated as the regions S_1, S_2, and S_3. For every point $v \in S_1 \cup S_2 \cup S_3$, we have a solution $\tilde{x} \mapsto v$. □

Remark 4 Consider an HCS and a solution $y \mapsto \phi$ of the inner CCS. Then, there may be multiple solutions of the HCS with respect to ϕ. When applying HCSs to the simulation of hybrid systems, the one in which we are interested is the *earliest* solution. □

Example 20 In Figure 5.2 (a), the guard constraint has three solutions inside each of the regions S_1, S_2 and S_3, for each trajectory described by the inner CCS. The particle bounces at the earliest solution in S_1. □

In our method for solving HCSs, we consider a restricted class of HCSs called *simple* HCSs.

Definition 29 (simple HCS) A *simple* HCS is an HCS $\langle \tilde{x}, \mathcal{D}_t \times \mathcal{D}, \{c_g\}, \langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \{c_i, c_c\}\rangle\rangle$, where $\tilde{x}$ consists of $(n{+}1)$ real variables, $\mathcal{D}_t \subseteq \mathbb{R}_{\geq 0}$, $\mathcal{D} \subseteq \mathbb{R}^n$, and constraints c_i, c_c, and c_g are defined as follows:

$$\begin{aligned}
c_i &\equiv \exists (\tau_0, v_0) \in \mathcal{D}_0\ (y(\tau_0) = v_0), \\
c_c &\equiv \forall \tau \in \mathcal{D}_t\ (\dot{y}(\tau) = f(y(\tau))), \\
c_g &\equiv (y(t) = (x_1, \ldots, x_n) \wedge grd(x_1, \ldots x_n) = 0).
\end{aligned}$$

$\mathcal{D}_0 = I_0 \times \cdots \times I_n$ corresponds to a box $(I_0, \ldots, I_n) \in \mathbb{I}^n$, where $\underline{I_0} \geq 0$, f is a Lipschitz continuous function $\mathcal{D} \to \mathbb{R}^n$, and *grd* is a differentiable function $\mathcal{D} \to \mathbb{R}$. □

The HCS in Example 19 is a simple HCS. Additionally, we define *solved* HCSs as in Section 3.1.

Definition 30 (solved HCS) Let $\langle \tilde{x}, \mathcal{D}_t \times \mathcal{D}, \{c_g\}, \langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \{c_i, c_c\}\rangle\rangle$ be a simple HCS. When, for every $v \in \mathcal{D}$, $\tilde{x} \mapsto v$ is a solution of the HCS, then the HCS is *solved*. □

5.3 Box-Consistency for HCSs

Interval-based solving of an HCS means refining a box in $\mathbb{I}^{n+1}$ into another box to ensure that the system is box-consistent, as in the context of RCSs (Section 3.3). We can induce the box-consistency for hybrid constraints from Definition 14 in the previous chapter by modifying the definition so that it conforms to the HCSs in Definition 29.

Definition 31 (box-consistency for HCS) Let $\langle \tilde{x}, \mathcal{D}_t \times \mathcal{D}, \{c_g\}, \langle y, \mathcal{M}(\mathcal{D}_t, \mathcal{D}), \{c_i, c_c\}\rangle\rangle$ be a simple HCS, let *grd* be the function described in the guard constraint c_g, and let ϕ be a continuous trajectory satisfying c_i and c_c. Consider interval extensions *Grd* of *grd* and Φ of ϕ. Note that Φ should enclose all the possible trajectories ϕ. For a domain $\mathcal{D}_t \times \mathcal{D}$, the HCS is *box-consistent* if and only if every i-th projection of $\mathcal{D}_t \times \mathcal{D} = I_0 \times I_1 \times \cdots \times I_n$ satisfies the following conditions with respect to the other projections:

$$I_0 \;=\; apx_{\mathbb{I}}(\{v_0 \in I_0 \mid Grd(\Phi(apx_{\mathbb{I}}(v_0 \pm [0, h_{min}]))) \ni 0\}) \tag{5.1}$$

for $i = 0$, where $h_{min} \in \mathbb{R}_{>0}$.

$$\begin{aligned} I_i \;=\; apx_{\mathbb{I}}(\{v_i \in I_i \mid\; & v_i \in \Phi(I_0).i \\ & \wedge\; Grd(I_1, \ldots, I_{i-1}, apx_{\mathbb{I}}(v_i), I_{i+1}, \ldots, I_n) \ni 0\}) \end{aligned} \tag{5.2}$$

for $i \in \{1, \ldots, n\}$. The parameter h_{min} is derived from the VNODE method because it is difficult to compute $\Phi(apx_{\mathbb{I}}(v_0))$ (see Section 2.3). □

Example 21 Consider the HCS in Example 19. For boxes $B_i = (I_{i,0}, \ldots, I_{i,4})$ $(i \in \{1 \ldots, 7\})$ illustrated in Figure 5.2 (b), HCSs $\langle \tilde{x}, I_{i,0} \times \cdots \times I_{i,4}, \{c_g\}, \langle y, \mathcal{M}(I_{i.0}, I_{i,1} \times \cdots \times I_{i,4}), \{c_i, c_c\}\rangle\rangle$ are box-consistent. □

Example 22 For Example 19, a box $B = ([0.110, 0.148], [0.491, 0.636], [0.866, 0.982], [4.10], [-1.85, -1.48])$, where $B = apx_{\mathbb{F}_{10,-11,12,3}}(B_4)$, is appropriate for composing the box-consistent HCS. □

5.4 Technique for Solving HCSs

In this section, a technique for computing box-consistent HCSs is described. The technique applies the BranchAndPrune algorithm (Section 3.3.1) to HCSs with instantiations of the sub-algorithms, HCSRevise, HCSReviseL, and HCSReviseU (these are variants of BC3Revise, BC3ReviseL and BC3ReviseU in Section 3.3.3). The proposed method computes a set of boxes that encloses all the solutions in the initial domain. The accuracy of the results is specified by the positive real-valued parameters w_{max} and h_{min}. An input w_{max} to BranchAndPrune determines the maximum width of the intervals in a result. The parameter h_{min} is used to enforce the box-consistency as described in Definition 31. Interval extensions of constraints should be satisfied within h_{min} from each bound of the time domain.

The Branch (Figure 3.2) and Prune (Figure 3.3) algorithms are used in BranchAndPrune. Prune reduces the domain of an HCS by enforcing the box consistency. Prune takes a set $\{c_i, c_c, c_g\}$ of instantaneous, continuous, and guard constraints, and a box B as inputs. Prune iteratively reduces each component of B until the fixpoint is reached.

To reduce the time component $B.1$, Prune calls the HCSRevise algorithm (line 4). For other components $B.i$ ($i \in \{2, \ldots, n{+}1\}$), Prune calls the instantiations for RCSs such as HC4Revise and BC3Revise. Details on the reduction procedures are supplied in Sections 5.4.1 and 5.4.2.

5.4.1 Reduction of the Time Domain

The HCSRevise and HCSReviseL algorithms for filtering the time domain are illustrated in Figures 5.3 and 5.4. We can reduce the time domain efficiently by applying the interval Newton method that solves continuous and guard constraints simultaneously. At line 2 in Figure 5.3, the algorithm constructs functions H and $\dot{H}$ for the interval Newton method. The essential idea is that the solving process of IVP-ODEs described in Section 2.3, denoted hereafter as Φ, is used to construct an interval Newton operator. Given an interval $T \in \mathbb{IF}$, the value of the interval function $\Phi(T)$ is obtained by iterative calculations with respect to CCS constraints c_i and c_c. This process Φ is prepared at line 1. Accordingly, the functions H and $\dot{H}$ are functions $\mathbb{IF} \to \mathbb{IF}$ given by

$$H(I_0) = Grd(\Phi(I_0)), \quad \dot{H}(I_0) = \sum_{i=1}^{n} \left(\frac{\partial Grd(\Phi(I_0))}{\partial X_i} \cdot \dot{\Phi}(I_0).i \right),$$

Input: initial constraint c_i, continuous constraint c_c,
guard formula $grd(x_1, \ldots, x_n) = 0$, box $B = (I_0, I_1, \ldots, I_n)$
Output: box-consistent box B

1: $\Phi :=$ establish the process for solving $c_i \wedge c_c$
2: $(H, \dot{H}) :=$ construct operators from Φ and Grd
3: $I_0 := \mathrm{HCSReviseL}(H, \dot{H}, I_0) \uplus \mathrm{HCSReviseU}(H, \dot{H}, I_0)$
4: $(I_1, \ldots, I_n) := (I_1, \ldots, I_n) \cap \Phi(I_0)$
5: return $(I_0, I_1, \ldots, I_n)$

Figure 5.3 HCSRevise algorithm.

Input: interval function H, interval function $\dot{H}$, time interval I_0
Output: interval I_0

1: **if** $0 \notin H(I_0)$ **then**
2: return $\emptyset$
3: **if** $0 \in H([\underline{I_0}, \underline{I_0} + h_{min}])$ **then**
4: return I_0
5: **else**
6: $I_0 := N^*_{H,\dot{H}}(I_0)$
7: $(I_{0,l}, I_{0,u}) := ([\underline{I_0}, \mathrm{m}(I_0)], [\mathrm{m}(I_0), \overline{I_0}])$
8: $I_{0,l} := \mathrm{HCSReviseL}(H, \dot{H}, I_{0,l})$
9: **if** $I_{0,l} \neq \emptyset$ **then**
10: return $I_{0,l}$
11: **else**
12: return $\mathrm{HCSReviseL}(H, \dot{H}, I_{0,u})$

Figure 5.4 HCSReviseL algorithm.

where Grd is an interval extension of grd used in c_g, Φ is an interval extension of ϕ, and $\partial Grd(X_1, \ldots, X_n)/\partial X_i$ and $\dot{\Phi}$ are interval extensions of the derivatives $\partial grd(x_1, \ldots, x_n)/\partial x_i$ and $d\phi(\tau)/d\tau$ ($i \in \{1, \ldots, n\}$). To compute $\partial Grd(X_1, \ldots, X_n)/\partial X_i$, we apply automatic differentiation to the computation of Grd.

At line 3 of HCSRevise, procedures HCSReviseL (Figure 5.4) and HCSReviseU reduce the lower and upper edges of the time interval, respectively. The procedure of HCSReviseL is as follows (HCSReviseU is similar except that it operates on the upper edge instead of the lower edge):

1. First, check the guard constraint for the current time domain I_0 (line 1).

When the interval $H(I_0)$ does not contain 0, the algorithm returns $\emptyset$ because the guard constraint should not hold over I_0.
2. Check whether $0 \in H([\underline{I_0}, \underline{I_0}+h_{min}])$ is satisfied, where h_{min} is the minimal step width for solving ODEs. If so, return the interval I_0 (line 3).
3. Calculate the fixpoint of the interval Newton method $I_0' = N^*_{H,\dot{H}}(I_0)$ (line 6). To obtain $N^*_{H,\dot{H}}(I_0)$, $I_0' = N^*_{H,\dot{H}}(I_0)$ is repeatedly computed until the ratio of I_0' to I_0 is under a threshold.
4. Split the refined I_0 in two and apply HCSReviseL recursively for each interval (lines 7–12).

The above procedure reduces the time domain of an HCS by applying the interval Newton method or by casting out a sub-interval at line 2 of HCSReviseL. Hence, the computed result encloses the solutions of the HCS. As we are applying the interval Newton method, we can guarantee the existence of a unique solution within a contracted box, under a certain condition.

Theorem 3 (existence and uniqueness of a solution (1)) Let $\langle \tilde{x}, I_0 \times \cdots \times I_n, \{c_g\}, \langle y, \mathcal{M}(I_0, I_1 \times \cdots \times I_n), \{c_i, c_c\}\rangle\rangle$ be a simple HCS, and let $N_{H,\dot{H}}$ be the interval Newton operator constructed from the constraints as described above. When $I_0' = N_{H,\dot{H}}(I_0)$, a unique solution of the HCS exists within $I_0' \times \ldots \times I_n'$, where $(I_1', \ldots, I_n') = (I_1, \ldots, I_n) \cap \Phi(I_0')$, if the following conditions hold:

- A unique continuous trajectory ϕ over I_0 exists with respect to the constraints c_i and c_c.
- The function *grd* described in c_g is differentiable over $I_1 \times \ldots \times I_n$.
- $I_0' \subseteq \mathrm{int}(I_0)$. □

Proof. From the first and second conditions, a unique function $grd \circ \phi$ exists and is differentiable over I_0. Because $Grd \circ \Phi$, which is composed by HCSRevise is an interval enclosure of $grd \circ \phi$, the property of the interval Newton method proves that a single root of $grd \circ \phi$ exists within I_0. The third condition is for the guarantee by the interval Newton method. ■

Checking the conditions in Theorem 3 is automated as follows. For the first condition, VNODE validates the existence and uniqueness of a solution in the computed enclosure. The third condition will be tested in the computation of HCSReviseL and HCSReviseU.

5.4.2 Reduction of the Continuous State Domain

Once the HCSReviseL and HCSReviseU algorithms reduce the time domain I_0 of an HCS, VNODE (Section 2.3) can compute an interval enclosure $\Phi(I_0)$ of the every possible continuous trajectories ϕ over I_0 (line 4 of Figure 5.3). Since the solutions of HCSs should be the values satisfying the guard constraint c_g, we can further reduce this enclosure.

We consider an RCS $\langle x, \mathcal{D}, \mathcal{C} \rangle$ consisting of the variable $x = (x_1, \ldots, x_n)$, the domain $\mathcal{D} = I_1 \times \ldots \times I_n$, where $(I_1, \ldots, I_n)$ is the intersection of $\Phi(I_0)$ and the HCS's domain, and the constraint set $\mathcal{C} = \{grd(x_1, \ldots, x_n) = 0\}$, where $grd(x_1, \ldots, x_n) = 0$ is specified by the guard constraint c_g. Thus, we can reduce the domain $I_1 \times \ldots \times I_n$ by applying the filtering procedure, such as BC3Revise$(grd(x_1, \ldots, x_n) = 0, (I_1, \ldots, I_n))$, for example.

In the solving process by Prune, we can attach different instantiations of Revise for each constraint in the system (line 4 of Figure 3.3). We attach (i) HCSRevise with the corresponding set of constraints in the HCS, and (ii) an instance for RCSs (e.g., BC3Revise) with the guard constraint $grd(x_1, \ldots, x_n) = 0$. The output of Prune with the above configuration forms the box-consistent system.

Lemma 4 (box-consistency of the revised systems) Consider a simple HCS $\langle \tilde{x}, I_0 \times \cdots \times I_n, \{c_g\}, \langle y, \mathcal{M}(I_0, I_1 \times \cdots \times I_n), \{c_i, c_c\} \rangle \rangle$. Assume we compute $(I'_0, \ldots, I'_n) =$ Prune$(c_i \wedge c_c \wedge (grd(x_1, \ldots, x_n) = 0), (I_0, \ldots, I_n))$, where Prune is configured as indicated above. Then, the HCS $\langle \tilde{x}, I'_0 \times \cdots \times I'_n, \{c_g\}, \langle y, \mathcal{M}(I_0, I_1 \times \cdots \times I_n), \{c_i, c_c\} \rangle \rangle$ is box-consistent. □

Proof. The each bound of I'_0 satisfies the constraints of the HCS because it is checked by HCSReviseL (at line 3) and HCSReviseU. This is equivalent to checking of the condition 5.1 in Definition 31. Other conditions 5.2 are satisfied because $I_1, \ldots, I_n$ are reduced by the Revise instantiation for RCSs. ■

5.4.3 Testing the Unique Existence of a Solution

A continuous trajectory described by an HCS may have multiple solutions (Remark 4). Hence, the number of solutions contained in a set of boxes computed by BranchAndPrune is unknown. We can obtain a proof of the uniqueness and existence of a solution within a result reduced by the interval Newton method (Theorem 3) under a certain condition.

Theorem 4 (existence and uniqueness of a solution (2)) Consider a simple HCS $\langle \tilde{x}, I_0 \times \ldots \times I_n, \{c_g\}, \langle y, \mathcal{M}(I_0, I_1 \times \cdots \times I_n), \{c_i, c_c\}\rangle\rangle$. Suppose a call to the procedure PRUNE$(c_i \wedge c_c \wedge (grd(x_1, \ldots, x_n) = 0), (I_0, \ldots, I_n))$ returns a box $(I'_0, \ldots, I'_n)$. In the computation, suppose the procedure HCSREVISEL or HCSREVISEU reduces a time interval I''_0 to $N_{H,\dot{H}}(I''_0)$, and the following conditions hold. Then, a unique solution of the HCS exists in the domain $I'_0 \times \cdots \times I'_n$.

- A unique continuous trajectory ϕ over I''_0 exists with respect to c_i and c_c.
- The function *grd* described in c_g is differentiable over $I''_1 \times \ldots \times I''_n$.
- $N_{H,\dot{H}}(I''_0) \subseteq \mathrm{int}(I''_0)$.
- $I'_0 \subseteq I''_0$. □

Proof. From the first to third conditions, it is proved that a root of $grd \circ \phi$ uniquely exists in I''_0 from Theorem 3. Since the whole computation by PRUNE completely encloses the solution, the unique solution also exists within $I'_0 \times \cdots \times I'_n$. ■

Example 23 Recall the boxes in Figure 5.2 (b) described in Example 21. Each of the boxes B_4, B_5, B_6, and B_7 encloses a unique solution. The boxes B_1, B_2, and B_3 should not be guaranteed to contain a solution. □

5.4.4 Computing an Enclosure for the Earliest Solution

The union of boxes computed by the proposed technique may enclose multiple solutions. A box that encloses the earliest solution is selected as follows.

1. Compute the clusters of boxes by concatenating two boxes if they are adjacent.
2. Find the cluster containing the earliest time.
3. If this cluster intersects with the initial value set, then discard this cluster and search for the next earliest cluster.

Note that we assume each of the clusters computed in Step 1 and the initial value set in Step 3 enclose a unique solution.

5.5 Implementation

We implemented the proposed method on top of the `Elisa` system described in Section 3.3.4, which is an implementation of the BRANCHANDPRUNE algorithm. Our implementation consists of about 4000 lines of C++ code.

The main classes are illustrated in Table 5.1. We implemented classes shown in the upper half of the table for representing continuous constraints. We represent

Table 5.1 Main classes in the implementation.

object type	class
variables	`Time`, `ContVar`
constraints	`ContConstraint`
models	`HybridModel`
algorithm	**class**
HCSRevise	`HybridNewtonReduction`
Select	`HybridChoice`

guard constraints as real constraints in the implementation. HCSRevise was implemented as the `HybridNewtonReduction` class by extending the `Reduction` class in `Elisa`. In `HybridNewtonReduction`, we implemented two strategies to manage the solving process of IVP-ODEs. These strategies are optimised for computing only the earliest solution, and for computing all the solutions within an input box, respectively. The Select procedure used in Prune algorithm (Figure 3.3, line 3) was tweaked by the `HybridChoice` class to preferentially apply HCSRevise before applying other Revise implementations for RCSs.

The `VNODE-LP` solver described in Section 2.3 was used to solve IVP-ODEs. We extended it to control the upper bounds of step sizes in the computation. Our implementation also caches the results by `VNODE-LP` for reuse. The parameters were set as $k = 20$, $atol = 10^{-20}$, and $rtol = 10^{-20}$. We also used the `FADBAD++` library [77] for automatic differentiation.

5.6 Examples and Experiments

Section 5.6.1 describes a simulation of a bouncing particle by modeling each bounce of the particle as an HCS. We then describe several HCS examples that involve nonlinear constraints (Section 5.6.2). Section 5.6.3 reports the results of comparisons with the Mathematica system. Section 5.6.4 reports the results of computing all the solutions of the above examples within the input box.

The parameters in the proposed method were set as $w_{max} = 10^{-2}$ and $h_{min} = 10^{-13}$. BC5Revise was used for reduction of the continuous state domain. In the experiments of Sections 5.6.1 and 5.6.3, we modified the implementation to terminate the computation after an enclosure for the earliest solution was obtained.

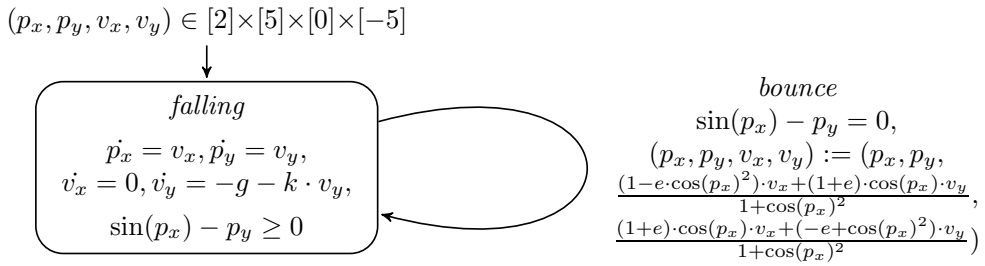

Figure 5.5 Model of bouncing particle in HA.

5.6.1 Interval-based Simulation of Bouncing Particle

In this section, we describe an interval-based simulation of the bouncing particle model of Example 13. Here, we modify the HA as illustrated in Figure 5.5 (assuming $g = 9.8, k = 0.3$, and $e = 0.8$).

From the model, we have the initial domain $[2] \times [5] \times [0] \times [-5]$ and a trace (Definition 21) of an execution of the form $\{\mathit{falling}^1, \ldots, \mathit{falling}^k\}$. Then, we can construct an HCS corresponding to each step of the execution. For example, the first step is described by an HCS $\langle \tilde{x}, [0, 10^3] \times \mathbb{R}^4, \{c_g\}, \langle y, \mathcal{M}(\mathbb{R}_{\geq 0}, \mathbb{R}^4), \{c_i, c_c\}\rangle\rangle$, where

$$
\begin{aligned}
\tilde{x} &= (t, p_x, p_y, v_x, v_y), \\
y &= (y_{p_x}, y_{p_y}, y_{v_x}, y_{v_y}), \\
c_g &\equiv (y(t) = (p_x, p_y, v_x, v_y) \wedge \sin(p_x) = p_y), \\
c_i &\equiv \exists (\tau_0, v_0) \in [0]\times[2]\times[5]\times[0]\times[-5]\ (y(\tau_0) = v_0), \\
c_c &\equiv \forall \tau \in \mathcal{D}_t\ (\dot{y}(\tau) = (y_{v_x}(\tau), y_{v_y}(\tau), 0, -g - k \cdot y_{v_y}(\tau))).
\end{aligned}
$$

Table 5.2 shows the results of solving the HCSs for the first five steps. We also solved the HCSs with the proposed method that did not apply the interval Newton method at line 6 of the algorithms HCSReviseL (Figure 5.4) and HCSReviseU. The computation results are shown in the lower rows. Each row corresponds to a solution of an HCS. Each column shows:

- The step number of the execution.
- The resulting time domain I_0.
- The number of times Branch was called.
- The number of times HCSRevise was called.
- The execution time in milliseconds of the simulation (for the whole steps from 1 to n).

Table 5.2 Computation results for bouncing particle model.

	(a) proposed method			
n	result (I_0)	branch	revise	time (ms)
1	$0.56636310070[488, 589]$	0	2	40
2	$1.51931342141[670, 914]$	1	28	120
3	$2.6883363074[045, 587]$	0	54	210
4	$3.333749635[478, 754]$	1	73	280
5	$4.33428886[471, 587]$	0	81	320
	(b) proposed method (without interval Newton narrowing)			
1	$0.56636310070[400, 534]$	1	158	360
2	$1.5193134214[155, 205]$	1	301	700
3	$2.688336307[375, 478]$	1	500	1130
4	$3.333749635[347, 843]$	1	648	1490
5	$4.33428886[412, 619]$	1	795	1840

The widths of the first results were around 10^{-12} (determined as about 10 times the h_{min} parameter). The subsequent results widened as the initial domains computed from the previous HCSs widened.

From the first HCS solution and the reset function described in the HA, a bounce of the particle was computed to set up the initial values for the next phase (we computed by the natural interval extension of the reset function). Accordingly, we had another HCS for the second bounce. For the second HCS, there are two theoretical solutions because the initial state also satisfies the guard constraint. In this case, we computed the enclosure for the second earliest solution by ignoring the initial state as described in Section 5.4.4. The simulation for the following steps of the execution were computed by the same procedure.

Figure 5.6 illustrates the boxes enclosing trajectories of the particle bouncing off the surface three times. These boxes were computed while solving three HCSs, each of which correspond to the parabolic motion of the particle (dashed-boxes in the figure represents results for the first and third continuous phase).

In the solving process for the first HCS, we confirmed that the time domain I_0 was quadratically reduced by the interval Newton operator. For example, in the solution for the first bounce, we had

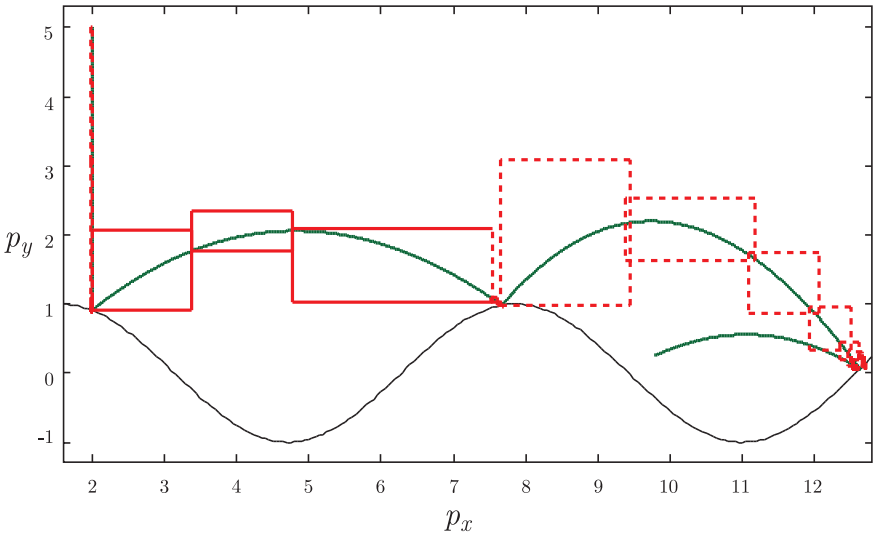

Figure 5.6 Trajectory of bouncing particle and interval enclosure of trajectory.

$$I_0^0 = [0.5366024006905468, 0.6462371408659776],$$
$$I_0^1 = [0.5655469230670141, 0.5675738062165443],$$
$$I_0^2 = [0.5663629650559695, 0.5663632653947441],$$
$$I_0^3 = [0.5663631007048800, 0.5663631007048839].$$

We confirmed that the numbers of revising the time domains and execution time were reduced by the interval Newton method as shown in Table 5.2. The reductions above provided the guarantee of the existence and uniqueness of a solution within the domain. Each of the results in the upper rows was guaranteed to contain a solution in this way.

5.6.2 Examples of Nonlinear HCSs

Here, we provide several HCSs that involve nonlinear constraints as continuous and guard constraints.

Example 24 We consider the problem of detecting the intersection of a trajectory following the continuous constraint described by the Van der Pol equation

$$
\begin{aligned}
y &= (y_1, y_2), \quad \mathcal{D}_t = [0, 100], \quad \mathcal{D} = \mathbb{R}^2, \\
c_i &\equiv (y(0) = (1, 0)), \\
c_c &\equiv \forall \tau \in \mathcal{D}_t \ (\dot{y}_1(\tau) = y_2(\tau) \wedge \dot{y}_2(\tau) = 10 \cdot (1 - y_1(\tau)^2) \cdot y_2(\tau) - y_1(\tau)),
\end{aligned}
$$

and the guard constraint described by an ellipse

$$
\begin{aligned}
\tilde{x} &= (t, x_1, x_2), \\
c_g &\equiv (y(t) = (x_1, x_2) \wedge \frac{x_1^2}{9} + \frac{x_2^2}{255} - 1 = 0).
\end{aligned}
$$

□

Example 25 The detection of the intersection of a trajectory following the continuous constraint described by the Lorenz equation

$$
\begin{aligned}
y &= (y_1, y_2, y_3), \quad \mathcal{D}_t = [0, 25], \quad \mathcal{D} = \mathbb{R}^3, \\
c_i &\equiv (y(0) = (15, 15, 36)), \\
c_c &\equiv \forall \tau \in \mathcal{D}_t \ (\dot{y}_1(\tau) = 10 \cdot (y_2(\tau) - y_1(\tau)) \\
&\qquad \wedge \dot{y}_2(\tau) = y_1(\tau) \cdot (28 - y_3(\tau)) - y_2(\tau) \\
&\qquad \wedge \dot{y}_2(\tau) = y_2(\tau) - \frac{8}{3} \cdot y_3(\tau)),
\end{aligned}
$$

and the guard constraint described by a sphere

$$
\begin{aligned}
\tilde{x} &= (t, x_1, x_2, x_2), \\
c_g &\equiv (y(t) = (x_1, x_2, x_3) \wedge x_1^2 + x_2^2 + (x_3 - 28)^2 - 700 = 0).
\end{aligned}
$$

□

Example 26 An HCS derived from the verification of a circuit involving a tunnel diode [29].

$$
\begin{aligned}
\tilde{x} &= (t, vt, it), \quad y = (y_{vt}, y_{it}), \\
\mathcal{D}_t &= [0, 7 \cdot 10^{-9}], \quad \mathcal{D} = \mathbb{R}^2, \\
c_g &\equiv (y(t) = (vt, it) \wedge (vt - 0.35) \cdot (vt - 0.055)), \\
c_i &\equiv (y(0) = (0.35, 0.00002)), \\
c_c &\equiv \forall \tau \in \mathcal{D}_t \ (\dot{y}_{vt}(\tau) = (-(0.0692 \cdot y_{vt}(\tau)^3 - 0.0421 \cdot y_{vt}(\tau)^2 \\
&\qquad + 0.004 \cdot y_{vt}(\tau) - 8.9579 \cdot 10^{-4}) + y_{it}(\tau)) \cdot 10^{12} \\
&\qquad \wedge \dot{y}_{it}(\tau) = (-y_{vt}(\tau) - 200 \cdot y_{it}(\tau) + 0.3) \cdot 10^6).
\end{aligned}
$$

□

Table 5.3 Comparison of results (1).

	(a) proposed method			
problem	result (I_0)	branch	revise	time (ms)
(1) b1	$0.56636310070[488, 589]$	0	2	40
(2) b2	$1.51931342141[670, 914]$	1	28	120
(3) vdp	$10.412056185[399, 407]$	1	167	120
(4) lorenz	$10.097265[363, 417]$	1	440	560
	(b) proposed method (without interval Newton narrowing)			
(1) b1	$0.56636310070[400, 534]$	1	158	360
(2) b2	$1.5193134214[155, 205]$	1	301	700
(3) vdp	$10.412056185[398, 407]$	1	288	220
(4) lorenz	$10.097265[363, 417]$	1	674	870

Table 5.4 Comparison of results (2).

	(c) Mathematica (symbolic)			
problem	result (t)	time		
(1) b1	0.56636310070	73		
(2) b2	unsolvable	–		
(3) vdp	unsolvable	–		
(4) lorenz	unsolvable	–		
	(d) Mathematica (numeric)		(d') Mathematica (numeric)	
(1) b1	0.56636309967	4	0.56636310070	18
(2) b2	1.51931341936	4	1.51931342141	21
(3) vdp	10.41204598059	13	10.41205618538	343
(4) lorenz	10.09936434992	31	10.09726539120	1511

5.6.3 Comparison with Existing Methods

To evaluate the computational efficiency and to confirm the accuracy of the results, we used several solvers, including the proposed method, to solve the following HCS problems:

(1, 2) The first and second bounces of the bouncing particle (Section 5.6.1).
(3) The Van der Pol equation and an ellipse (Example 24).
(4) The Lorenz equation and the sphere (Example 25).

We solved the above problems with the following methods:

(a) The proposed method.
(b) The proposed method that does not apply the interval Newton method.
(c) The symbolic `DSolve` solver with the `Minimize` function in Mathematica 7.0 [82].
(d, d') The numerical `NDSolve` solver with the `EventLocator` option in Mathematica. We solved the problems with the default settings in (d), and by setting `WorkingPrecision` to 28 and `MaxSteps` to `Infinity` in (d').

Tables 5.2 and 5.3 report the computed (interval) values for the time variable t (represented by I_0 and t), the profiling results, and the execution time. As shown by the results for (a) and (b), the interval Newton method decreased the number of reductions and execution time. The results show that our method has several advantages over Mathematica:

- In the results (c), `DSolve` of Mathematica computed a rigorous solution but treated only the problem (1).
- The results (d) and (d') showed that our method solved HCSs more efficiently than the `NDSolve` method. Additionally, `NDSolve` uses approximation algorithms and cannot ensure the achieved accuracy of a result, whereas our method guarantees the accuracy of a result.
- Another advantage of our method is that we can give intervals to the initial values and the coefficients in constraints. `DSolve` and `NDSolve` do not handle ODEs with uncertain parameters.

5.6.4 Computation of Multiple Solutions

The computation results of Examples 24–26 are shown in Table 5.5. We here computed sets of boxes that enclose all the solutions within the input boxes. We set the parameter w_{max} as 10^{-2}, 10^{-6}, and 10^{-11} for Examples 24, 25, and 26, respectively. Each column corresponds to:

- The computed time domain I_0 (we select the earliest box).
- The number of boxes computed as results.
- The number of boxes that are guaranteed to contain a solution.
- The number of calls to Branch and HCSRevise.
- The execution time.

The upper and lower part of the table show the results with and without applying the interval Newton method as in the previous sections.

Although the numbers of branches in the upper half were less than the results in the lower half, it did not affect the improvement of the execution time. In the

Table 5.5 Results of computing all solutions.

	(a) proposed method					
problem	result (I_0)	boxes	ex.	branch	revise	time (ms)
vdp	$10.412056185[396, 409]$	24	16	35	610	19033
lorenz	$10.097265[362, 418]$	2	2	5	418	10713
diode	$[0, 1] \cdot 10^{-12}$	2	1	1	21	3473
	(b) proposed method (without interval Newton narrowing)					
vdp	$10.412056185[363, 443]$	23	–	34	3695	21920
lorenz	$10.097258[021, 943]$	32	–	31	1061	10383
diode	$[0, 9.29] \cdot 10^{-13}$	2	–	1	54	3346

computation, the solving process for ODEs was expensive (it took many iterations with small time intervals). The solving process with the interval Newton method became further expensive when the number of calls to the ODE solver and number of cached box enclosures of trajectories increased. We consider that our implementation that caches the results by the ODE solver can be improved to accelerate the solving processes.

For the first problem, we had 24 boxes in the upper results and 23 boxes in the lower results. The method with the interval Newton method guaranteed that 16 boxes out of 24 boxes contained unique solutions. However, the method outputed more redundant boxes than the other method because the results by the ODE solver became less accurate with our caching mechanism.

For the third problem, the interval Newton method guaranteed the existence of a solution only for a box, although the other box also contained a solution. It failed because the solution existed at time $t = 0$, and the trajectory was not defined for $t < 0$, even though it was required to guarantee the existence of the solution.

Chapter 6

Bounded Reachability Analysis of Hybrid Systems

This chapter is intended to construct a model checking framework for hybrid systems described by HA with unsafe regions. It computes the reachable region of a model, and verifies the reachability to the unsafe states. Model checkers for hybrid systems such as References [43, 67, 29] have difficulties in verification, especially when the models belong to the class of nonlinear hybrid systems, where vector fields in the continuous state space or conditions for discrete changes are expressed by nonlinear constraints. Since most of the existing tools take linear hybrid systems as inputs, users need to linearize a problem by hand for each instance.

We propose a satisfiability modulo theories (SMT) framework for the bounded model checking (BMC) of nonlinear hybrid systems. In the framework, the bounded length of executions of a model is described by a predicate logic formula involving arithmetic constraints [4, 28, 23, 37]. Checking the satisfiability of the formula corresponds to the reachability analysis of the model (in this sense, we call this computation *bounded reachability analysis*). The computation may become possible for systems that are too large for unbounded execution. An SMT solver enumerates propositional models of the formula using a SAT (propositional satisfiability) solver and then checks the consistency of these models by calling *theory solvers* that handle the conjunctions of arithmetic constraints. BMC for possibly nonlinear hybrid systems is simply encoded using formulas involving ODEs [23]. However, there have been only a few SMT-based implementations that support nonlinear hybrid systems (e.g., by Bu et al. [11]).

More specifically, this chapter presents a framework for the bounded reachability analysis of HA involving nonlinear constraints.

- In the framework, an HA is encoded into a predicate logic formula involving constraints described in Section 5, namely, instantaneous, continuous, and guard constraints. We describe a phase of continuous changes between two discrete changes as an HCS.
- We propose a set of algorithms for checking the satisfiability of the encoded formula (Section 6.3). The algorithms work tightly with (i) a SAT solver that enumerates possible sets of constraints, and (ii) a theory solver based on the HCS solver that simulates a phase of continuous change. Basically, the SAT solver enumerates a candidate set of constraints of which the satisfiability is checked using the theory solver. The theory solver helps the SAT solver decide whether a discrete change should occur, since the decision is confirmed by simulating continuous changes that may cause the discrete change. The theory solver often restarts a simulation to refine the accuracy of over-approximation by dividing an initial interval value. In our method, the SAT solver manages the tree of divided initial values.
- In the proposed algorithms, the theory solver efficiently computes a set of boxes that enclose a counter-example by using the interval Newton method. Ordinary over-approximation methods do not necessarily guarantee that a computed enclosure contains a counter-example. In contrast, the HCS solver we adopt guarantees the existence of a unique solution in a result when the checking of certain conditions succeeds. This work focuses on the search of such sets of boxes in which a unique counter-example exists. When the algorithms fail to find such sets of boxes, we can still prove that the model has no counter-example by exhaustively searching the rest of the state space. When a check certainty succeeds, useful results can be obtained, such as "an over-approximation containing a unique counter-example," and "an initial box certainly reaching the unsafe region."

We have implemented the proposed method as a tool called `hydlogic` (Section 6.4), and used it to analyze several examples including those with nonlinear constraints (Section 6.5).

6.1 Constraint-based Representation of Hybrid Systems

We describe hybrid systems with unsafe states as predicate logic formulas that involve constraints in HCSs (Section 5.2). We explain how a reachability problem of an HA is translated into satisfiability checking of a formula. This encoding method is a modification of the former methods [4, 23].

6.1.1 Encoding RTTSs

Before describing how we encode HA, we present how to encode a k-step execution of RTTSs (Definition 16) into a logic formula. Such an execution is of the form

$$s^0 \xrightarrow{t^1} s^1_{\_} \xrightarrow{0} s^1 \xrightarrow{t^2} \cdots \xrightarrow{t^k} s^k_{\_} \xrightarrow{0} s^k, \tag{6.1}$$

where, for $i \in \{1, \ldots, k\}$,

- $s^0, s^i, s^i_{\_} \in \mathcal{S}$ and $t^i \in \mathbb{R}_{>0}$,
- $(s^i_{\_} \xrightarrow{0} s^i) \in \mathcal{T}$ corresponds to a discrete change, and
- $(s^{i-1} \xrightarrow{t^i} s^i) \in \mathcal{T}$ corresponds to a continuous change.

In the following, we call the pair of adjacent transitions $s^{i-1} \xrightarrow{t^i} s^i_{\_} \xrightarrow{0} s^i$ the *i-th step* of an execution.

Definition 32 A k-step execution of an RTTS $S = \langle \mathcal{S}, \mathcal{T}, \mathcal{S}_0 \rangle$ is encoded into a formula $[\![S]\!]^k$ as follows:

1. Prepare the following variables:
   - $(k+1)$ Boolean variables b^i_s $(i \in \{0, \ldots, k\})$ for each state $s \in \mathcal{S}$ representing whether the state is activated in the i-th step.
   - k variables t^i over $\mathbb{R}_{>0}$ $(i \in \{1, \ldots, k\})$ representing the duration between each two discrete changes.
2. The following formula expresses that a unique state in $\mathcal{S}$ is activated in the i-th step
$$UQ^i = \bigotimes_{s \in \mathcal{S}} b^i_s,$$
where $\otimes$ means that exactly one of the arguments is true.
3. Let s, s', and $s'_{\_}$ be states in $\mathcal{S}$. We describe that whenever two variables b^{i-1}_s and $b^i_{s'}$ are true, there should be transitions $s \xrightarrow{t^i} s'_{\_} \xrightarrow{0} s'$ $(t^i \in \mathbb{R}_{>0})$
$$TRANS^i = \bigwedge_{s, s' \in \mathcal{S}} (b^{i-1}_s \wedge b^i_{s'}) \Rightarrow ((s \xrightarrow{t^i} s'_{\_}) \in \mathcal{T} \wedge (s'_{\_} \xrightarrow{0} s') \in \mathcal{T}).$$
Note that the encoding method for $(s \xrightarrow{t^i} s'_{\_}) \in \mathcal{T}$ and $(s'_{\_} \xrightarrow{0} s') \in \mathcal{T}$ is not detailed here. We later describe the encoding method for HA.

4. Finally, we describe the initial state, and conjunct all the above formulas.

$$[\![S]\!]^k = \bigvee_{s_0 \in \mathcal{S}_0} b_{s_0} \wedge UQ^0 \wedge \bigwedge_{i=1}^{k} (UQ^i \wedge TRANS^i).$$

□

Satisfiability of the formula translated from an RTTS corresponds to whether an execution exists that is consistent with the RTTS's specification.

Lemma 5 For an RTTS S, suppose $[\![S]\!]^k$ is a formula encoded as described above. If $[\![S]\!]^k$ is satisfiable by a valuation that enables the Boolean variables $b^0_{s^0}, b^1_{s^1}, \ldots, b^k_{s^k}$ corresponding to the states in $\mathcal{S}$, and assigns real numbers to the variables $t^1, \ldots, t^k$, then an execution exists in the form of Equation 6.1. If $[\![S]\!]^k$ is unsatisfiable, S permits no execution. □

Proof. Straightforward from Definition 16. ■

6.1.2 Encoding Method for HA

Here, we consider a k-step execution of an HA (Lemma 3) of the form

$$\langle q^0, x^0 \rangle \xrightarrow{t^1} \langle q^0, x^1_- \rangle \xrightarrow{0} \langle q^1, x^1 \rangle \xrightarrow{t^2} \cdots \xrightarrow{t^k} \langle q^{k-1}, x^k_- \rangle \xrightarrow{0} \langle q^k, x^k \rangle.$$

Each element in the above execution is specified as follows ($i \in \{1, \ldots, k\}$):

- $q^0, q^i \in \mathcal{Q}$, $x^0, x^i, x^i_- \in \mathbb{R}^n$, and $t^i \in \mathbb{R}_{>0}$.
- $\langle q^0, x^0 \rangle$ is implied by $(q^0, X') = \mathit{Init}$ and $x^0 \in X'$.
- For a discrete change $\langle q^{i-1}, x^i_- \rangle \xrightarrow{0} \langle q^i, x^i \rangle$, $(q^{i-1}, q^i) \in \mathcal{E}$, $\mathit{grd}_{(q^{i-1}, q^i)}(x^i_-) = 0$, $x^i = \mathit{rst}_{(q^{i-1}, q^i)}(x^i_-)$, and $x^i \in \mathit{Inv}_{q^i}$ should hold.
- For a continuous change $\langle q^{i-1}, x^{i-1} \rangle \xrightarrow{t^i} \langle q^{i-1}, x^i_- \rangle$, a discrete state q^{i-1} specifies a vector field $f_{q^{i-1}}$. Then, a continuous trajectory $\phi : [0, t^i] \to \mathbb{R}^n$ is determined with the ODE $\dot{\phi}(\tau) = f_{q^{i-1}}(\phi(\tau)) \wedge \phi(0) = x^{i-1}$. Since $f_{q^{i-1}}$ is Lipschitz continuous, a unique trajectory is determined by the IVP-ODE. For all $t \in (0, t^i)$, $\phi(t) \in \mathit{Inv}_{q^{i-1}}$ should hold. x^i_- is obtained as $\phi(t^i)$.

Now we describe how we can encode a k-step reachability to unsafe states of an HA, i.e., k-step unsafety (Definition 23), into a predicate logic formula. In

the encoded formula, we utilize the constraint $cnt(t, x_-)$ that corresponds to a continuous constraint c_c of HCSs (Definition 29), which is defined as follows:

$$cnt(t, x_-) \equiv \forall \tau \in [0, t_{max}]\ (\dot{y}(\tau) = f(y(\tau))) \wedge (y(t) = x_-),$$

where t, x_-, and y are variables ranging over $\mathbb{R}_{>0}$, $\mathbb{R}^n$, and $\mathbb{R}_{\geq 0} \to \mathbb{R}^n$, respectively, f is a function $\mathbb{R}^n \to \mathbb{R}^n$, and t_{max} is a constant in $\mathbb{R}_{>0}$.

Definition 33 (encoding method for HA) Given an HA $S = \langle \mathcal{Q}, \mathbb{R}^n, \mathcal{E}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{R}, Init \rangle$, a k-step execution of S is encoded as a formula $[\![S]\!]^k$ as follows:

1. Prepare the following variables:
   - (k+1) Boolean variables b_q^i ($i \in \{0, \ldots, k\}$) for each discrete state $q \in \mathcal{Q}$ representing whether the state is activated in the i-th step.
   - k Boolean variables b_e^i ($i \in \{1, \ldots, k\}$) for each discrete state transition $e \in \mathcal{E}$ representing the activation of the transition.
   - (k+1) variables x^i ($i \in \{0, \ldots, k\}$) and k variables x_-^i ($i \in \{1, \ldots, k\}$) over n-real vectors representing the continuous state after the i-th transition and before the i-th transition, respectively.
   - k variables t^i over $\mathbb{R}_{>0}$ ($i \in \{1, \ldots, k\}$) representing the time at which the i-th transition takes place.
   - k variables x_{inv}^i over n-dimensional real vectors and k variables t_{inv} over $\mathbb{R}_{\geq 0}$ ($i \in \{1, \ldots, k\}$).
2. The following formulas express that a unique discrete state is activated and a unique transition takes place in the i-th step

   $$UQ^i = \bigotimes_{q \in \mathcal{Q}} b_q^i, \quad UE^i = \bigotimes_{e \in \mathcal{E}} b_e^i,$$

   where $\otimes$ means that exactly one of the arguments is true.
3. Let q be a discrete state specified in $Init.1$. The initial state is described by the following formula

   $$INIT = b_q^0 \wedge x^0 \in Init.2.$$

4. Taking a discrete state transition $e = (q, q') \in \mathcal{E}$ at step i implies enabling the discrete states q at step $(i-1)$ and q' at step i. Moreover, the guard constraint should be satisfied by x_-^i, and the initial state x^i for the next step is determined by the reset function (checking of the invariant is described

in the next step). For transitions in $\mathcal{E}$, we describe the following formulas

$$EDGE^i = \bigwedge_{e=(q,q')\in\mathcal{E}} (b_e^i \Rightarrow (b_q^{i-1} \wedge b_{q'}^i)),$$

$$TRANS^i = EDGE^i \wedge \bigwedge_{e\in\mathcal{E}} (b_e^i \Rightarrow (grd_e(x_-^i) = 0 \wedge x^i = rst_e(x_-^i))).$$

5. The following formula expresses the i-th continuous state evolution that corresponds to the discrete state q determined in the $(i-1)$-th step

$$CONT^i = \bigwedge_{q\in Q} (b_q^{i-1} \Rightarrow cnt_q^i(t^i, x_-^i)).$$

The invariant in the i-th step is described through the following formula

$$INV^i = (cnt_q^i(t_{inv}^i, x_{inv}^i) \wedge 0 \le t_{inv}^i \le t^i \wedge x_{inv}^i \notin Inv_q) \Rightarrow \neg b_q^i.$$

6. Finally, conjunct all the formulas described above. We also express that the unsafety (to be falsified in model checking) holds. In this framework, unsafety properties are represented as discrete states $US \subseteq \mathcal{Q}$ in a model. For each variable b_{us}^i corresponding to $us \in US$, we express that us will be reached within the k-step execution

$$[\![S]\!]^k = INIT \wedge UQ^0 \wedge \bigwedge_{i=1}^{k} (UQ^i \wedge UE^i \wedge TRANS^i \wedge CONT^i \wedge INV^i) \wedge \bigvee_{i=1}^{k} \bigvee_{us\in US} b_{us}^i.$$

□

Lemma 6 For an HA, suppose $[\![S]\!]^k$ is a formula encoded for k steps as described above. If $[\![S]\!]^k$ is unsatisfiable, then S is safe for its k-step execution. □

Proof. The encoding method follows the translation scheme in Definition 20, and the conditions in Lemma 3. ■

6.1.3 Encoding Method for Tiny HCC Programs

We can also consider to encode executions of Tiny HCC programs. To compute an execution, we interpret the program for every step in the execution, while

enumerating qualitatively different executions [48]. Interval-based interpretation method for Tiny HCC based on the consistency techniques (Sections 3.3 and 5.4) and entailment checking (Section 3.2) is described in Reference [49]. Enumeration of different executions is done by the SAT solver as in the method for HA.

6.2 Basic Procedure of Proposed Method

In this section, we describe a *lazy theorem proving* method [19] that checks the safety of a model S of an HA with unsafe regions. The pseudo-algorithm is illustrated in Figure 6.1 showing the basic procedure of the method that checks the satisfiability of the formula $[\![S]\!]^k$. Input to the algorithm is a model S and a maximum number of steps $k \in \mathbb{N}$ to verify. The algorithm returns one of the following values:

- **sat** (satisfiable);
- **unsat** (unsatisfiable); or
- **unknown** indicating that it cannot be decided whether the formula is satisfiable or not (due to the too coarse initial condition).

When **sat** is returned, the existence of a counter-example, which signals the unsafety of the systems, is guaranteed.

At line 1, the model S is encoded into $[\![S]\!]^k$. When reading a predicate logic formula *lf*, the algorithm first translates *lf* into a propositional logic formula *bf* by mapping each constraint in *lf* to a propositional variable (these maps are preserved in a table), and then substitutes *bf* into P. A flag *uk* initialized at line 2 indicates whether the satisfiability of the formula is decidable or not. Then the SAT solver processes the proposition P and computes a valuation ν for the propositional variables within them (line 4). If there is no valuation, the algorithm terminates and returns **unknown** or **unsat**.

A valuation ν implies an initial domain $\mathcal{D}_{0,\nu}$ (i.e., a set of initial values) and a trace γ_ν (Definition 21) at line 7. Given an initial domain $\mathcal{D}_{0,\nu}$ and a trace γ_ν, we can compute an over-approximation Φ of the set of hybrid trajectories that are sketched by $\mathcal{D}_{0,\nu}$ and γ_ν, as described in Section 5.6.1. When the approximation is guaranteed to contain a unique trajectory, the procedure returns **sat** (line 9). When no trajectory exists with respect to $\mathcal{D}_{0,\nu}$ and γ_ν, the procedure proceeds to the next loop to deduce another trace (line 11). Otherwise, the procedure checks whether the initial domain can be refined, and applies the refinement (line 14) or not (line 16), before proceeding to the next loop.

Input: model S, maximum step k
Output: satisfiability $sat \in \{\mathbf{sat}, \mathbf{unsat}, \mathbf{unknown}\}$

```
1:  P := encode [[S]]^k
2:  uk := false
3:  loop
4:    ν := SOLVE(P)
5:    if ν = false then
6:      return uk ? unknown : unsat
7:    Φ := compute the set of hybrid trajectories φ conforming to the initial
      domain D_{0,ν} and trace γ_ν
8:    if Φ contains a unique solution then
9:      return sat
10:   else if Φ = ∅ then
11:     CONTINUE()
12:   else
13:     if D_{0,ν} can be refined then
14:       P := P ∧ REFINE(D_{0,ν}); CONTINUE()
15:     else
16:       uk := true; CONTINUE()
17: endloop
```

Figure 6.1 Basic procedure of bounded reachability analysis.

6.3 Algorithms for Checking the Satisfiability

In this section, we propose a set of algorithms that is equivalent to the procedure described in the previous section. The algorithms check the safety of a model S of an HA with unsafe regions, and return **sat**, **unsat**, or **unknown**.

In the algorithms, we use interval-based techniques to deduce the satisfiability of constraints in a formula by computing a set of boxes that may enclose the solution of the constraints. As in DPLL(T) [30], we tightly integrate a modern SAT solver and the HCS solver described in Section 5.5. Our method incrementally runs a SAT solver, for each step, to enumerate combinations of active constraints in a formula, e.g. the discrete state to enable in the current step, the continuous constraint in the discrete state, and the guard constraint for a possible transition from the current state. Then, the interval-based HCS solver computes an enclosure of states that cause the next discrete change. With this result, the algorithms check the consistency of the set of constraints for the current step.

As in the previous work [15, 67, 28, 23], we dynamically refine an over-approximation of continuous changes to obtain a more accurate enclosure. Refinements are done by splitting one of the components of an initial boxed value. The refined initial values are enumerated by the SAT solver. Refinements are guided by whether or not computed intervals are proved to enclose a unique solution, or whether or not an initial interval value is precise enough.

6.3.1 Incremental Solving

The IncSolve algorithm illustrated in Figure 6.2 checks the satisfiability of the formula $[\![S]\!]^k$. At lines 1–2, the algorithm translates S into $[\![S]\!]^k$ and reads the subformula $\mathit{INIT} \wedge \mathit{UQ}^0$ describing the initial states into the proposition database P (P is always modified by appending formulas).

In the loop starting from line 5, the algorithm incrementally checks the satisfiability of $[\![S]\!]^i$ for $i \in \{1, \ldots, k\}$. At line 6, the subformulas UQ^i, CONT^i, INV^i, and EDGE^i are read. Then the SAT solver processes the proposition P and computes a valuation for the propositional variables within them (lines 7–10). If there is no valuation, the algorithm terminates and returns **unknown** or **unsat**. Note that TRANS^i is not handled here. The algorithm returns **sat** if the current discrete state is unsafe (line 12).

The decision of a discrete state transition $e \in \mathcal{E}$ to take place is computed in the HcsPropag procedure described in the next section (line 14). For each $e \in \mathcal{E}$ from the state q^i, HcsPropag checks whether the transition will be enabled or not, and returns a triple $(\mathit{res}_e, B_e, q_e)$ that consists of:

- the result res_e of the check;
- a box enclosure B_e of reachable continuous states that is consistent with the guard constraint grd_e; and
- the next discrete state q_e to proceed to.

At lines 15–26, the algorithm analyzes the results.

- When the box B_e is guaranteed to contain a unique solution of grd_e, the algorithm learns an initial condition for the next step and proceeds to the next loop (line 17).
- When grd_e is unsatisfiable, the algorithm learns that it does not need to re-check this transition in the sequel (line 20). This is effective when the algorithm refines the initial domain and re-simulates the execution from the domain.
- Otherwise, grd_e may be satisfied or may not. Thus, the algorithm tries to refine the initial domain (see Section 6.3.3) at line 23. If it cannot be

Input: HA S, maximum step k
Output: satisfiability $sat \in \{\mathbf{sat}, \mathbf{unsat}, \mathbf{unknown}\}$

1: encode S to obtain $[\![S]\!]^k$
2: $P := INIT \wedge UQ^0$
3: $uk := false$
4: $i := 1$
5: **while** $1 \leq i \leq k$ **do**
6: $P := P \wedge (UQ^i \wedge UE^i \wedge CONT^i \wedge INV^i \wedge EDGE^i)$
7: $sat := \textsc{Solve}(P)$
8: **if** $\neg sat$ **then**
9: return uk **? unknown : unsat**
10: $(q^i, \mathcal{D}_{i-1}, cnt^i_{q^i}, Inv_q) :=$ collect true-valued constraints from P
11: **if** $q^i \in US$ **then**
12: return **sat**
13: $\mathcal{E}' := \{(q, q') \in \mathcal{E} \mid q = q^i\}$
14: $\{(res_e, B_e, q_e)\}_{e \in \mathcal{E}'} := \textsc{HcsPropag}(\mathcal{E}', \mathcal{D}_{i-1}, cnt^i_{q^i}, Inv_q)$
15: **for** $e \in \mathcal{E}'$ **do**
16: **if** $res_e = true$ **then**
17: $P := P \wedge ((q^{i+1} = q_e) \Rightarrow (\forall j \in \{1, \ldots, n\}\ ((x^{i+1}).j \in Rst_e(B_e).j)))$
18: **else**
19: **if** $B_e = \emptyset$ **then**
20: $P := P \wedge (\neg e)$
21: **else**
22: **if** $\neg$ (the initial domain is precise enough) **then**
23: $P := \textsc{Refine}(P); i := 1; \textsc{Continue}()$
24: **else**
25: $uk := true$; $P := P \wedge (\neg e)$
26: **endfor**
27: $i := i + 1$
28: **endwhile**
29: return uk **? unknown : unsat**

Figure 6.2 IncSolve algorithm.

refined, i.e., the initial domain is too coarse to divide, the algorithm turns on the flag uk.

If there is no possible transition, the algorithm returns **unknown** or **unsat** (line 29).

6.3.2 Propagation by Solving HCSs

The HcsPropag algorithm (Figure 6.3) computes a continuous state evolution simultaneously evaluating the guard constraints to determine the next transition to take place. This is done by constructing an HCS for each candidate transition, and solving it with the method described in Section 5. The procedure is equivalent to *theory propagation* in DPLL(T). The inputs consist of a set $\mathcal{E}$ of candidate transitions, an initial domain $\mathcal{D}_0$ that describes an initial constraint (denoted by $\mathcal{D}_{i-1}$ in IncSolve), a continuous constraint cnt_q, and an invariant Inv_q for the current step.

The destination state q_e and the guard constraint grd_e are given by a transition $e \in \mathcal{E}$ (line 3). At line 5, an initial domain is prepared by setting a maximum time interval beyond the initial time and the invariant box for the current state.

An HCS is solved at line 6 and a set of boxes are obtained as a result. Here, we regard the process of solving HCSs described in Section 5.4 as a contracting map $Solve_{HCS} : \mathbb{I}^{n+1} \to \mathcal{P}(\mathbb{I}^{n+1})$ that refines an input box into a set of boxes that are appropriate for box-consistent HCSs. A resulting set of boxes is analysed at lines 7–14. When a result is empty, the algorithm returns *true* and $\emptyset$ (line 8). As described in Section 5.4, the $Solve_{HCS}$ process may guarantee that a box contains a unique solution of the HCS. The algorithm returns *true* if the existence of a solution is guaranteed, or *false* otherwise.

6.3.3 Over-approximation Refinement

The Refine procedure called from IncSolve tries to refine an over-approximation by dividing the initial box and re-computing the over-approximation for each of the divided boxes. In a refinement, an initial box is divided along one of the components of the box (each time the component is changed in a round-robin manner). In our method, we have a parameter $w_{min} \in \mathbb{R}_{>0}$, and the width of component chosen above should be larger than w_{min}. When an initial box is refined, the solver learns an additional formula on the initial constraint. In the formula, we use Boolean variables id_i ($i \in \mathbb{N}$) which give an identifier to each initial box. Beforehand, we give id_0 to $INIT$ by adding the formula $id_0 \Leftrightarrow INIT$ to the proposition database P. Let $\mathcal{D}_0$ be an initial boxed value, and assume that $\mathcal{D}_0$ is divided into boxes $\mathcal{D}_{0,1}$ and $\mathcal{D}_{0,2}$. Then, we construct the following formula and add this to the solver.

Input: set $\mathcal{E}$ of transitions, initial domain $\mathcal{D}_0$, constraint cnt_q, invariant Inv_q
Output: set R of tuples $(res, B, q) \in \{true, false\} \times \mathbb{I}^n \times \mathcal{Q}$

1: $R := \emptyset$
2: **for** $e \in \mathcal{E}$ **do**
3: $(q_e, c_{g,e}) :=$ collect the destination state and the guard constraint of e
4: $c_i :=$ construct an instantaneous constraint with respect to $\mathcal{D}_0$
5: $B := (I_0, \ldots, I_n)$, where $I_0 = [0, t_{max}]$ and $I_1 \times \cdots \times I_n = Inv_q$
6: $BS := Solve_{HCS}(B)$, // see Section 5.4
where $HCS = \langle \tilde{x}, I_0 \times Inv_q, \{c_g\}, \langle y, \mathcal{M}(I_0, Inv_q), \{c_i, cnt_q\}\rangle\rangle$
7: **if** $BS = \emptyset$ **then**
8: $R := R \cup \{(false, \emptyset, q_e)\}$
9: **else**
10: **for** $B \in BS$ **do**
11: **if** B is proved to contain a solution **then**
12: $R := R \cup \{(true, B, q_e)\}$
13: **else**
14: $R := R \cup \{(false, B, q_e)\}$
15: **endfor**
16: return R

Figure 6.3 HcsPropag algorithm.

$$((id_0 \wedge q^{i+1}) \Rightarrow (id_1 \oplus id_2)) \wedge (\neg id_1 \vee \neg id_2) \\ \wedge (id_1 \Rightarrow (x^0 \in \mathcal{D}_{0,1})) \wedge (id_2 \Rightarrow (x^0 \in \mathcal{D}_{0,2})).$$

After a refinement, the Continue() command at line 23 of IncSolve restarts the solving loop from step $i = 1$. Accordingly, one of the identifiers id_1 and id_2 is selected, and the computation of refined over-approximation starts. Note that not id_1 and id_2 but id_0 may be selected because a search along a different path may be still on the way.

6.3.4 Example of Reachability Analysis

We describe how the proposed method verifies the hybrid system in Example 15. Here, we change the initial domain to $(p, \gamma, c) \in [-1, 0] \times [\pi/6, \pi/4] \times [0]$. Parameters are set as $r = 2$ and $\omega = \pi/4$. Figures 6.4 and 6.5 illustrate the computation.

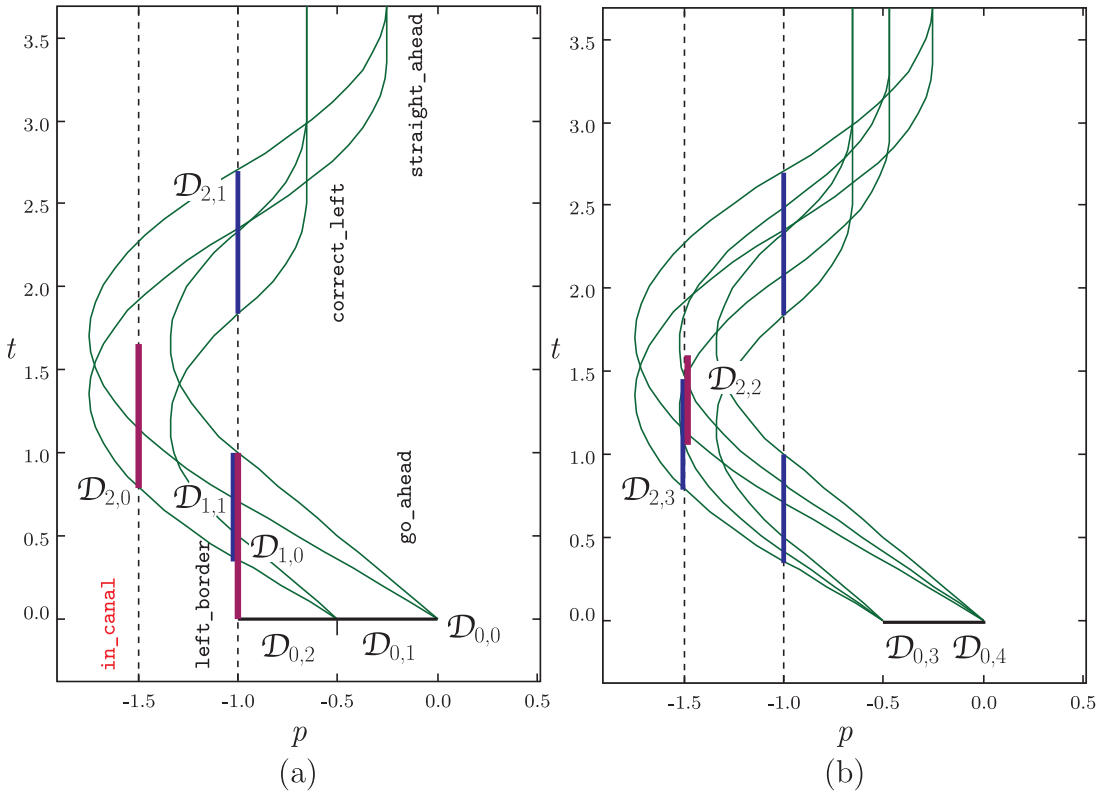

Figure 6.4 Process of solving car steering example.

- Figure 6.4 shows computed domains for p along the time line. The horizontal line at $t = 0$ denotes the initial domains for p. The vertical lines correspond to the boundary values for the discrete state transitions. Bending lines show the results of numerical computation using the boundary values of the initial domain.
- Figure 6.5 illustrates the enumeration of refined initial domains and decisions on transitions. Enumeration starts from the upper-left of the figure. A simulation from an initial domain proceeds horizontally as directed by the arrows. Each node is labeled by a tuple consisting of a result of a guard constraint evaluation, a domain satisfying the guard constraint, and the next state to transit. Refinements of the initial domains are shown by the identifier numbers directed by the dotted arrows.

The computation proceeds as follows:

a. In the computation for the first step ($i = 1$), the initial state `go_ahead`

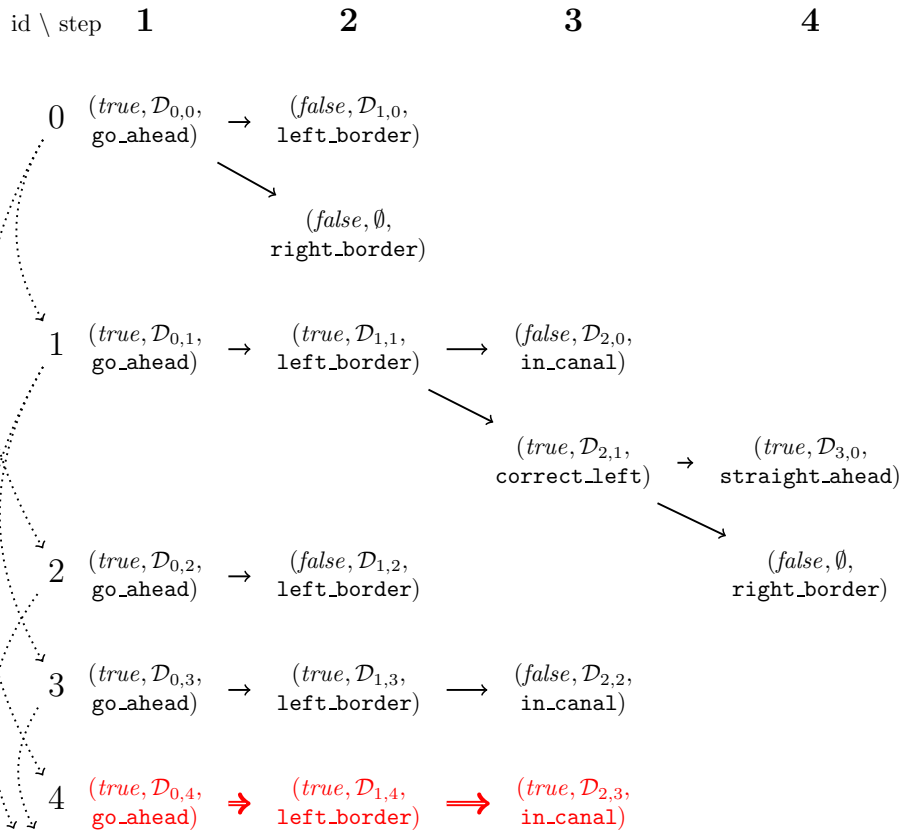

Figure 6.5 Enumeration of possible execution paths.

and the initial domain $\mathcal{D}_0 = ([0, t_{max}], [-1, 0], [\pi/6, \pi/4], [0])$ are activated at line 10 of IncSolve based on a result of the SAT solver (we denote $\mathcal{D}_0$ by $\mathcal{D}_{0,0}$ in the following, where an identifier for (refined) domains is subscripted). Then, HcsPropag constructs HCSs with respect to the state `go_ahead` and the transitions from `go_ahead` to `left_border` and `right_border`. By solving these HCSs, HcsPropag computes the results $\{(\mathit{false}, \mathcal{D}_{1,0}, \texttt{left\_border}), (\mathit{false}, \emptyset, \texttt{right\_border})\}$. Since all the results contain *false*, which means the existence of a solution is not proved, the algorithm refines $\mathcal{D}_{0,0}$ to $\mathcal{D}_{0,1}$ and $\mathcal{D}_{0,2}$ by dividing the component $[-1, 0]$ for p into $[-0.5, 0]$ and $[-1, -0.5]$.

b. Subsequently, from the refined initial domain $\mathcal{D}_{0,1}$, HcsPropag computes the result $\{(\mathit{true}, \mathcal{D}_{1,1}, \texttt{left\_border})\}$. IncSolve proceeds to the second step of the execution, and HcsPropag computes the results $\{(\mathit{false}, \mathcal{D}_{2,0}, \texttt{in\_canal}), (\mathit{true}, \mathcal{D}_{2,1}, \texttt{correct\_left})\}$ from $\mathcal{D}_{1,1}$. Since $\mathcal{D}_{2,0}$ is returned with *false*, $\mathcal{D}_{0,1}$ is refined into $\mathcal{D}_{0,3}$ and $\mathcal{D}_{0,4}$ along the

component for γ, i.e., $[\pi/6, \pi/4]$ into $[\pi/6, 5\pi/24]$ and $[5\pi/24, \pi/4]$ (Figure 6.4 (b) illustrates these domains). The algorithm also computes the execution path that moves into `correct_left` at step 3, and then reaches `straight_ahead` at step 4 of the execution.

c. From $\mathcal{D}_{0,2}$, the algorithm is still unable to decide whether the transition to `left_border` may be enabled or not. The algorithm applies another refinement.
d. From $\mathcal{D}_{0,3}$, the algorithm computes the domain $\mathcal{D}_{2,2}$ without the guarantee of existence, as in the computation from $\mathcal{D}_{0,1}$. Note that the execution path from $\mathcal{D}_{2,2}$ to `straight_ahead` is not computed here again, since the existence of this path was learned in Step b.
e. Finally, from $\mathcal{D}_{0,4}$, the algorithm computes a counter-example guaranteed to reach `in_canal` at step 3 of the execution.

6.4 Implementation

We built a prototype implementation called `hydlogic` of the method described in this chapter. `hydlogic` is implemented in OCaml, C, and C++, and consists of about 2000 lines of code. Inputs to `hydlogic` are a textual description of an HA, the maximum step k, and the threshold w_{min} for the refinement of initial domains. `hydlogic` translates an input model into a formula as explained in Section 6.1. Then, the core component checks the satisfiability of the formula by the method in Section 6.3. `hydlogic` integrates the following external solvers.

- The Decision Procedure Toolkit (DPT) [32] is used as an incremental SAT solver. DPT is an implementation of a DPLL-based SAT solver in OCaml. It has a flexible API for adding clauses incrementally and controlling search procedures.
- We use the implementation described in Section 5.5 for solving HCSs.

6.5 Experiments

We present experimental results for three examples. It shows how the complexity scales by the unrolled execution steps, the size of models, and the size of boxes given by initial constraints. We also compared the tool with `HSolver` [67] and `PHAVer` [29]. Note that the experimentation here were done on a 2.4GHz Intel Core 2 Duo processor with 4GB of RAM.

6.5.1 Car Steering Problem

Example 27 Consider the car steering problem given in Example 15. We first analyzed the unsafety of the model. `hydlogic` took 692.74 seconds and 1716 times of refinements to prove the existence of a counter-example. We set the minimum width w_{min} of the initial boxed values to 0.05, and the time domain I_0 in the HCSs to $[0, 3]$.

We then modified the guard constraint for the edge that enters `in_canal` as $p = 2$, and analyzed the model again. `hydlogic` returned the result **unknown** in 91.88 seconds. Refinements were done 48 times. The result was **unknown** because we could not prove the existence property for some of the guard constraints and the initial values. For example, when the initial domain for p was $[0, 0.05]$, the existence of a solution to the guard constraint of `left_border` was not proved. We confirmed that all the evaluation of the transition to `in_canal` had no solution.

We tried to solve the same instance of this problem by `HSolver` but the computation did not terminate after 10 minutes (though `HSolver` could solve another instance of the problem). □

6.5.2 Navigation Benchmark

Example 28 We present results for the navigation benchmark problem taken from References [29, 67]. This problem models an object at a position $(p_x(\tau), p_y(\tau)) \in \mathbb{R}^2$ moving within a grid of $n \times n$ areas of size 1×1 ($n \in \mathbb{N}$). Each area in the grid determines the velocity $(v_x(\tau), v_y(\tau))$ of the object as

$$(\dot{v_x}(\tau), \dot{v_y}(\tau))^T = A \cdot ((v_x(\tau), v_y(\tau))^T - v_d),$$

where

$$A = \begin{pmatrix} -1.2 & 0.1 \\ 0.1 & -1.2 \end{pmatrix}, \quad v_d = (\sin(i \cdot \pi/4), \cos(i \cdot \pi/4))^T.$$

$i \in \{0, \ldots, 7\}$ is determined by the each area. The lower left corner area of the grid has the coordinate $(0, 0)$. We used a map specified by the following matrix

$$M = \begin{pmatrix} \mathrm{U} & 2 & 4 \\ 4 & 3 & 4 \\ 2 & 2 & \mathrm{U} \end{pmatrix},$$

where U denotes an unsafe region, and the other numbers indicate the values i of the corresponding areas. We set the initial discrete state as the $(1, 2)$ cell in the grid, and set the initial values as $(p_{x,0}, p_{y,0}, v_{x,0}, v_{y,0}) \in [0, 1] \times [1, 2] \times [0.5] \times [0]$.

We analyzed the reachability to the unsafe region using `hydlogic`. Parameters were set as $k = 4, w_{min} = 0.25$, and $t_{max} = 10$. We proved the existence of an execution from the initial state, where $(p_{x,0}, p_{y,0}) \in [0.25, 0.5] \times [1.5, 1.75]$, to the unsafe region at $(3, 3)$. The analysis took 60.1 seconds and 34 refinements.

We then modified the initial condition for p_y to $p_{y,0} \in [1, 1.5]$, and tried to find an execution path reaching to the unsafe region at $(1, 1)$. We analyzed for $k \in \{2, \ldots, 7\}$ and each computation returned **unknown** because there were some guard constraints not guaranteed to be satisfied. We confirmed those guard constraints are not for the boundary of the unsafe region. The computation took 37.93, 40.33, 43.46, 70.37, 70.36, and 70.38 seconds, respectively.

`PHAVer` could check the safety of several instances of this problem [29]. An advantage of `hydlogic` is that it can prove the existence of an execution reaching the goals specified as unsafe areas. As previously experimented [67], `HSolver` could not solve this problem. □

6.5.3 Tunnel Diode Oscillator Circuit

Example 29 The third example taken from [29] models an RLC circuit involving a tunnel diode. The two-dimensional continuous state $(it, vt) \in \mathbb{R}^2$ expresses the current it through the inductor and the voltage drop vt of the tunnel diode. The vector fields are given by

$$\begin{pmatrix} \dot{vt}(\tau) \\ \dot{it}(\tau)) \end{pmatrix} = \begin{pmatrix} (-it_d \cdot vt(\tau) + it(\tau)) \cdot 10^{12} \\ (-vt(\tau) - 200 \cdot it(\tau) + 0.3) \cdot 10^6 \end{pmatrix},$$

where

$$it_d = \begin{cases} 6.0105 \cdot vt(\tau)^3 - 0.9917 \cdot vt(\tau)^2 + 0.0545 \cdot vt(\tau) \\ \qquad\qquad \text{if } vt(\tau) \leq 0.055 \ (q_0), \\ 0.0692 \cdot vt(\tau)^3 - 0.0421 \cdot vt(\tau)^2 + 0.004 \cdot vt(\tau) + 8.9579 \cdot 10^{-4} \\ \qquad\qquad \text{if } 0.055 \leq vt(\tau) \leq 0.35 \ (q_1), \\ 0.2634 \cdot vt(\tau)^3 - 0.2765 \cdot vt(\tau)^2 + 0.0968 \cdot vt(\tau) - 0.0112 \\ \qquad\qquad \text{if } 0.35 \leq vt(\tau) \ (q_2, q_3, q_3'). \end{cases}$$

We describe the model as an HA, where each discrete state corresponds to an equation for it_d. In the experimentation, we unrolled the model for 7 steps and tried to find an execution. We set the initial constraint as taking q_3 for the discrete state, and $(it, vt) \in [0.0006] \times [0.45]$ for the continuous state. `hydlogic` computed an enclosure of an execution with the guarantee of the existence. It took 4332

seconds. Most of the cputime was spent on the computation by `VNODE-LP` to solve the ODEs because `VNODE-LP` could only take small time steps (around 10^{-8}) in its iterative computation.

Reference [29] reported that `PHAVer` took a model, which was linearized beforehand, and proved that the continuous state stayed within a certain region. `HSolver` also solved a reachability problem based on this example in a reasonable time [67]. Our method might solve reachable sets more efficiently by applying recent techniques for solving ODEs with uncertain initial domains. By detecting that a box enclosure of a state in an execution is included in the initial domain, we can verify the safety for the infinite steps. □

Chapter 7

Related Work

In this chapter, we first overview the existing work on the reliable detection of discrete changes in comparison with our interval-based technique for solving HCSs (Section 7.1). Next, we describe related work on the modeling frameworks for hybrid systems (Section 7.2). Finally, we discuss methods for reachability analysis (Section 7.3).

7.1 Detection of Discrete Changes

As pointed out by Carloni et al. [12], simulators for hybrid systems based on numerical methods often compute qualitatively wrong results even for simple models such as the bouncing particle. We contend that interval-based techniques that guarantee the existence of a solution provide a reliable framework for the problems.

Several techniques have been proposed that tackle difficulties in numerical computation [14, 10, 64, 24]. Most of these techniques are based on the *discontinuity locking* approach, which is an iterative method as follows. First, a system is integrated by "locking" the definition of the derivative. Second, after each step of the integration, it is determined whether the result signals an event occurrence with respect to a guard constraint. If so, the exact time of occurrence is searched (e.g., by an interpolation procedure). These numerical techniques are aimed at a slightly different goal than our own. The techniques output floating-point numbers that seem not to be completely incorrect, but contain some errors, while our technique outputs a complete enclosure of a solution. Moreover, these techniques do not consider uncertainties in a problem.

Park and Barton [64] handled the *discontinuity sticking* problem, which was the problem of detecting the same discrete event right after a discrete change because of computation errors. They introduced into the detection process the

interval Newton method to guarantee the existence of a single event within a time interval. However, the guarantee existed only for interpolation polynomials reduced from the original systems.

More recently, Esposito and Kumar [24] proposed a robust technique based on *extrapolation polynomials* that could detect a discrete change occuring in the vicinity of a model singularity. Our method fails to handle these problems because the underlying ODE solver fails to guarantee the existence of a solution when the vector field is ill-defined.

Another study on the reliable detection of discrete changes was done in the context of model checking to over-approximate the reachable regions of models. We will discuss this work in Section 7.3.1.

7.2 Modeling Languages for Hybrid Systems

7.2.1 Relationship between HA and HCC

In Section 4, we have not shown the relationship between the two frameworks HA and (Tiny) HCC. An example that simulated an execution of an HA by an HCC program was shown [13]. However, the translation method used in the example did not simulate every execution of the HA (with respect to an uncertain initial value and non-determinism in the evaluation of guard constraints).

Falaschi [26] proposed a translation method from HCC into HA. Although the method was designed for generic HCC programs, the translation was only for bounded executions of HCC programs.

7.2.2 Constraint-based Languages

HCC, which was introduced in Section 4.3, was developed as an extension to the generic *concurrent constraint programming* framework [71]. Another generic language scheme of constraint programming is *constraint logic programming* (CLP). Hickey and Wittenberg [46] presented an approach using the CLP(F) language, which could describe analytic relations between real variables and functions. The implementation of CLP(F) supported interval arithmetic. However, the suppression of interval divergence in computing discrete changes was not considered.

7.3 Reachability Analysis of Hybrid Systems

Model checking techniques for hybrid systems have been developed diligently for nearly two decades [3, 76, 12]. Much of the existing work related more or less to

the reachability analysis of models. Such methods were well described as the CEGAR (counter-example guided abstraction refinement) framework [15] in which two procedures were integrated: abstraction of models (i.e., over-approximation of continuous states); and reasoning on the abstracted models.

In the following sections, we first overview methods that simulate the continuous behavior of models by computing the over-approximation (Section 7.3.1). Secondly, for an instance of the reasoning part of the model checking framework, we describe the existing SMT-based frameworks for verifying hybrid systems (Section 7.3.2).

7.3.1 Over-Approximation-based Simulation

Nedialkov and von Mohrenschildt [61] and Ramdani and Nedialkov [66] proposed interval-based methods for simulating hybrid systems. These methods have been developed on top of the interval-based VNODE method (Section 2.3), on which our method also depends. The former method [61] was limited in efficiency and the handling of nonlinear guard conditions. The successive work [66], which was recently proposed, handled HA that involved nonlinear constraints, including ODEs, invariant constraints, and guard constraints. This work presented a technique based on intervals and constraint programming to detect discrete changes with respect to continuous trajectories and guard (or invariant) constraints. However, they did not show the details on the constraint-based technique. Our technique (Section 5.4) adopts the interval Newton method, and has an advantage in its efficiency and guarantee of the existence of a unique solution.

Various safety verification techniques that abstract continuous state space by over-approximations have been developed [3, 76]. Some of the techniques represented the state space by a set of boxes [78, 45, 67, 22], and other techniques represented the state space by polyhedra [75, 15, 29]. Most of these techniques were developed in the CEGAR-like frameworks that repeatedly refined over-approximations in the verification. Hereafter, we review other approaches that explicitly compute over-approximation in which accuracy is considered in some sense (e.g., the maximal width of a box).

Janssen et al. [50], Cruz [17], Lin and Stadtherr [53], Ramdani et al. [65], and others have applied constraint programming techniques to the interval-based solving of ODEs, which compute the over-approximation of the possible continuous trajectories. Their frameworks allowed the use of parameters in ODEs as well as the addition of various constraints, such as value-restriction constraints [17]. Although these frameworks did not handle constraints equivalent to the guard constraints in this study, the frameworks could be integrated with ours.

Polytopes provide efficient representations of the continuous states of hybrid

systems. For example, Girard et al. [31] proposed a method based on zonotopes, Sankaranarayanan et al. [70] proposed a method called template polyhedra, and Collins and Goldsztejn [16] proposed a hybrid method of intervals and polytopes to suppress the wrapping effect. Integration of interval-based and polytope-based methods is a future research topic.

In many of the above methods, models are described by HA. Discrete changes in HA are detected by intersecting a reachable region of continuous trajectories, an invariant constraint of a discrete state, and a guard constraint. The computation is rather simple compared to the detection methods for the forward simulation of hybrid systems (e.g., the detection of the earliest discrete change described in Section 5.4.4).

7.3.2 SMT-based Bounded Model Checking

Fränzle and others [28] proposed an SMT framework that integrated SAT solving techniques and interval-based consistency techniques for real constraints. Their method brought useful approaches in SAT solving (e.g., conflict-driven learning and non-chronological backtracking) into the interval constraint solving. Integration of the SAT and interval-based solvers in our method is still limited, and could be improved to integrate more tightly.

Eggers et al. [23] used an interval-based solver for ODEs in the above SMT framework for the bounded model checking of HA. However, their method did not support either nonlinear ODEs or nonlinear guard constraints. Their method was also limited in its integration with the SMT framework. The method collected ODEs and solved them in a round-robin manner. Our method solves ODEs incrementally, while the SMT framework unrolls an execution path. To certify the result and reduce the search space, our method utilizes the existence property of a unique solution obtained by the interval-based solver.

Ratschan et al. [67, 22] proposed to translate a safety verification problem of a hybrid system into a constraint satisfaction problem. They also provided an interval-based implementation of the method that supported nonlinear constraints. Their method was not an SMT framework, but was based on a specific set of complex constraints. Our method provides a simpler and more modular SMT framework that uses generic solvers for (nonlinear) equations and ODEs.

Also, other methods were proposed including a method based on quantifier elimination techniques to verify hybrid systems described by *inductive invariants* [37], and a method based on *convex programming* that addressed the bounded model checking of a class of nonlinear HA [11].

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The main contribution of this thesis is that we have provided a reliable framework for the simulation and verification of nonlinear hybrid systems (Chapter 4) based on interval analysis (Chapter 2) and numerical constraint programming (Chapter 3).

HCSs in Chapter 5 formalize the problem of detecting discrete changes in the executions of hybrid systems, based on constraints whose domains are real numbers and trajectories over time. We revealed some relations between RCSs, CCSs, and HCSs by defining CCSs and HCSs based on the definition of RCSs. We also extended the box-consistency notion that approximates the solutions of RCSs for applying to HCSs. This formalism provides a basis for the discussion on the modeling languages in Chapter 4 and 6.

Our proposed technique for solving HCSs efficiently computes tight and validated box enclosures of solutions. The interval Newton method we applied accelerates the reduction of input box enclosures. This method may provide a guarantee of the existence of a solution within an enclosure of the solution, which helps enable the reliable analysis of hybrid systems in Chapter 6.

In Chapter 4, we described the semantics of two modeling languages HA and Tiny HCC. We provided a scheme for translating models in these languages into RTTSs, and also sketched executions of the models as hybrid trajectories.

Based on the contributions above, we have developed a framework for the bounded model checking and the reliable simulation of nonlinear hybrid systems (Chapter 6). The `hydlogic` tool takes an HA as an input and analyzes the bounded reachability of the input model. Computation by the tool is based on an SMT approach that translates a model into a predicate logic formula involving constraints of HCSs. The `hydlogic` tool enumerates the conjunctions of HCSs

that is deduced from the original formula, and examines the satisfiability by our interval-based solver for HCSs.

8.2 Future Work

8.2.1 Generalization of HCSs

HCSs can be generalized by handling continuous evolutions not only over time, but also over multidimensional axes. Interesting applications of this approach include systems biology. Although our problems are described as initial value problems, we could also consider problems that involve multiple boundary conditions.

Our technique for solving HCSs can be improved in various ways. Since our technique handles only small uncertainties in initial boxed values, more efficient methods for enclosing continuous trajectories are needed. A possible approach is to overcome the *wrapping effect* in computation by improving the the representation of uncertain values (e.g., by polytopes). There have been recent developments in solving ODEs with uncertainties (e.g., References [53, 65]), which we could adopt them as solvers for CCSs.

8.2.2 Development of the Modeling Languages

Translations between HA and HCC programs, as well as bisimulation equivalence between (the subsets of) the two could be discussed. Our method handles Tiny HCC that contains a minimal set of HCC constructs, while HCC has various extensions such as variable hiding, and default constraints [40]. Recently, as a successor of HCC, the HydLa language has been proposed [79]. HydLa supports *constraint hierarchies* to facilitate the construction of *well-defined* models (e.g., not over- or under-constrained), which we have found is often difficult in the modeling.

8.2.3 Towards More Powerful Tools

The procedure of bounded model checking methods coincides with mathematical induction by showing the existence of bounded executions (i.e., the base case), and then proving the existence of a loop in the executions. To detect such a loop by the `hydlogic` tool, a model should exhibit strong contraction of reachable regions. We consider that some transformations on the models are needed, so that a broader class of problems can be regarded as contracting systems.

The implementation of `hydlogic` could be improved in various ways. For ex-

ample, the interval-based constraint solving can be accelerated by propagating reduced intervals between the different computation steps in iterative reasoning [28]. We can also improve the refinement method by utilizing backward-reachability computation from the unsafe region. Another direction is to combine multiple theory solvers for various classes of numeric constraints. For example, combining techniques such as linear programming and formula manipulation with interval-based solvers will be a reasonable approach to many problems.

Bibliography

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

[2] R. Alur, T. Dang, J. Esposito, R. Alur, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In *Proc. of the First International Workshop on Embedded Software*, *LNCS* 2211, pages 14–31. Springer-Verlag, 2001.

[3] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. In *Proc. of the IEEE*, volume 88, pages 971–984, 2000.

[4] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119(2):17–32, 2005.

[5] F. Benhamou. Interval constraint logic programming. In *Selected Papers from Constraint Programming: Basics and Trends*, *LNCS* 910, pages 1–21. Springer-Verlag, 1994.

[6] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising hull and box consistency. In *Proc. of the 16th International Conference on Logic Programming (ICLP'99)*, pages 230–244, 1999.

[7] F. Benhamou and L. Granvilliers. Continuous and interval constraints. *Handbook of Constraint Programming*, pages 571–604, 2006.

[8] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(intervals) revisited. In *Proc. of the 1994 International Symposium on Logic Programming*, pages 124–138. MIT Press, 1994.

[9] F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[10] L. G. Birta, T. I. Oren, and D. L. Kettenis. A robust procedure for discontinuity handling in continuous system simulation. *Transactions of the Society for Computer Simulation International*, 2(3):189–205, 1985.

[11] L. Bu, J. Zhao, and X. Li. Path-oriented reachability verification of a class of nonlinear hybrid automata using convex programming. In *Proc. of Verifica-*

tion, Model Checking, and Abstract Interpretation (VMCAI'10), *LNCS* 5944, pages 78–94. Springer-Verlag, 2010.

[12] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1:1–193, 2006.

[13] B. Carlson and V. Gupta. The hcc programmer's manual. http://www-cs-students.stanford.edu/~vgupta/hcc/papers.html, 1997.

[14] M. B. Carver. Efficient integration over discontinuities in ordinary differential equation simulations. *Mathematics and Computers in Simulation*, XX:190–196, 1978.

[15] E. Clarke, A. Fehnker, Z. Han, B. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, *LNCS* 2619, pages 192–207. Springer-Verlag, 2003.

[16] P. Collins and A. Goldsztejn. The reach-and-evolve algorithm for reachability analysis of nonlinear dynamical systems. In *Proc. of the second Workshop on Reachability Problems*, volume 223 of *Electronic Notes in Theoretical Computer Science*, pages 87–102, 2008.

[17] J. Cruz. *Constraint Reasoning for Differential Models*. IOS Press, 2005.

[18] P. J. L. Cuijpers and M. A. Reniers. Lost in translation: Hybrid-time flows vs real-time transitions. In *Proc. of the 11th International Conference on Hybrid Systems: Computation and Control (HSCC'08)*, *LNCS* 4981, pages 116–129. Springer-Verlag, 2008.

[19] L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. of the 18th International Conference on Automated Deduction*, volume 2392, pages 438–455. Springer-Verlag, 2002.

[20] R. Dechter. *Constraint Processing*. Elsevier, 2003.

[21] Dynasim AB. Dymola. http://www.dynasim.se/.

[22] T. Dzetkulic and S. Ratschan. How to capture hybrid systems evolution into slices of parallel hyperplanes. In *Preprints of the third IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, pages 274–279, 2009.

[23] A. Eggers, M. Fränzle, and C. Herde. SAT modulo ODE: A direct SAT approach to hybrid systems. In *Proc. of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, *LNCS* 5311, pages 171–185. Springer-Verlag, 2008.

[24] J. M. Esposito and V. Kumar. A state event detection algorithm for numerically simulating hybrid systems with model singularities. *ACM Transactions on Modeling and Computer Simulation*, 17(1):1–22, 2007.

[25] D. Eveillard, D. Ropers, H. de Jong, C. Branlant, and A. Bockmayr. A multi-scale constraint programming model of alternative splicing regulation. *Theoretical Computer Science (Computational Systems Biology)*, 325(1), 2004.

[26] M. Falaschi, A. Policriti, and A. Villanueva. Time limited model checking. In *Proc. of International Workshop on Specification Analysis and Validation for Emerging Technologies in Computational Logic (SAVE'01)*, 2001.

[27] M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2006.

[28] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[29] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(3):263–279, 2008.

[30] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. of the 16th International Conference on Computer-Aided Verification (CAV'04)*, *LNCS* 3114, pages 175–188. Springer-Verlag, 2004.

[31] A. Girard and C. L. Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *Proc. of the 11th International Conference on Hybrid Systems: Computation and Control (HSCC'08)*, *LNCS* 4981, pages 215–228. Springer-Verlag, 2008.

[32] A. Goel and J. Grundy. Decision Procedure Toolkit (version 1.2). http://dpt.sourceforge.net/, 2008.

[33] F. Goualard. Gaol: NOT Just Another Interval Library (version 2.0.2). http://sourceforge.net/projects/gaol/, 2006.

[34] L. Granvilliers. RealPaver. http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/.

[35] L. Granvilliers and F. Benhamou. RealPaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156, 2006.

[36] L. Granvilliers and V. Sorin. Elisa (version 1.0.4). http://sourceforge.net/projects/elisa/, 2005.

[37] S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *Proc. of the 20th International Conference on Computer Aided Verification(CAV'08)*, *LNCS* 5123, pages 190–203. Springer-Verlag, 2008.

[38] S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *Proc. of the International Conference on Computer-Aided*

Design (ICCAD'04), pages 210–217, 2004.

[39] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1-2):3–49, 1998.

[40] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. Bobrow. Programming in hybrid constraint languages. In *Hybrid Systems II*, *LNCS* 999, pages 226–251. Springer-Verlag, 1995.

[41] W. Hartong, L. Hedrich, and E. Barke. Model checking algorithms for analog verification. In *Proc. of the 39th annual Design Automation Conference*, pages 542–547. ACM, 2002.

[42] T. A. Henzinger. The theory of hybrid automata. *Verification of Digital and Hybrid Systems, NATO ASI Series F: Computer and Systems Sciences*, 170:265–292, 2000.

[43] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:110–122, 1997.

[44] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.

[45] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HyTech: hybrid systems analysis using interval numerical methods. In *Proc. of the third International Workshop on Hybrid Systems: Computation and Control (HSCC'00)*, *LNCS* 1790, pages 130–144. Springer-Verlag, 2000.

[46] T. J. Hickey and D. K. Wittenberg. Rigorous modeling of hybrid systems using interval arithmetic constraints. In *the 7th International Workshop on Hybrid Systems: Computation and Control (HSCC'04)*, *LNCS* 2993, pages 402–416. Springer-Verlag, 2004.

[47] IEEE standard for floating-point arithmetic. IEEE Std 754-2008, 2008.

[48] D. Ishii, K. Ueda, and H. Hosobe. A branching approach to the interval-based evaluation of ask constraints in Hybrid CCP. In *Proc. of CP'07 Doctoral Programme*, pages 49–54, 2007.

[49] D. Ishii, K. Ueda, and H. Hosobe. An interval-based approximation method for discrete changes in Hybrid cc. In *Trends in Constraint Programming (Post-Proceedings of the CP'06 Workshops)*, pages 245–255. ISTE, 2007.

[50] M. Janssen, P. Van Hentenryck, and Y. Deville. A constraint satisfaction approach for enclosing solutions to parametric ordinary differential equations. *SIAM Journal on Numerical Analysis*, 40(5):1896–1939, 2002.

[51] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer-Verlag, 2001.

[52] O. Knueppel, D. Husung, and C. Keil. PROFIL/BIAS (version 2.0.8). http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html, 2009.

[53] Y. Lin and M. A. Stadtherr. Fault detection in continuous-time systems with uncertain parameters. In *Proc. of the American Control Conference (ACC'07)*, pages 3216–3221, 2007.

[54] J. Lunze and F. Lamnabhi-Lagarrigue. *Handbook of Hybrid Systems Control: Theory, Tools, Applications*. Cambridge University Press, 2009.

[55] The MathWorks. MATLAB/Simulink/Stateflow. http://www.mathworks.com/products/simulink/.

[56] G. Melquiond, H. Brönnimann, and S. Pion. Boost interval arithmetic library (version 1.34.0). http://www.boost.org/doc/libs/1_34_0/libs/numeric/interval/doc/interval.htm, 2006.

[57] R. E. Moore. *Methods and applications of interval analysis*, volume 2 of *Studies in Applied Mathematics*. SIAM, 1979.

[58] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[59] N. S. Nedialkov. VNODE-LP: a validated solver for initial value problems in ordinary differential equations. Technical Report TR CAS-06-06-NN, McMaster University, 2006.

[60] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.

[61] N. S. Nedialkov and M. von Mohrenschildt. Rigorous simulation of hybrid dynamic systems with symbolic and interval methods. In *Proc. of the American Control Conference (ACC'02)*, volume 1, pages 140–147, 2002.

[62] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

[63] G. J. Pappas. Bisimilar linear systems. *Automatica*, 39(12), 2003.

[64] T. Park and P. I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation*, 6(2):137–165, 1996.

[65] N. Ramdani, N. Meslem, and Y. Candau. Reachability of uncertain nonlinear systems using a nonlinear hybridization. In *Proc. of the 11th International Conference on Hybrid Systems: Computation and Control (HSCC'08)*, *LNCS* 4981, pages 415–428. Springer-Verlag, 2008.

[66] N. Ramdani and N. S. Nedialkov. Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint propagation techniques. In *Preprints of the third IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, pages 156–161, 2009.

[67] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Transactions on Embedded*

Computing Systems (TECS), 6(1)(8), 2007.

[68] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier, 2006.

[69] S. M. Rump. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*, pages 77–104. Kluwer, 1999.

[70] S. Sankaranarayanan, F. Ivancic, and T. Dang. Symbolic model checking of hybrid systems using template polyhedra. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, *LNCS* 4963, pages 188–202. Springer-Verlag, 2008.

[71] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91)*, pages 333–352, 1991.

[72] A. J. van der Schaft. Bisimulation of dynamical systems. In *Proc. of the 7th International Workshop on Hybrid Systems: Computation and Control (HSCC'04)*, *LNCS* 2993, pages 555–569. Springer-Verlag, 2004.

[73] A. J. van der Schaft and H. Schumacher. *An Introduction to Hybrid Dynamical Systems*, volume 251 of *Lecture Notes in Control and Information Sciences (LNCIS)*. Springer-Verlag, 2000.

[74] L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. Cyber-physical systems: a new frontier. *Machine learning in cyber trust: security, privacy, and reliability*, pages 3–14, 2009.

[75] B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan. Modeling and verification of hybrid dynamical system using checkmate. In *Proc. of the 4th International Conference on Automation of Mixed Processes*, pages 323—328. Shaker Verlag, 2000.

[76] B. I. Silva, O. Stursberg, B. H. Krogh, and S. Engell. An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In *Proc. of the 40th IEEE Conference on Decision and Control*, pages 2867–2874, 2001.

[77] O. Stauning. FADBAD++ (version 2.0). http://www.fadbad.com/fadbad.html, 2006.

[78] O. Stursberg, S. Kowalewski, I. Hoffmann, and J. Preusig. Comparing timed and hybrid automata as approximations of continuous systems. In *Hybrid Systems IV*, *LNCS* 1273, pages 361–377. Springer-Verlag, 1997.

[79] K. Ueda, D. Ishii, and H. Hosobe. A constraint-based modeling language for hybrid systems. In *Proc. of the fifth Symposium on System Verification*, pages 1–6, 2008. (in Japanese)

[80] P. van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Anal-*

ysis, 34(2):797–827, 1997.

[81] P. van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.

[82] Wolfram Research. Mathematica (version 7.0). http://www.wolfram.com/products/mathematica/index.html, 2009.