CLP(F) Modeling of Hybrid Systems

A Dissertation

Presented to

The Faculty of the Graduate School of Arts and Sciences

Brandeis University

Department of Computer Science

Timothy J. Hickey, Advisor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

David Karger Wittenberg

May, 2004

This dissertation, directed and approved by David Karger Wittenberg's committee, has been accepted and approved by the Graduate Faculty of Brandeis University in partial fulfillment of the requirements for the degree of:

DOCTOR OF PHILOSOPHY

Dean of Arts and Sciences

Dissertation Committee:

Timothy J. Hickey, Dept. of Computer Science, Chair.

Mitch Cherniack, Computer Science

Martin Cohn, Computer Science

Pascal van Hentenryck, Brown University

Dedication

To my daughter, Ruth Meira Kagno Wittenberg, my wife, Cynthia Solov Kagno, and to the memory of my grandmother, Ruth Edith Budinoff Wittenberg.

Acknowledgments

My thanks to Tim Hickey for suggesting this line of inquiry, as well as for his tremendous support for every step along the way. He always seemed to have a barely makable conference deadline in mind, forcing me to write sections quickly, but he never chose a deadline I couldn't make. Getting just the right pressure is an art, and Tim excels at it.

I thank Alex Feinman, Seth Landsman, and Heather Quinn for their proofreading early versions of several sections, and their reminders about what had to be done. In particular, Alex was always available to give comments on short notice, and seemed to know the the answer to every technical problem I ran into.

The many people who shared office space with me helped to make solitary work less lonely. Alan Blair, José Castaño, Paul Darwen, Bob Knippen, John Langton, Jess Littman, Aaron Macks, Anna Rumshisky, and Roser Saurí Colomer all contributed to a friendly atmosphere. Peter Møller Neergaard helped me with some strange LaTeX problems.

Joan Bolker not only wrote the book on writing dissertations, she gave me a lot of good advice, and the occasional well aimed kick in the rear. Scott Buchanan not only kept my computer running, he also provided encouragement.

The Geezers and Neurons Ultimate frisbee group provided an excuse to get away from the computer screens occasionally.

The faculty of the Computer Science Department were always encouraging. I particularly thank Marty Cohn, Harry Mairson, and Jim Storer for their advice.

Myrna Fox and Jeanne DeBaie, the staff in the department made sure that all the strange paperwork happened, and, as importantly, they supplied cookies which were an important morale boost for us all.

None of this work would have been possible without the support of my wife, Cynthia Kagno and my daughter Ruth. They were remarkably accepting of my working ridiculous hours, and tolerated my disappearances to conferences and my office with equanimity.

ABSTRACT

CLP(F) Modeling of Hybrid Systems

A dissertation presented to the Faculty of
the Graduate School of Arts and Sciences of
Brandeis University, Waltham, Massachusetts

by David Karger Wittenberg

We wish to rigorously model hybrid systems. Because models of hybrid systems are often used in safety-critical applications, it is crucial to get accurate results with explicit limits on the errors in the model and calculations. This requires a rigorous approach. Any technique to model these systems which does not account for rounding error or error in the approximation to the solution of the ODEs governing the problem is not rigorous. We use CLP (Constraint Logic Programming) over interval arithmetic to provide an explicit limit on rounding errors and on the ODE solution errors.

Preface

This work grew out of several areas I have worked in. In the late 1980s, I worked on the verification of the VAX SVS[1] (Secure Virtual System) computing system, which was a trusted operating system designed to receive a National Center for Secure Computation (NCSC) A1 certification (A1 was the highest defined rating).

Later I did some work on the issues of determining probabilities of extremely rare events [Wit03] as an approach to trying to understand when one could believe that a program had a failure probability of 1 failure in 10^9 hours, as the FAA specified for programs controlling commercial aircraft. It wasn't clear to me then, and isn't now, that one can reasonably assert that any large system has a reliability that high. Hatton and Roberts's [HR94] work showing that very high quality numeric software introduces a cumulative numerical error of about 1% per 4000 lines of code contributed to my interest in calculating numerical answers correctly.

When I saw Hickey's CLIP [Hic00b] system which made it possible to soundly do calculations over the real numbers on standard hardware, I became interested in the question of how reliable numeric computation is, and whether we could combine techniques from program verification with CLIP to produce more reliably correct calculations over the reals.

This work may be considered as a start on a larger project of making a system to analyze hybrid systems which would be easy enough to use for engineers to make use of, and extremely likely to provide correct answers. This would be done by extending

[1]For a description of the SVS system, see Karger *et al.* [KZB$^+$90] and [KZB$^+$91]. The NCSC ratings of that time are described in the "Orange Book" [ora85].

CLIP's interface to make it easier to use, integrating the work we did on compiling constraint solvers with CLIP [HW99a], and then validating the resulting system.

Contents

List of Figures

Chapter 1

Introduction

This thesis addresses the problem of rigorously modeling hybrid systems. It involves combining several areas of research that have historically been separate. We present some practical models, including working code (with timings) and argue that this direction can be continued to provide much more confidence in the system's analysis than was previously possible.

The principle underlying this work is that studies of safety critical systems must be as rigorous as possible. From this comes the requirement to bound all measurements, parameters, and behaviours which can not be known exactly. Because all measurements have associated error-bars, this seems to require some sort of sensitivity analysis, or the ubiquitous use of interval arithmetic. We do not wish to use approximations unless we can rigorously bound the error that the approximation introduces. We use intervals ubiquitously. The idea behind interval arithmetic is to express every real value X as an interval $x = [x_{min}, x_{max}]$ with $x_{min} \leq x_{max}$.

1.1 Hybrid Systems

The area which we seek to improve is software to model or analyze hybrid systems. A hybrid system consists of a digital part (typically a small computer), and an analog part (the physical system which the computer interacts with.) The study of hybrid systems concentrates on systems used in safety critical applications such as air traffic control [TLS99] or automobile, and automated highway systems [LL98] control. The analysis tools for hybrid systems are also being used to analyze biological systems [BFH$^+$04, LT04] We present a short history of hybrid system work in Chapter 2. The hybrid systems we study have analog behaviour described by ODEs (Ordinary

Differential Equations).

An important feature in a hybrid system modeling technique is the ability to prove *safety properties* of a hybrid system. A safety property is a guarantee that if the system is started in a reasonable state, it never enters some prohibited region of the state space.

1.2 Requirements for Rigor

Because the systems we are interested in are safety-critical, have points of chaotic behaviour, or would be extremely expensive to repair, it is important that the results we get are correct. Ideally, one might prove the correctness of the programs, but that is beyond our current ability, and would still leave problems with round off. We make some effort to use techniques which are easily verified, though we have not yet formally verified any of these programs.

In discussing the correctness of programs which give numerical results, it is important to distinguish between *accuracy* and *precision.* Accuracy is a measure of how close the calculated result is to the correct result, while precision is a measure of how closely the answer is described. An extremely precise answer may be completely wrong, while an accurate answer must be at least as precise as it is accurate. Floating point calculations using the IEEE 754 floating point standard [IEE85] double format (64 bits total – the most commonly used) always give 53 bits of precision, but the results are often not nearly that accurate.

In order to be fully rigorous, one must account for all sources of error, or even uncertainty. There are several sources of error to deal with:

- measurement errors - sensors which give the wrong readings
- construction errors - parts not exactly the size specified
- calculation errors - see von Neumann's list in Section 3.1.2
- physics errors - uncertainties in the parameters of the equations for the system
- "messy physics" errors - areas where the behaviour is hard to model

The approach proposed here covers all of these error sources (except blunders in calculation) in a single elegant way.

1.3 Tools used

The tools we use are Constraint Logic Programming (CLP) [JM94], interval arithmetic [Moo66], and CLP(F) [Hic00a], a Constraint Logic Programming language, which combines CLP with interval arithmetic, as well as adding the capability to use ODEs directly in the description of a system. Logic programming is a relative of functional programming in that the programmer states characteristics of the results desired, rather than specifying a procedure to achieve them. This style often results in shorter, simpler programs which are easier to understand than procedural programs. Like many CLP languages, CLP(F) is an extension to Prolog. A trivial CLP(F) program to calculate the square root of 2 is:

```
| ?- {X^2=2,X>0}.
```

which returns

```
X = 1.41421356237309... ? ;

no

| ?-
```

Note that one specifies the desired result (a number X such that $X^2 = 2$) rather than a procedure for taking square roots. CLP(F) allows constraints on function variables as well as on interval variables. We discuss these tools at greater length in Chapter 3.

1.4 Contributions of this Thesis

The approach we propose is to model a hybrid system by a CLP(F) program in which the ODEs can be expressed directly as constraints on function variables. This will allow the program to directly represent the hybrid system in the sense that one can analyze the hybrid system by simply analyzing the corresponding program.

The major innovation in this thesis is using CLP(F) with its function variables and its ubiquitous intervals in modeling hybrid systems. This allows us to unify the handling of measurement errors (sensitivity analysis), round off errors, incomplete (in a limited way) specifications, and through techniques discussed in Section 6.3.4, points where the ODEs are non-analytic or even discontinuous, all while retaining a fully rigorous approach.

The innovations reported in this thesis are:

- Improved rigor in dealing with errors by using intervals ubiquitously in modeling hybrid systems
- Improved ease in describing hybrid systems by using a language in which ODEs can be expressed directly

- Improved rigor in modeling of hybrid systems which are described by non-analytic (or even discontinuous) ODEs.

- Rigorous modeling of hybrid systems whose specifications are incomplete or poorly understood by using a "bounding box" constraint which includes all possible behaviours.

- Improved ease of analysis. Because CLP allows constraints to be propagated in either direction, one can easily ask questions like "What parameters give a particular period" as well as modeling to find out what period a set of parameters produces.

Many earlier systems can solve some of these problems. The advantage gained by using CLP(F) is that all of them are dealt with in one system, and in a consistent, elegant way.

1.5 Organization of this Thesis

Because this thesis describes using a tool (CLP(F)) in an area (hybrid systems) in which it has not previously been used, I start with introductions to both areas. Chapter 2 provides an introduction to hybrid systems, and Chapter 3 provides an introduction to CLP(F), which is the primary tool I use. Chapter 3 also provides background on interval arithmetic and Constraint Logic Programming (CLP) which are combined in CLP(F).

The heart of the thesis is Chapter 4, Chapter 5, and Chapter 6 which are where all the new work is. Chapter 4 describes how I use CLP(F) to model hybrid systems by providing a rigorous model. Chapter 5 shows how to handle a more complex model, and how one can use CLP(F) to analyze the behaviour of that model (Section 5.3). It concludes with a few examples, and a discussion of the advantages CLP(F) has over other modeling techniques.

Chapter 6 first extends the model to more complex systems, and then introduces new techniques necessary for rigorously modeling the non-analytic and discontinuous ODEs in more complex systems.

Chapter 7 describes several ways in which this work can be extended.

Appendix A is a snapshot of a draft of a manual for clip and CLP(F). While it's rather incomplete, it's the best documentation available now.

Appendix B shows the CLP(F) code used to model the n-tanks problem for $n = 4$. This code is discussed in detail in Chapter 6.

1.6 Overview of Thesis

This section is a overview of the thesis, provided as both a short piece which one could read to get an idea of what this thesis covers, and an outline of the work to guide the reader of the whole work.

The major innovation in this thesis is a principle that rigor is necessary, and the use of clip to provide that rigor.

The principle underlying this work is that studies of safety critical systems must be as rigorous as possible. From this comes the requirement to bound all measurements, parameters, and behaviours which can not be known exactly, as well as errors introduced by calculations. The innovation in this thesis is to use CLP (Constraint

Logic Programming) and interval arithmetic together to model hybrid systems rigorously. Clip is a CLP language over functions on real intervals to use in this modeling. Using clip allows one to use a single technique to rigorously model all of the main sources of preventable error.

The sources of error which are bounded by clip include roundoff error from working with floating point numbers, tolerances in the building of equipment, error bars in sensor's measurements, uncertainty about the ODEs (Ordinary Differential Equations) used to describe the system, and poorly understood behaviour near boundary conditions.

Constraints are a natural technique to use when working with hybrid systems, both because safety properties are clearly expressed as constraints, and because the use of constraints allows one to rigorously model uncertainties in the system. Interval arithmetic fits naturally with CLP because an interval is essentially a pair of constraints, one constraint providing an upper bound, and the other constraint providing a lower bound. Interval arithmetic is a good choice for modeling physical systems because intervals are a natural way to describe measurements.

Much of the time, we use intervals implicitly. All of the parameters that we calculate with are implemented as intervals. Most of the time, our calculations look like a standard ODE calculation, but clip automatically uses intervals for all the parameters as well as for the coefficients in the Taylor expansions it uses to describe functions. In small areas where one can not use ODEs as a model (perhaps because the ODEs are non-analytic at some point, or near a boundary where the physics is poorly understood), we simply constrain values to be within a "box" chosen to be a reasonably small area in which we can be sure that the true value lies. This allows us to provide a rigorous analysis even where we can not rigorously use normal methods.

A nice feature of CLP languages is that they allow the user to specify any of the variables and have the system calculate constraints on the remaining variables. This means that the same program allows one to ask both "With these parameters what state will the program be in at time t?" and "What parameters are required for the program to be in state s at time t?" by simply plugging in different variables.

Chapter 2

Hybrid Systems

A hybrid system is defined as a system composed of a digital part (typically a small computer) and an analog part (typically a physical system with sensors and actuators). All computer controlled or monitored processes in the real world are hybrid systems. As a field of study, “Hybrid Systems” has come to include the study of the analog part of a system in an area where reliability is at a premium, typically because of the cost (in lives or money) of a failure of such a system. Some hybrid systems papers study only the analysis of the analog part of a system. Hybrid systems research grew out of real time computation, control theory, and program verification in order to prove properties such as stability about complex safety critical systems. The field of hybrid systems is the study of systems in which discrete events and continuous dynamic events interact. In the chemical engineering literature, hybrid systems are sometimes called “combined discrete/continuous processes”.

The history of hybrid systems starts with Fahrland’s 1970 paper [Fah70] which asked “Why limit the modeling to either discrete event or continuous when situations are evolving that require more interdisciplinary solutions”. Very little was done for the next twenty years, and Fahrland’s work is rarely cited. Fahrland may have been influenced by Roger Brockett who was also at Case Institute of Technology, and who later did some seminal work on hybrid systems. The first conference on the subject was the 1991 REX workshop titled Real Time: Theory in Practice [dBHdRR91] where the term “hybrid automata” was introduced. Since that time, real time systems and hybrid systems work has diverged, with real time work focusing more on the computer with its latency issues, and hybrid systems focusing more on accurate modeling of the analog part of the system. While it’s not clear how to put a real time model (explicit limits on the time for processing) in the standard formalism for hybrid automata, the techniques introduced in this thesis can easily model digital components as long as their latency can be specified.

A hybrid automaton [ACH+95, HHWT97, HHMWT00, Hen96, LSVW99, LSV01] is a formal model of a hybrid system. Most hybrid automata models are expanded finite state automata with two different forms of state transition. The first (a "jump" transition), is the standard state transition in a finite state automata, which takes zero time, and changes the state of the digital part of the system. The second (a "flow" transition), is the change in the analog part of the system over time, corresponding to the physical process occurring. There are many versions of hybrid automata, which differ in what they can model. Some earlier models required that the flow transitions be piece-wise linear. Many later models allow flow transitions described by ODEs (Ordinary Differential Equations), sometimes limited to linear ODEs.

There has been considerable research on developing formal models of hybrid systems. Among others, Davoren and Nerode developed logics [DN00], Maler *et al.* [MMP91], Lynch *et al.* [LSVW99, LSV01], Henzinger *et al.* [Hen96], and Alur *et al.* [ACH+95] developed formal models. From our point of view, a limitation of these models is the difficulty in applying them to real systems, and the amount of overhead that must be relied on to trust the results.

Another major research push has been in the development of practical tools for modeling and analyzing realistic hybrid systems. For example, Kowalewski *et al.* [KSF+99] describe eight such systems (Matlab, Simulink, gPROMS, Shift, Dymola, BASIP, SMV, HyTech) that can be applied to model and/or analyze fairly complex hybrid systems. All of these systems sacrifice some degree of rigor in order to handle moderately complex hybrid systems. For example, HyTech does not allow the physical system to be governed by ODEs (Ordinary Differential Equations), the other systems that do allow ODEs use some form of approximation that introduces errors which are then ignored. The simulation based systems rely on the user to correctly identify the worst case when there is a non-deterministic choice.

An important use of hybrid systems is to prove "safety properties", which are statements of the form "the water level in this tank never exceeds some value". Because safety properties are constraints, they fit naturally in a CLP approach.

2.1 Compendium

In this section I try to give a flavor of the current state of the art by listing at some examples used in Hybrid Computing papers (and a few examples used in real-time systems as hybrid automata were being developed).

- Cat and Mouse problem A trivial problem used pedagogically (Proposed at REX workshop [dBHdRR91]) [Woo91] using Z; [MMP91] [NOSY93] In a 1-dimensional model, a mouse starts running to its hole, and at a time Δ later, the cat starts chasing the mouse (starting from the same position). Both animals run at constant velocity. Does the cat catch the mouse? (It is possible for the answer to be indeterminate due to the granularity of time, or to the error-bars in the problem description.) Maler *et al.*[MMP91] use the following notation: The mouse's velocity is V_m, the cat's velocity V_c, the hole is distance X_0 from the start position. The cat's position is x_c and the mouse's position is x_m. If at some time, $x_c = x_m > 0$, the cat wins. If $x_m = 0$ before the cat wins, the mouse wins.

- Communications Time Out [HMP91]
- Traffic Lights [HMP91]
- Two Tanks — Kowalewski *et al.*[KSF+99] use 6 methods to model a fluid flow problem previously studied by Stursberg *et al.* [SKHP97] also used by Henzinger et al. [HHMWT00]
- Simple thermostat [ACHH93] - possibly from [NSY91]
- Thermostat with delay[HHMWT00] - This system is discussed in Chapter 4.2
- Water level monitor [ACHH93]
- Complex Thermostat [NOSY93] This version cools a reactor, using control rods which cause cooling, with a constraint on the minimum time between moves of each rod.
- Hot water heater (bathroom boiler) Ciarlini et al. [CF00]
- Billiards [NOSY93]
- timing based mutex with skewed clocks [ACHH93]
- leaking gas burner [ACHH93] (problem comes from Chaochen, Hoare, and Ravn - A calculus of durations, IFIP, 1991)
- Lots of different models of air traffic avoidance - Henzinger et al. [HHMWT00]
- Idle speed control of auto engine Balluchi et al. [BBB+00]
- Many biological systems [BFH+04, Neo04, LT04]

2.2 History

2.2.1 Pre-History

Nicollin and Sifakis [NS91] wrote an early survey of timed systems, which they admit is both incomplete and biased, but provides a basis for understanding the later work.

Real time computing has been around since the 1960s. The specification for a real-time system typically stated that any interrupt would be handled in some specified time. Hybrid automata come out of the real time computing work. Earlier models include Lamport's temporal logic of actions[Lam94], and Milner's CCS[Mil80] (Calculus of Communicating Systems). Various Temporal logics were used to describe what are now called hybrid systems. CCS was mainly used for describing distributed computing (with no analog component, and often with no time constraints.)

Much of the groundwork for Hybrid systems was laid at a REX (Research and Education in Concurrent Systems) workshop held in the Netherlands in June, 1991. The proceedings were published [dBHdRR91]. This led to a series of hybrid system conferences [GNRR93, AKNS95, AHS96, AKNS97, AKL+99] which were followed by a series of "Hybrid Systems: Computation and Control conferences" [SH98, VvS99, LK00, BSV01, TG02, MP03, AP04]

Maler *et al.*[MMP91] consider time only through an "age" function, which is the length of the time interval since the most recent change in the continuous state. While timed systems using temporal logic [Lam94, AL91] assumed that all actions were instantaneous, Maler *et al.* introduced the now standard model of hybrid automata with some (continuous) actions taking non-zero amounts of time, while digital state changes still took zero time. This was the first model which considered continuous changes rather than instantaneous changes occurring after a specified delay. Essentially, the older model allowed one continuous variable, time, while the newer model allows an arbitrary number of continuous variables. By modeling continuous changes, it makes it possible to describe what actually happens in the system, rather than worst case descriptions assuming that every transition happens at the latest legal time.

Berber [BL91] introduced Communicating Shared Resources, which uses a process algebra. They were sometimes called “Reactive Systems” [HG91].

Lynch and Tuttle introduced Input/Output Automata [LT88], which are precursors to Hybrid I/O automata [LSVW99]. Dolginova and Lynch [DL97] use these Hybrid I/O automata to model platoon join maneuvers in intelligent highway systems.

[HMP91] use what might be called a hybrid automata to model a simple timeout and a set of traffic lights.

Lynch and Vaandrager [LV91] want general formal model for real-time systems that generalizes the work of Lynch and Attiya [LA90]. They introduced “timed automaton”, an automaton with a (real-valued) absolute time for each state. Rather than putting fairness or liveness properties explicitly in the system, they model them using safety properties.

The Oxford Timed CSP Group has enhanced Hoare’s CSP (Communicating Sequential Processes) [Hoa85] by adding timing[SDJ$^+$91]. CSP is designed to describe several systems which may at times have to wait for one another. Events occur instantaneously, but one can specify one event to happen at a specific delay after another event. When more than one process can move, it is non-deterministic which one takes a step. Timed CSP appears well suited to modeling communications systems. Despite the apparent mis-match between distributed computing and modeling analog systems and the complexity of the notation, Timed CSP has been used to model an impressive set of problems: Telephony switches, Aircraft (turbine) engine starter, automotive guided vehicle, ethernet link protocol, lab robot, and a flexible manufacturing system.

Ciarlini and Frühwirth [CF00] automatically translate formal models of hybrid automata (using first order temporal logic) by a CLP program to get test specifications.

Timed Automata

Timed Automata are an earlier model for describing real time systems introduced by Alur and Dill in 1991 [AD91]. Timed Automata are a formal model which allow one to describe time between events very precisely. Except for the requirement that the system be described by a finite automaton, they are clearly powerful enough to describe the crucial issues in real time computing.

The main thing lacking in Timed Automata is a model of the continuous part of the system. The only continuous variable is a clock.

2.2.2 Hybrid Systems

Fahrland [Fah70] introduced the idea of hybrid systems (which he called "Combined Simulation") in 1970. Very little was published for the next 20 years. The first use I can find of the term *Hybrid Automata* describing a particular system is in Alur *et al.* [ACHH93] (which only allows linear continuous functions) which appeared in the first workshop on the theory of Hybrid Systems [GNRR93] held in late 1992, though the term hybrid system was coming into use in the REX workshop in 1991 [dBHdRR91]. Nicollin *et al.*[NOSY93] define a similar model for hybrid systems where each step consists of a flow transition followed by a jump transition.

Recent work on hybrid systems includes defining models [LSVW99] – cited above [LSV01] [ACH+95] [MMP91] – cited above [GJS96] and calculating the behaviour of the analog parts [HHMWT00]. – cited above

The Intelligent Highway Group at Berkeley has developed the SHIFT programming language [DGS] for describing evolving hybrid systems. [Mos99] provides a survey of a dozen simulation packages describing how much support each of them provides for simulating hybrid systems.

John Lygeros introduced the game theoretic approach to hybrid systems in his thesis.[LGS96] He has continued work on this approach [TLS00].

2.3 CLP Approaches to Hybrid Systems

We are not the first to use constraint logic programming to model and analyze hybrid systems. Gupta *et al.* [GJSB95, GJS96] introduced a ground breaking approach called "hybrid cc" which allowed one to formally describe hybrid systems using a logic programming language with constraints. Urbina [Urb96] has pioneered another approach using CLP($\mathcal{R}$)[JMSY92] to model and analyze hybrid systems. Delzanno and Podelski [DP99, DP01] have explored analyzing hybrid systems using CLP(Q,R) [Hol95], a system which handles linear constraints with real and/or rational coefficients, as well as Boolean constraints. Their approach is to define a translator from Shankar's guarded command language [Sha93] to CLP(Q,R).

2.4 Intervals Approaches to Hybrid Systems

We are not the first to apply interval arithmetic techniques to the problem of rigorously modeling hybrid systems. HyperTech [HHMWT00] take a major step towards reliability of their results by using interval arithmetic ODE solving as a tool to add rigor to the very successful HyTech system. This system merges, for the first time, the rigor of the formal model approaches and the practicality of the more engineering-based approaches by employing validated ODE solving. Our approach has several advantages over earlier models:

- CLP(F) is declarative, so that it describes system being modeled directly
- CLP(F) is logic based, so it can be viewed directly as a theorem prover using CLP logic
- CLP(F) is constraint based. It doesn't require one to fully specify a system. CLP(F) allows one to understand some properties of a system based on initial assumptions.

Chapter 3

Overview of Tools Used

This chapter provides a background for the rest of the thesis by describing the three main technologies underlying our analysis of hybrid systems. None of the work described here is mine, this chapter is here purely to provide background. The first technology is interval arithmetic, which was introduced by Moore in 1966 [Moo66]. The second is constraint logic programming (CLP) which was introduced by Jaffar and Lassez in 1987 [JL87]. The third is CLP(F), an implementation of Constraint Logic Programming with function variables and constraints, introduced by Hickey in 1994 [Hic94] and more fully described in 2000 [Hic00a]. CLP fits well with interval arithmetic, as constraints are a sensible way to describe an interval (with the interval's maximum and minimum each corresponding to a constraint), and CLP(F) takes advantage of this natural fit by combining CLP and interval arithmetic in a single elegant system.

The idea behind interval arithmetic is to express every real value X as an interval $x = [x_{min}, x_{max}]$ with $x_{min} \leq x_{max}$. In CLP(F), the endpoints x_{min} and x_{max} are floating point numbers. If $x_{min} = x_{max}$ it is a legal interval, and represents an X which is a floating point number. The interval x is chosen such that one can prove that the true value of X lies in the interval x. This allows us to deal rigorously with rounding errors in calculations and error bars in measurement. Section 3.1 provides an in depth discussion of interval arithmetic.

CLP [JM94] is a programming technique where possible answers are limited by sets of constraints. The program proceeds by using the set of constraints currently known to try to further constrain the set of possible answers. When the set of possible answers can no longer be shrunk, the program terminates. It is important to note that CLP does not necessarily provide a solution to the set of constraints, it merely provides a description of where any solutions must lie. Section 3.2 provides more details about CLP.

CLP(F) is an extension of CLP which allows one to specify constraints about functions (such as their derivatives, ranges, and values at specific points) as well as specifying constraints about real variables. This gives one the ability to describe ODEs, which is vital for almost any non-trivial hybrid system. Section 3.4 describes CLP(F) in more detail.

Interval arithmetic is an obvious choice for modeling hybrid systems, as the interface between the analog and the digital part involves imperfect hardware whose description must include error bars. Intervals are a very clear way of explicitly expressing error bars. Because CLP(F) provides constraints on the range of values that each variable can take on, it is well suited to describing intervals and its logical semantics makes it ideal for proving safety properties.

3.1 Interval Arithmetic

3.1.1 Problems with Floating Point Arithmetic

In order to be rigorous in our descriptions of hybrid systems, we must have accurate computation. Floating point numbers[1] introduce some problems with accuracy.

The standard method of describing non-integer numbers on a computer is to use a finite set of floating point numbers to model an infinite set of real numbers. This method has many problems, all stemming from the properties of floating point

[1] David Goldberg has written a useful tutorial [Gol91] on floating point arithmetic, and David Priest [Pri97] a helpful addendum showing how even implementations which conform to IEEE-754 can give different results for the same program.

numbers. The fundamental problem with floating point (FP) numbers is that they do not fit well with the number systems mathematicians favor (Reals, Rationals, Integers). FP is a proper subset of $\mathbb{Q}$, but FP is incomparable with $\mathbb{Z}$.

Operations on floating point numbers [Kah96] do not have the nice properties (associative, distributive ...) that we learned in third grade that all numbers have. FP is not closed under multiplication or addition, let alone division and subtraction. The lack of closure means that division is not the inverse operation to multiplication, and subtraction is not the inverse operation to addition. This introduces a class of round-off errors in which operations which are mathematically equivalent over the Reals give different results over FP.

Even with integers expressed in FP notation, we run into problems. Consider a large integer L, say 10000000000000000 and a small integer S, say 1. L is chosen to be larger than the FP mantissa, but smaller than the largest value representable in FP. Then $(S + L) - L$ is equal to S, but if calculated in FP, the result will be zero. Using FP to compute $S + (L - L)$ will correctly yield S. The correct choice of order of operations can help, but it is often not obvious in advance which order will yield the result which is closest to the correct value.

Even testing for equality correctly in floating point numbers is tricky, as two different (real) values may map to the same FP number, and, in the other direction, two values which should be the same may have been rounded differently because of a different order of operations.

These problems are not purely theoretical. There is a significant collection of examples where using limited precision calculation results in huge errors, and a few cases where one gets the same wrong answer even as one uses higher precision arithmetic.

Some examples follow:

Edalat and Heckmann [Eda01] give the following pathological example, which they attribute to Jean-Michel Muller:

$$a_0 = \frac{11}{2}, \qquad a_1 = \frac{61}{11}, \qquad A_{n+1} = 111 - \frac{1130 - \frac{3000}{a_{n-1}}}{a_n}$$

Let $a_n^{(k)}$ be a_n computed with k decimal digits of precision (They used the Unix program `bc` which allows one to specify the number of decimal digits of precision to be used in a calculation). They compute with five decimal digits of precision, to get a sequence which appears to converge to 100.

$a_0^{(5)}$	5.500	$a_3^{(5)}$	5.632	$a_6^{(5)}$	-3.241	$a_9^{(5)}$	100.209
$a_1^{(5)}$	5.545	$a_4^{(5)}$	5.648	$a_7^{(5)}$	283.1	$a_{10}^{(5)}$	100.012
$a_2^{(5)}$	5.590	$a_5^{(5)}$	5.242	$a_8^{(5)}$	103.738	$a_{11}^{(5)}$	100.001

At this point, one might feel confident that the sequence has converged, but just to make sure, they calculate a_{100} to higher and higher precision:

$a_{100}^{(5)}$	$100 + 10^{-5}$	$a_{100}^{(100)}$	$100 + 10^{-17}$	$a_{100}^{(130)}$	$6 - 10^{-8}$
$a_{100}^{(30)}$	$100 + 10^{-30}$	$a_{100}^{(110)}$	$100 + 10^{-7}$	$a_{100}^{(140)}$	$6 - 10^{-8}$
$a_{100}^{(60)}$	$100 + 10^{-57}$	$a_{100}^{(120)}$	-3.790		

After looking at the calculation with 60 digits of precision, one could justifiably feel quite confident that the sequence has a limit of 100. The calculations at 100 and 110 digits of precision (where the values are still close to 100, but not quite as close) are somewhat worrisome, but one is still startled by the values with 120 and 130 digits of precision. In fact, the correct answer is 6, not 100.

In addition to pathological cases, there are documented cases of normal problems surprisingly giving severely incorrect results. Günther-Jürgens, Endebrock, and

Klatte [GJEK87] give one such example, which resulted when a user doing statistical calculations involving multiple linear regression re-did some calculations on a different computer and got a different result. His variable `RESI` which represented the sum of squares for a least squares fit was 5.04 on one machine, and 10.01 on the other. It turned out that the machines had the same floating point hardware, but different versions of the FORTRAN compiler. The local computer consultants were interested, and reduced the problem to a few operations on a 5 X 5 matrix. From that, they produced the results in Table 3.1 showing different results from the same calculation on two different models of CDC CYBER computer and several different compilers. The "R" notation is a change in rounding mode which they don't define further.

CYBER	Compiler	Result
172	FTN 4.6	33.11
172	FTN 4.6 (R)	6.64
172	MNF	20.44
172	M77	20.44
76	MNF	15.22
76	FTN 4.8	33.11
76	FTN 4.8 (R)	2.09
76	FTN5	23.67
76	FTN5 (R)	2.60
76	FTNN5 (R=0)	42.51
76	M77 (R)	15.22

Table 3.1: Single Precision Results for `RESI` (from Günther-Jürgens *et al.* [GJEK87])

They then ran the same calculation on different computers with both single and double precision floating point, and get the results in Figure 3.2. The "DM" column shows the number of digits in the mantissa[2] of floating point representation. The "V" column gives the number of correct digits for the double precision calculations (The

[2]Recent work uses the term "significand" instead of mantissa.

correct value was calculated later by other means).

	Single Precision		Double Precision		
Computer	DM	Result	DM	Result	V
Fujitsu VP200	56	10.425	112	10.626840483201	11
CDC Cyber 170	48	[1.4388, 42.514]	96	10.626840483200	11
CDC Cyber 180	48	1.1149	96	10.626840482649	10
Cray-1M	48	7.7812	96	10.626840482649	10
CDC CYBER 205	47	213.76	94	10.626840484621	10
CGK TR440	38	45149.	84	10.626839084364	6
ICL 1906S	37	-140.49	74	10.626839287645	6
Sperry 1100	27	[25299, 3668400]	60	10.627052877666	4
IBM 3032	24	[-84223, 151810]	56	10.589253698243	2
NCR DM V	24	-9617700	52	10.377879736013	2
Prime 250	24	26809000	48	54.408453452592	0
Siemens 7.536	24	-84223	56	10.627538120607	4
DEC PDP-11	23	11824000	55	10.626978262677	5

Table 3.2: `RESI` Results for Different Computers (from Günther-Jürgens *et al.* [GJEK87])

Note that in single precision calculations, results varied greatly. Since the calculation was a sum of squares of errors, negative values are theoretically impossible.

Often people reasonably assume that a commercial software package will allow them to use statistics without becoming a statistics expert. So they then tried a number of commercial software packages using single precision arithmetic, and got the results in Table 3.3. This table shows that those hopes are often in vain.

The original user had checked that the matrix had a non-zero determinant, but did not realize that he should check for a large condition number. A naive user could well not be aware that calculations on matrices with large condition numbers are prone to large error.

A correct answer was found both by using an interval technique on the original

Package	`RESI`
NAG-Library (G02CJF, CYBER 76)	627.35
SPSS (New Regression, CYBER 76)	108.65
SPSS (Nonlinear, CYBER 76)	10.63
BMDP (BMDPlR, CYBER 76)	197.07
BMDP (BMDPAR, CYBER 76)	206.03
GLIM (DEC VAX)	10.63

Table 3.3: Standard Software calculations of `RESI` (from Günther-Jürgens *et al.* [GJEK87])

problem, and by transforming the problem into an equivalent matrix with a reasonable condition number.

It is impossible to know how often users fall into such traps, as one only learns of the rare cases in which the error is found, and the person involved is willing to accept the embarrassment of pointing this out. From the wildly different answers the statistical packages gave, one might guess that this is a common problem.

Acton's book [Act96] on pitfalls of computing with floating point numbers contains numerous examples of computations where one can easily get very large roundoff errors.

There are several approaches to the problems of doing arithmetic on computers. The most popular is to use floating point processors, and hope that the rounding errors aren't too large. This has the advantage of being very simple, but does not inspire confidence. Another approach is numerical analysis [Neu01]. This is a rather painstaking process, which must be redone for any significant change in the problem being considered.

In the 1980s, there was work on adding features for scientific computation to various programming languages (see Kaucher, Kulisch, and Ullrich [KKU87] for several examples.)

Another source of errors is insufficiently exact data, which Mikhlin [Mik91] categorizes as "inevitable error". Understanding this sort of error and how measurement errors propagate to calculated values (often called "sensitivity analysis") is a common problem for scientists, to which Taylor [Tay97] devotes a book. Interval arithmetic provides an automatic way of overestimating the errors in a final result stemming from the errors in measurement.

We use interval arithmetic [Moo66] to make these errors explicit. Interval arithmetic uses an interval (X_{min}, X_{max}) to represent each real number X. The true value of X is within the interval representing X. Hickey, Ju, and van Emden [HJvE01] have shown that by careful use of IEEE 754 floating point standard [IEE85] rounding directives, it is possible to soundly and efficiently perform arithmetic operations on potentially unbounded intervals with floating point numbers as end points using standard commercial floating point hardware.

Interval arithmetic can also be done without using floating point hardware. Edalat and Heckmann [EH02] use "linear fractional transformations", in which they use infinite precision rationals as the end points of intervals, and calculate as much precision as is needed. This is a sort of lazy evaluation. Possibly their system could be substituted for the `sound_ops` library currently used in clip. `sound_ops` is the C library for interval arithmetic which clip uses.

3.1.2 Errors and blunders in scientific computing

In the seminal paper [vNG47] "Numerical Inverting of Matrices of High Order," von Neumann and Goldstine describe four sources of errors in scientific computing:

1. *Theory Errors* – errors in the underlying scientific theory

2. *Measurement Errors* – errors in the observational data

3. *Truncation Errors* – errors due to approximation of solutions by algebraic expressions

4. *Roundoff Errors* – errors due to the fixed precision of computer arithmetic

These errors are inherent in physical science computations, but interval arithmetic provides tools to lessen the damage caused by the last three of them. The last two are due to the continuous nature of the problems we are trying to solve and the essentially finite nature of digital computation. We start by describing how interval arithmetic helps eliminate (or at least document) these errors.

1. *Measurement Errors* Because we do all our calculations on intervals, it is easy to enter a measured value as an interval (or as a value with an error estimate.) We then carry the calculations through allowing the measurement to take on any value in the interval. This means that if a calculation is heavily dependent on the precision of an imprecise measurement, the answer given will contain the possible range of answers.

2. *Truncation Errors* The use of intervals allows us to use classical series approximations, while still maintaining complete accuracy. We express the error term at the end of the series as an interval, and carry that through our calculations. If we were to truncate a series too soon, it would lead to an overly wide interval in the answer.

3. *Roundoff Errors* IEEE-754 [IEE85] provides control over rounding. By using the ability to specify whether a value will be rounded up or rounded down, we insure that our calculated intervals are a superset of the actual intervals.

There are obviously some important sources of error that are not on this list, these are the "avoidable" errors or blunders, that arise from the mistakes made by individuals involved in setting up the computation.

Some of the most common of these mistakes are:

1. *Analysis Blunders* – this occurs when the analyst makes a mistake in estimating the errors introduced by his computational solution. Interval arithmetic usually makes these errors explicit.

2. *Algorithmic Blunders* – these are the errors that arise when the idealized algorithm proposed by the analyst does not correctly solve the mathematical problem.

3. *Programming Blunders* – these are the errors that arise when the algorithm is incorrectly implemented in a particular programming language. One can make these less likely by making the language in which the problem is described closely match the natural description of the problem. CLP(F) tries to do this by making ODEs a part of the language.

4. *Software-use Blunders* – this covers the case of users who apply the software in ways for which it was not intended, e.g., applying it with variables that are outside of the range for which it was designed. The use of intervals helps to catch this by giving wide result intervals in cases where the program is inaccurate.

5. *Hardware Blunders* –these are the errors that arise when the hardware does not implement the arithmetic operations according to the published specifications (currently IEEE-754 [IEE85] is most widely used.) In this thesis, we will assume that the underlying hardware correctly implements IEEE-754, but this is clearly an area where formal verification is sorely needed.

In the everyday world of scientific computing, most of the practitioners are working scientists and not numerical analysts. This significantly raises the likelihood of the computed results being subject to one or more of these computational blunders.

Interval Arithmetic [AK93, AH83, HS81, Moo66, Neu90, JKDEW01, CW98, LvG00] is one well-studied approach to handling some of these blunders. The idea behind this method is to free the analyst from the difficulties of estimating the effects of roundoff, measurement, and truncation errors by automatically computing an over-estimate of the error for each variable in the model. This is done by representing each variable in the mathematical model by a floating point interval (or a set of intervals) and defining all arithmetic operations and mathematical functions on these intervals in such a way that the actual result of any primitive operation or function is guaranteed to be contained in the computed result. Interval arithmetic is very old. Archimedes used an interval technique to compute π by considering inscribed and circumscribed regular polygons for a circle. There were a few papers in the 1920s and 1930s [Bur24, You31] though it was not formalized until Moore's [Moo66] work in the 1960s. Kearfott provides an excellent introduction to Intervals [Kea96]. Hayes [Hay03] provides a less technical introduction.

Just because the error is automatically computed does not mean that the vast literature of classical numerical analysis is not useful. Interval arithmetic algorithms are still subject to many of the same kinds of algorithmic errors that arise with floating point computation, but rather than result in incorrect values, these blunders typically result in wide intervals. For example, evaluating $1 - \cos^2(x)$ in floating point arithmetic for x small, will result in a loss of about $2b$ bits of precision if x is approximately 2^{-b}. If the same expression is evaluated using interval arithmetic, one gets an interval containing about 2^{2b} floating point numbers. Thus, interval arithmetic introduces a new kind of problem as well as making an old one slightly worse:

- *Precision Problem* – this covers the case where the program is not able to sufficiently narrow the result intervals.
- *Performance Problem* – this covers the case where the calculation requires an unreasonably large amount of time.

When either of these problems is detected, the insights of classical numerical analysis can then be used to suggest ways of restructuring the computation to increase the interval convergence rate of the algorithm.

Interval Arithmetic techniques can be used by expert numerical analysts to produce software which adheres rigorously to the mathematical model and computes result intervals which are guaranteed to contain the theoretical result values (barring any blunders by the analyst). Interval arithmetic thus has brought numerical analysis fully into the realm of pure mathematics for the first time, since one no longer needs to rely on heuristic arguments about the statistical unimportance of the roundoff and truncation errors.

The price for this mathematical rigor is that interval arithmetic routines often take longer to provide an answer and always overestimate the error. Empirically it is noted that these overestimates are sometimes so large as to render the classical algorithms useless (particularly iterative algorithms with many iterations). There has been considerable effort in the last thirty years at developing interval arithmetic algorithms which are not only mathematically correct, but are efficient and precise as well [AK93, Boh96] . As promising as this recent work in interval analysis has been, it does not address most of the common blunders in scientific computing and still requires the working scientist to use tools for solving very specific types of numerical problems or to write a solver in some standard language but using interval variables rather than doubles or floats at certain places. This latter option does little to remove many of the opportunities for blunders, and in some sense may aggravate this situation since it requires scientists to be aware of the difference between interval variables and numerical variables and when to use which.

3.2 Constraint Logic Programming

Constraint Language Programming (CLP) was introduced by Jaffar and Lassez [JL87] in 1987, and further developed by Van Hentenryck in his thesis [vH89]. CLP grew out of logic programming (see Lloyd's book [Llo87] for information on logic programming). Logic programming, in turn, grew out of automatic theorem proving and artificial intelligence in the early 1970s. Jaffar and Maher [JM94] provide an excellent survey of CLP systems and the history of CLP.

CLP essentially adds an algebra to Prolog. If H is an algebra of terms, CLP(H) corresponds to Prolog. CLP($\mathcal{R}$) [JMSY92] was an early attempt to add the algebra of real numbers to prolog, but because floating point numbers are a poor approximation of the reals, rounding errors can cause their usual problems and the logical semantics of a CLP are violated. There have been other systems which added the algebra of numbers to CLP, but most of them suffer from the same problem with floating points poorly representing reals.

An annual conference on "Principles and Practice of Constraint Programming" has been held since 1995 [MR95, Fre96, Smo97, MP98, Jaf99, Dec00, Wal01, Hen02, Ros03]. Many of the workers in constraint programming approached it as a way of dealing with questions in artificial intelligence.

3.3 Interval Arithmetic Constraints

The key idea behind interval arithmetic constraints [BO97, Ju98, Cle87, Hic94, OV93, HvEW98, Hyv89, vE97a, vE97b, Ben95, BMvH94, vHMD97, Res88, vHMB98] is to view numeric computing problems as constraint systems that relate a set of real (or complex) variables or functions. The variables whose values are to be computed are initially unbounded in this model (i.e., they have the value $[-\infty, \infty]$). The goal of the computation is to shrink the intervals of the variables in such a way that no solution to the original system is removed. The shrinking is done by iteratively applying various contraction operators which are automatically generated from the constraint set.

There are several interval arithmetic constraint solvers which are currently available. In all of these the user specifies only the constraints to be solved and the system determines which contractors to apply and in which order.

For example, in IAsolver [HQvE00], the constraints are compiled into a set of primitive constraints (much the same as 3-address code is generated from expressions by a compiler), and contractors for these primitive constraints are repeatedly called until a fixed point is reached or some resource bound is exceeded. Note that since IEEE floating point has a successor function, there are only a finite number of contractions which can take place before a fixed point is reached.

The Numerica system [vHMD97] uses interval arithmetic techniques to solve non-linear optimization problems with a relatively small number of constraints and variables. This system uses three types of contractors, one of which is the multi-variable interval Newton contraction, and also relies on domain-splitting to search for solutions.

These two systems share the property that the choice of contractors and the strategy in which they are selected is hidden from the user. Thus, if the user correctly enters the mathematical formulas describing the problem to be solved, then the system will produce intervals which are guaranteed to contain the correct solution.

There are also several Constraint Logic Programming languages which provide general non-linear interval arithmetic constraint solving. In CLP(RI) [Sys96], for example, a constraint is specified by a standard mathematical expression enclosed in curly braces. In CLP(F) [Hic94, Hic00a], the constraint language is extended to allow for differentiable functions and ODE constraints.

This scheme eliminates all possible blunders except for misuse of the software. It does still allow precision problems (i.e. the resulting intervals being unnecessarily large) and performance problems. The price of such protection is that the user plays no role in formulating the algorithm to solve the problem, and if the system is not able to sufficiently narrow the intervals quickly enough the user has few if any alternatives.

3.4 CLP(F)

CLIP [Hic00b, HJ] is an implementation of a CLP language whose domain is the reals, and which was first built on SICStus Prolog [CWA+91] by Hickey and Ju, and later ported to GNU Prolog [Dia02]. CLIP implements CLP(I) and CLP(F). CLP(I) is a constraint language over the reals. CLP(F) extends CLP(I) to allow constraints which specify arithmetic and analytic relations among real and functional variables. Hickey provides a description of the language CLP(F) in [Hic01]. CLIP is open source software, available at `interval.sourceforge.net`

Earlier CLP(Intervals) systems include Benhamou, McAllester and Van Hentenryck's [BMvH94] work on `Newton` , and Benhamou Older [BO97] and Benahmou's [Ben95], work on CLP(BNR).

DeVille *et al.* [DJvH02] explore a technique for minimizing the size of intervals resulting from solving ODEs using constraints and intervals. As they point out, their techniques would fit well with CLP(F), and might improve the performance of CLP(F).

CLP(F) is related to QSIM [Kui93] in that each attempts to find an over approximation of the possible states of a system of ODEs (QSIM uses the term "qualitative behaviours") .

At this time, there is very little documentation of CLP(F). There are some descriptions in Hickey's papers [Hic01, Hic00a], and there is a manual in early draft form [WH04], which appears as appendix A of this thesis. In this section, we provide an overview of CLP(F), its semantics, its implementation, and its use.

3.4.1 CLP(F) Constructs

CLP(F) is a superset of Prolog [Pro95, DEDC96], and uses Prolog's syntax and user interface. As is common in CLP languages, the constraints are enclosed in curly braces "{ }". The different types of variables are declared using the `type` predicate. Constraints over functions are enclosed in "{[ ]}". The CLP(F) interpreter provides answers to queries in the form of a sequence of solution sets, where each solution set provides a real interval for each of the constraint variables. The soundness property of CLP implies that every correct solution to the query must be contained in one of the solution sets (assuming that the program eventually terminates). On the other hand, not every element of the solution set is guaranteed to be a solution (and indeed, there may not be any actual solutions in any particular solution set returned by the interpreter).

Rigorous Numeric Constraint Solving

The CLP(F) constraint language allows one to express queries about real variables and analytic functions over a finite interval of the reals, using any equations and inequalities constructed using the arithmetic operators and the standard mathematical functions (sin, cos, tan, asin, acos, atan, exp, log, exponentiation, integral powers X^Y, and others). For example, given the query:

```
| ?- {X^2=2,X>0}.
```

CLP(F) responds

```
X = 1.41421356237309... ? ;
no
```

```
| ?-
```

The ellipsis indicates that all the digits shown are correct.

Multiple Solutions and Non-determinism

Sometimes there may be more than one solution to a given constraint. In this case the simple solver returns a single interval that contains all solutions:

```
| ?- {X^2=2}.
X = [-1.41421356237309536751922678377,
      1.41421356237309536751922678377] ?
no
| ?-
```

The user must guess whether this is because the system has failed to narrow the interval around one solution or if this is a case where there are multiple solutions. There is a moderately sophisticated solver `solve_clip(METHOD,VARS,N)` which allows one to specify the solving method, (See section A.7.1 for a list of supported solving methods, and a description of each of them.) the list of variables that should be solved, and a parameter N representing how much work should be done (e.g. a maximum allowed width for intervals, or a maximum depth for a divide and conquer splitting routine). In earlier work [HW99a, HW99b] we discuss methods of determining which is the case, and choosing an appropriate method to use in `solve_clip`.

Here, to find the discrete set of solutions one must explicitly apply a divide-and-conquer approach where one divides the interval into subintervals and searches for solutions in each one. This is done using the “queue” method of the `solve_clip` solver and typing a semicolon after each solution that it finds:

```
| ?- {X^2=2},solve_clip(queue,[X],0.000001).

X = 1.41421356237309... ? ;

X = -1.41421356237309... ?

(10 ms) no

| ?-
```

The “no” at the end indicates that there are no more solutions to that query.

If the result returned by CLP(F) is too wide an interval, one can adjust tuning parameters which control how many attempts at narrowing clip will make. These parameters offer a complex trade-off between speed and precision. The tuning parameters are discussed in section A.7.1.

3.4.2 Analytic Constraints and ODEs in CLP(F)

In CLP(F) the constraint domain allows one to declare variables representing various analytic values including:

- *real numbers, X*
- *infinitely differentiable functions, F, on a finite interval [a,b]*
- *vectors of numbers, functions, or, recursively, vectors*

CLP(F) also allows one to constrain functions and real variables by equations involving derivatives and arithmetic, trigonometric, or exponential functions. In addition, one can constrain a function to take specific values at specific points and to have a range that lies within an interval. See Appendix A for a precise description of the constraint language

Consider the following mathematical constraint Q on the function variable F and real variables A and E:

$$Q(F, A, E) \equiv$$

$$(F \in \mathcal{H}([0,1]), F' = F, F([0,1]) \subseteq [-1000, 1000], F(0) = 1, F(A) = 2, F(1) = E)$$

Q can be represented and solved by presenting the following constraint to the CLP(F) interpreter:

```
| ?- set_clip(sensitivity, 0.0).

| ?- type([F],function(0,1)), {[ ddt(F,1)=F,   F in [-1000,1000],
       eval(F,0)=1,eval(F,A)=2, eval(F,1)=E ]}.
```

where the `type` predicate indicates that $F \in \mathcal{H}([0,1])$, i.e., F is an infinitely differentiable function in some open neighborhood of the interval $[0,1]$. The `set_clip` command is used to tune CLP(F) by controlling under what conditions it puts a changed constraint back on the stack. This enables the user to trade off computation time against the precision of the result. See section A.7.1 for a list of tuning variables and their meanings. The logical interpretation of the code is discussed in the next section.

The output given by CLP(F) after 0.1 seconds is

```
A = .6931471...   E = 2.718281...
```

which represents the following answer constraint:

$$C(F, A, E) \equiv (A \in [0.6931471, 0.6931472) \ \wedge \ E \in [2.718281, 2.718282))$$

The soundness of the CLP(F) interpreter implies that it has proven a theorem about the query and its solution constraint:

$$\forall F, A, E \quad Q(F, A, E) \Rightarrow C(F, A, E)$$

In other words, if F, A, and E represent a solution to Q, then they must satisfy the answer constraint C. Note that one cannot infer from this theorem that Q has any solutions at all. In this particular case, Q clearly does have a solution

$$F(t) = \exp(t), \quad A = \ln(2), \quad E = e$$

which of course satisfies the answer constraint C.

3.4.3 How CLIP Works

The CLP(F) system solves analytic constraints by soundly approximating analytic functions using power series and introducing arithmetic constraints among the Taylor coefficients of the functions at the endpoints, at points in the interval, and over the entire range. Since CLP(F) represents functions as Taylor series, it can easily calculate derivatives of functions, and enforce constraints on those derivatives. Since CLP(F) includes the remainder term in Taylor series, it can be (and is) rigorous.

Consider again the constraint that specifies that F is a function on $[0, 1]$ such that

$F' = F$ and $F(0) = 1$ and $F(A) = 2$ and $F(1) = E$ and $F([0, 1]) \subset [-1000, 1000]$:

```
| ?- type([F],function(0,1)), {[ ddt(F,1)=F, F in [-1000,1000]
        eval(F,0)=1, eval(F,A)=2, eval(F,1)=E ]}.
```

The `type` predicate is used to declare that F is an infinitely differentiable function on the interval $[0, 1]$. Thus F is represented by a list of its Taylor coefficients at 0 $(F_{00}, F_{01}, F_{02}, .., F_{0n})$ and at 1 $(F_{10}, F_{11}, ...)$ and the ranges of its derivatives over $[0, 1]$ $(R_0, R_1, ...)$, related by the Taylor formula with remainder. The function F is then constrained to be equal to its first derivative (i.e. $F_{ij} = F_{i,j+1}, R_i = R_{i+1}$, and to take the value 1 at 0 ($F_{00} = 1$) and to take values in $[-1000, 1000]$ for all $x \in [0, 1]$ (i.e. $R_0 \subseteq [-1000, 1000]$). The variables A and E are not declared to be functions and hence are real constants by default. They are constrained so that $F(A) = 2$ and $F(1) = E$ (e.g. $F_{10} = E$ and for each $n = 1, 2, \ldots, 2 = F_{00} + F_{01}A + F_{02}A^2/2! + \ldots + Z_n A^n/n!$ for some $Z_n \in R_n$). The constraint solver finds A and E to 7 decimal digits of precision and also finds an interval for F (not shown here) and specifies intervals F_{ij} for its first 10 derivatives at 0 and 1, and intervals R_j for the range of its first 10 derivatives over $[0, 1]$. The number of derivatives (10) can be set to any value N (but space and time complexity grows quadratically with N). One can use CLP(F) to define higher order constraints which specify that two points lie on a trajectory defined by on ODE.

Hickey [Hic00a] provides a more thorough description of CLP(F).

3.4.4 Programs

CLP(F) programs are Prolog programs in which the bodies of rules may contain CLP(F) constraints. CLP(F) provides the full power of Prolog in addition to the power of the underlying constraint solver and both are combined within a single logical semantics. Moreover, by the soundness and completeness of CLP semantics [JM94], if a CLP interpreter returns N solution sets $C_1, \ldots, C_n$ for a query $Q(X, F)$ and then halts, then every solution of the query $Q(X, F)$ consisting of a real vector X and a vector F of real-valued functions, is contained in the union of the solution sets C_i.

The logical semantics of CLP(F) programs can be summarized in the following theorem [Hic00a].

Theorem 1 *Let P be a CLP(F) program, $Q(x)$ a CLP(F) query where x is a tuple of real variables, and assume the interpreter returns N answer constraints $\{x \in I_j\}$ for tuples of intervals $I_1, \ldots, I_N$ and then halts. Let P^* be the first order theory obtained from a logic reading of P (by Clark's Completion Semantics [Cla78][Llo87]), and let T be the first order theory of the domain F of analytic functions on real intervals. Then one can infer that*

$$P^* \cup T \vdash \forall x \left(Q(x) \Rightarrow x \in \bigcup_j I_j \right)$$

Corollary 1 *Notation as in the previous theorem. If the interpreter halts with no answer constraints (i.e., N=0), then one can infer*

$$P^* \cup T \vdash \neg \exists x \; Q(x)$$

i.e., the query is not satisfiable.

This theorem allows one to infer correctness of a CLP(F) simulator for a hybrid system as well as safety properties of the system directly from the corresponding

CLP(F) program.

Chapter 4

Modeling Non-linear Hybrid Systems in CLP(F)

We model hybrid systems using CLP(F) programs with parameterized ODE constraints. The parameters change both in response to physical changes (such as a water level crossing from above a pipe to below it), and state transitions of the digital part of the system (such as some element being turned on). This method of using the same modeling technique for what seem to be two distinct sorts of events simplifies the programming system.

We apply our techniques to a few widely studied hybrid systems. In this chapter we consider a system consisting of a thermostat with delay. In the next chapter, we consider a simple version of the "two tanks" system. In Chapter 6, we will consider a generalization of the two tanks system to n tanks, with other added complexities.

4.1 General CLP(F) Hybrid System Simulator

The program in Figure 4.1 is one way of implementing a general hybrid system simulator in CLP(F). The first parameter of the `evolve` procedure is the state of the hybrid system at some time `t0`, which consists of a discrete state S and a continuous state X. The second parameter is a list of values used to specify the particular hybrid system. The third parameter is the final (or ending) state of the hybrid system at some time $t1$, with $t0 \leq t1$.

Observe that the procedures are relatively simple. To evolve in a trajectory, `evolve` checks that it doesn't change digital state, and that the continuous state changes are in accordance with the ODEs. `evolveSC` allows the system to change discrete states, and models the system evolving from one discrete state change to the next. Typically, the continuous state will be represented by a pair of dependent

```
% evolving along a trajectory in state S
evolve((S,X0),C,(S,X1)) :-
   in_trajectory((S,X0),C,X1).

% evolving through discrete state changes
evolve((S0,X0),C,(S1,X1)) :-
   in_trajectory((S0,X0),C,X01),
   evolveSC((S0,X01),C,(S1,X11)), % X01,X11 are state change points
   in_trajectory((S1,X11),C,X1).

% these evolve from statechange point to statechange point
% The "eqstate" is there because CLP(F) doesn't unify interval
% variables in the head...    You have to explictly unify them.
evolveSC((S0,X0),C,(S1,X1)) :-
  eqstate(X0,X1),  statechange((S0,X0),C,S1).

evolveSC((S0,X0),C,(S1,X1)) :-
   statechange((S0,X0),C,S), % S01->S occurs at state X0
   in_trajectory((S,X0),C,X), % X is in the trajectory after (S,X0)
   evolveSC((S,X),C,(S1,X1)). % evolve from (S,X) to end...

% test if two continuous states are equal
eqstate(X0,X1) :-
  X0=(T0,A0),X1=(T1,A1),{T0=T1,A0=A1}.
```

Figure 4.1: A general simulator for hybrid systems

variables `(T,A)` where `A` is the value of some ideal system sensor at time `T`. The `in_trajectory` procedure looks up the ODE, `C`, that should hold in this state and applies that ODE to the initial values `X0` to get the a constraint relating `X0, X1` and `C`. We use a slightly modified version of this simulator in Section 6.4.

4.2 Rigorous Analysis of a Thermostat with Delay

This section shows how CLP(F) handles explicit time delays, and how we can use it to ask questions about what values parameters should have to give a particular behaviour, as well as asking how a system with particular parameters will behave. This sort of analysis requires no more programming than modeling does, as CLP languages are defined in such a way that constraints propagate in either direction. Thus, with a CLP language, one can choose which values to look for, simply by using variables for those parameters in a query. We consider the hybrid system of a thermostat introduced by Henzinger *et al.* [HHWT98]. This is a system consisting of a stirred pot of water with a temperature sensor and a heater in it. When the measured temperature goes above a threshold, the logic circuit shuts off the heater (after a small delay). Similarly, when the measured temperature goes below a threshold, the logic circuit turns on the heater (after a small delay). The safety property in question is to establish upper and lower bounds for the temperature of the water.

We use CLP(F) to define higher order constraints which specify that two points lie on a trajectory defined by on ODE. For example, in the simplest model of a thermostat we use the following CLP(F) procedure (taken from Figure 4.3, where

[0] This section describes work from Hickey and Wittenberg's paper [HW03] in the 2004 Florida AI Research Symposium.

`T0,T1` are times and `A0,A1` are temperatures at those times, `A` is the temperature function (so $A(T0) = A0$) and `Alpha, Beta` are the heat loss and the heater element components, respectively, of the ODE for `A`. The parameter `I` is a bound on the width of the interval on which `A` is defined. This bound is required because in CLP(F) all functions must be defined on finite intervals.

```
ode((T0,A0),[I,[Alpha,Beta]], A,(T1,A1)) :-
   type([A],function(0,I)),
{[ ddt(A,1) = Alpha * A  + Beta, eval(A,0)=A0, eval(A,T)=A1,
    A in [-1.0E100,1.0E100], T=T1-T0,   T in [0,I]   ]}.
```

This code means that the predicate `ode((T0,A0),[I,[Alpha,Beta]], A,(T1,A1))` is true if and only if

- `T0, T1, A0, A1, I, Alpha` and `Beta` are real numbers.
- `A` is a function defined on $[0, \mathrm{I}]$
- $\frac{dA}{dt} = \mathrm{Alpha} \cdot \mathrm{A} + \mathrm{Beta}$
- $\mathrm{A(T0)} = \mathrm{A0}$
- $\mathrm{A(T1)} = \mathrm{A1}$
- $\forall t \in [T0, T1] \mathrm{A(t)} \in [-1.0 \cdot 10^{100}, 1.0 \cdot 10^{100}]$

The CLP(F) system is able to use this definition to compute $(T1, A1)$ from $(T0, A0)$, or, as we will see below, to use this procedure to find values of the parameters `Alpha` and `Beta` which make the system behave in some desired fashion.

In this case the ODE is $f' = af + b, f(0) = a_0, f(t) = a_1$ which can be solved exactly. CLP(F) can handle trigonometric or exponential functions as well as the linear functions shown here. In the next chapters we consider ODEs which (as far as we know) have no closed form solutions.

To solve this exactly, use the solution

$$U \cdot e^{\alpha \cdot T} - \beta/\alpha$$

which we program as follows:

```
newode((T0,A0),[I,[Alpha,Beta]],A,(T1,A1)) :-
 %  type([A],function(0,1)),
  {
    C = Beta/Alpha,
    A0=U*exp(Alpha*T0) - C,
    A1=U*exp(Alpha*T1) - C
  }.
```

This has the same behaviour as the previous program, except that it doesn't use function variables, and will narrow its results more effectively.

Since CLP(F) uses brute force to model ODEs, it does not perform better on ones which are solvable analytically. In Chapter 6 we show an example of using CLP(F) to model a system with more complex functions.

In order to demonstrate CLP(F)'s ability to handle more complex examples, we can model a system consisting of a fluid with temperature $A(t)$ which is heated by a heating element whose temperature $B(t)$ has a non-linear component $\sin(B(t))$ in its defining ODE. The component $\sin(B(t))$ is not natural, but shows how CLP(F) can handle non-linear components which are common in real systems. This system is modeled by the procedure in Figure 4.2. We are unaware of a closed form for the solutions of this ODE.

```
ode2((T0,A0),[I,[Alpha, Beta, Gamma, Delta]] ,A,(T1,A1)) :-
   type([A,B],function(0,I)),
{[ ddt(A,1) = Alpha* A + Beta +Gamma*B,
    ddt(B,1) = Delta*(B + 0.1*sin(B)),
    eval(A,0)=A0,   eval(A,T)=A1, eval(B,0)=1,
    A in [-1.0E100,1.0E100], B in [-1.0E100,1.0E100],
    T=T1-T0, T in [0,I]
]}.
```

Figure 4.2: Thermostat with Non-Linear Component

4.2.1 Henzinger's Model and Analysis

In this section, we present the model of a thermostat with a delay in switching used by [HHMWT00]. Henzinger's model consists of a finite state controller with an analog input measuring the temperature in the tank. The controller has a 1-bit output to control a heater in the tank. The tank always loses heat at a rate directly proportional to the temperature, and, while the heater is on, is heated at 4 degrees/second. Mathematically, after the heater reaches equilibrium in the on position $A' = -A + 4$ and at equilibrium in the `off` position $A' = -A$. The controller switches the heater off within one second of the temperature going above a pre-set value, and turns the heater on within one second of the temperature dropping below another threshold. Note that this model assumes that the thermometer is perfect, the heater produces a constant and perfectly known heat output, the element heats and cools instantly and the physics of the tank are perfectly modeled by the ODE. Given those assumptions, they then use interval techniques to eliminate round off errors in proving safety properties.

4.2.2 CLP(F) Model of the Thermostat

In this section we present two models of a thermostat. The first simple model demonstrates the key ideas. The second illustrates how one can easily extend the simple model to a model that more faithfully represents the real hybrid system by more closely approximating the physics of the system.

The Simple Model

Our first CLP(F) model of a thermostat is shown in Figure 4.3. To clarify the key concepts, this first model assumes there are only two states: `on` and `off`. When the system state is on, the ODE governing the temperature A is $A' = -A + 4$. When the system state is `off`, the ODE is $A' = -A$. The system switches from `on` to `off` when the temperature rises above 2.3 and it switches from `off` to `on` when it drops below 1.8. The `in_trajectory` procedure models the trajectory by looking up the proper ODE for the current state and then calling the ODE procedure to constrain the new state variables `(T1,A1)`. It also adds the constraint that the temperature range is contained in `[-1000, 2.3]` (resp. `[1.8,1000]`). This is not needed for our simple example because the temperature rises monotonically and then falls monotonically and then rises again. With more complex models, the temperature might not behave so nicely so this constraint states that no point in the trajectory has passed the threshold for switching. The `statechange` procedure simply indicates the condition that signals a state change and provides the new state. The `ode` procedure models the specified ODE as we have described above. Finally the `test` procedure shows how this program can be used to model the behavior of the system. It initializes the list describing the system to be analyzed and then invokes the `evolve` procedure.

A simple query ("at what times is the temperature 2?") to this system and the resulting answer is shown in Figure 4.4.

One subtle point about this model is that the CLP(F) solver will only work effectively if a sufficiently small step size is explicitly given (this is the `I` parameter appearing in the `in_trajectory` and `ode` procedures. If the step size is too large, then the CLP(F) solver will return very wide, unhelpful intervals for all variables. One approach to handling this is to introduce pseudo states `(on,n)`, `(off,n)`, where `n` is an integer representing the number of full steps that have been taken on the current trajectory in the current state. The continuous part can be modeled as `(t,a,z)` where `t` is the total elapsed time, `a` is the temperature at time `t`, and `z` is the time relative to the current step. Such an extension of the current technique is straightforward and we do not show it here due to space limitations.

4.2.3 A More Realistic Model

In the example shown in Figure 4.5, we refine the previous model by using six states `on,sw0,cooling,off,sw1,heating` corresponding to the states in Henzinger's model. The state diagram is shown in Figure 4.6.

The model also represents the continuous state as a triple `T,A,Z` where `T` is the total elapsed time, `A` is the temperature at time `T`, and `Z` is the time since the system entered the current state. The `Z` parameter is needed to implement the "switch-

```
evolve(H,C,H).

evolve((S,X0),C,(S,X1)) :- %% evolving along a trajectory in state S
   in_trajectory((S,X0),C,X1).

evolve((S0,X0),C,(S1,X1)) :- % evolving through discrete state changes
   in_trajectory((S0,X0),C,X01),
   evolveSC((S0,X01),C,(S1,X11)), % X01,X11 are state change points
   in_trajectory((S1,X11),C,X1).

% these evolve from statechange point to statechange point
evolveSC((S0,X0),C,(S1,X1)) :-
                eqstate(X0,X1), statechange((S0,X0),C,S1).

evolveSC((S0,X0),C,(S1,X1)) :-
   statechange((S0,X0),C,S), % S01->S occurs at state X0
   in_trajectory((S,X0),C,X), % X is in the trajectory after (S,X0)
   evolveSC((S,X),C,(S1,X1)). % evolve from (S,X) to end...

% test if two continuous states are equal
eqstate(X0,X1) :-  X0=(T0,A0),X1=(T1,A1), {T0=T1,A0=A1}.

in_trajectory((S0,(T0,A0)), [I, Min, Max, ODEs],(T1,A1)) :-
 member(S0=ODE,ODEs), {T=T1-T0,T=<I},  ode((T0,A0),[T,ODE],A,(T1,A1)),
 ( (S0=on,      {[A in [-1000,Max] ]});
   (S0=off,     {[A in [Min,1000]]})).
statechange((S0,(T0,A0)), [_I, Min, Max, ODEs],S1) :-
 ( (S0=on,      {A0=Max},  S1=off);
   (S0=off      {A0=Min },  S1=on) ).

ode((T0,A0),[I,[Alpha,Beta]], A,(T1,A1)) :-
   type([A],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta,  eval(A,0)=A0,   eval(A,T)=A1,
    A in [-1.0E100,1.0E100],  T=T1-T0,   T in [0,I]
]}.
```

Figure 4.3: Simplest CLP(F) model of a thermostat

```
% to find full/half cycle times:  call as ?- test(S,(T,A)),{A=2}.
test(S,X) :-
   % Step, Min, Max,  ODES for on/off
 C=[ 2.0,   1.8, 2.3,  [on=[-1,4],off=[-1,0]]],
 evolve((on,(0.2,0)),C,(S,X)).

| ?- test(S,(T,A)),{A=2}.
A = 2, S = on,  T = 0 ?
A = 2, S = off, T = 0.3022808718... ? ;
A = 2, S = on,  T = 0.5029515673... ?
```

Figure 4.4: Query to Simple Model - When is Temp = 2?

ing" specification which states that the system waits some amount of time after the threshold is passed before switching on/off the heating element. Likewise, the time in which the system is heating/cooling before it "jumps" to the maximum/minimum value is given by a time unit. This represents a discontinuity in the model since the heating temperature is assumed to immediately rise to the maximum at the end of the element-heating period.

The `sw0, sw1` states are when the system is waiting before switching the heating element on or off. The `heating, cooling` states are when the element is warming up or cooling down. The `on, off` states are when the element is fully on or off. Observe that the ODEs for each state are specified in the variable `C` of the test procedure. Also, observe that the switching conditions are given declaratively in the `statechange` procedure. Finally, note that the system is assumed to be modeled by the following more complex non-linear family of ODEs, where the parameters $(\alpha, \beta, \gamma, \delta)$ vary from state to state:

$$\forall t \in [0, I] \;\; A'(t) \;\; = \;\; \alpha A(t) + \beta + \gamma B(t)$$

```
in_trajectory((S0,(T0,A0,Z0)),R, [Step, Min, Max, Delay, Stime, ODEs],
                        (T1,A1,Z1)) :-
  member(S0=ODE,ODEs), {Z1=T, T=T1-T0, T=<Step},
  ode((T0,A0),[T,ODE],A,(T1,A1)),
  ((S0=on,      {[A in [-1000,Max] ]});
   (S0=sw0,     {[T =<  Delay]});
   (S0=cooling, {[T <   Stime, A in [Min,1000] ]});
   (S0=off,     {[A in [Min,1000] ]});
   (S0=sw1,     {[T =<  Delay]});
   (S0=heating, {[T <   Stime, A in [-1000,Max] ]})),
      {[V in [0,Z1], eval(A,V)=R]}.

statechange((S0,(T0,A0,T)),[Step, Min, Max, Delay,Stime,ODEs],S1) :-
((S0=on,      {A0=Max},     S1 = sw0);
 (S0=sw0,     {T=Delay},    S1=cooling);
 (S0=cooling, {T=Stime},    S1=off);
 (S0=cooling, {A0=Min},     S1=sw1);
 (S0=off,     {A0=Min },    S1=sw1);
 (S0=sw1,     {T=Delay},    S1=heating);
 (S0=heating, {T=Stime},    S1= on);
 (S0=heating, {A0=Max},     S1= sw0)).

ode((T0,A0),[I,[Alpha,Beta, Gamma,Delta]], A,(T1,A1)) :-
   type([A,B],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta +Gamma*B, ddt(B,1) = Delta * B,
    eval(A,0)=A0,   eval(A,T)=A1,   eval(B,0)=1,
    A in [-1.0E100,1.0E100], B in [-1.0E100,1.0E100],
    T=T1-T0,   T in [0,I]
]}.

test(S,X,D) :-
  C=[ 2.0,   1.8, 2.3, 0.05, 0.1, [on=[-1,4,0,1],off=[-1,0,0,1],
       sw0=[-1,4,0,1],sw1=[-1,0,0,1], heating=[-1,4,-4,D],
                cooling=[-1,0,4,D]
       ]],
  evolve((on,(0.2,0),C,(S,X)).
```

Figure 4.5: More Complete Model of Thermostat

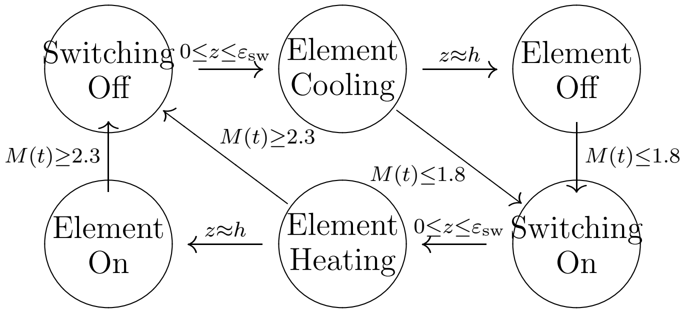

Figure 4.6: State model allowing thermostat to shut off before element is warm

```
| ?- {D in [-8,-7]}, test2(S,(T,A,Z),D),{T=0.5},S=off,
                 narrow_all(1000000).
A = 2
D = -7.6651...
S = off
T = 0.500*
Z = 0.1874810705022... ?;
(14280 ms) no
```

Figure 4.7: Example query of more complex model

$$\begin{array}{rcl} \forall t \in [0, I] \;\; B'(t) & = & \delta B(t) \\ T & = & T1 - T0, \;\; 0 \le T \le I \\ A(0) & = & A0, \;\; A(T) = A1, \;\; B(0) = 1 \\ A([0, I]), B([0, I]) & \subset & [-10^{100}, 10^{100}] \end{array}$$

The variable B represents the heat transfer from the heating element and we assume that the rate at which it heats and cools depends on its temperature.

Figure 4.7 shows an example in which the model is used to find all values of the ODE parameter δ in the range $[-8, -7]$ for which the system evolves to the state with

`S=off` and $A = 2$ in exactly 0.5 seconds.

4.3 Refining the Thermostat Model

One of the advantages to using a constraint based system is that one can use step-wise refinement to improve the accuracy of a model as one learns more about the underlying physics. Here we demonstrate this using the thermostat as an example.

The thermostat model as presented above could be refined in several ways to more accurately reflect the real Hybrid System. In this section we discuss possible refinements.

4.3.1 Accurately modeling measured data

The first refinement is to observe that in order to model the physical uncertainties, we could include in our model error terms for each physical measurement. In most cases the size of the error terms will be given by the manufacturer's specification. Interval arithmetic makes it straightforward to explicitly model the error-bars in various physical measurements. We simply consider each of the physical measurements to give an interval result rather than an absolute number.

4.3.2 Measuring Temperature

We use $A(t)$ to describe the actual average temperature of the water in the tank. Note that this is a platonic ideal, and cannot be measured at all. We could use $M(t)$ to denote the temperature measured by the thermometer and converted by the A/D converter. We would then assume that for all $t : |M(t) - A(t)| \leq \varepsilon_{\text{temp}}$. The value of

$\varepsilon_{\text{temp}}$ could be estimated from the specifications of the thermometer, and knowledge of the efficiency of the stirrer and the assumption that the temperature does not change too rapidly. To make this change in our code, we simply add the constraint above, and have jump transitions (which depend on measured values) use $M(t)$, while the flow transitions continue to use $A(t)$.

4.3.3 Heater Element Description

Henzinger *et al.* consider a heating element, and assume that the power transfered from the element to the water is a constant while the element is on. This is only a valid approximation when the water temperature is more or less constant.

Our model provides an improved approximation in that it includes the temperature of the heating element as part of the model. There is, of course, some variation in the heater output, and the actual power should be described by $g(t)$, which (after the power has been in the "on" position long enough to achieve equilibrium) differs from G by less than some small constant $\varepsilon_{\text{heat}}$.

We could alternatively have improved Henzinger's model by starting with a model of the heater being powered up or powered down by constraining the time $h_{\max}$ it takes the heater to get to within $\varepsilon_{\text{heat}}$ of G when turning on, or within $\varepsilon_{\text{heat}}$ of 0 when turning off, and noting that during that time, $-\varepsilon \leq g \leq G + \varepsilon$.

We could then refine this model by using a linear change from one state to the other with a sufficient error bar, so if the element was turned on at time t_0, $g(t) \in [-\varepsilon + ((t - t_0) * G - \varepsilon_{\text{heat}}), \varepsilon + ((t - t_0) * G + \varepsilon_{\text{heat}})]$ and we could also add similar error bars to the exponential decay model of the heating element which we use above. We could also use error bars for the heat loss and other parameters describing the Hybrid

System.

This sort of detailed analysis can be done for any hybrid system. What CLP(F) offers is a relatively simple way to make these refinements to an existing model.

Chapter 5

A more Complex Example: Two Tanks

5.1 Two Tanks in CLP

5.1.1 Description of the Two Tanks Problems

The “two tanks” problem is a system consisting of two water tanks. There is a flow of water in to the higher tank, and a horizontal pipe from the bottom of the higher tank to some point in the side of the lower tank. There is an outflow pipe at the bottom of the lower tank. A diagram of the system is shown in Fig. 5.1. In some versions of the problem, there are valves controlling some or all of the input flow, the flow in the pipe between the two tanks, and the output flow. The obvious questions to ask are “Is there an equilibrium given a set of flow rates?”, “Does either tank overflow before equilibrium is achieved?”, and, in the case where the model has program controlled valves, “Does some particular program have a specified safety property?”

Kowalewski *et al.* [KSF$^+$99] use six methods to model what they consider to be a realistic version of this two tanks problem previously studied by the same group (Stursberg *et al.* [SKHP97]). Later, Henzinger *et al.* [HHMWT00] provided another technique for studying a simplified version of this problem. Here, we consider the simplified version with no valves. In the next chapter we will add valves to show the CLP(F) system is well adapted to stepwise refinement of a problem.

Mathematics of the Two Tanks Problem

The precise problem we study can be described as follows. There are two tanks, an upper tank and a lower tank. The height of the water in the upper tank at time t is given by $f_1(t)$ and the height of the water in the lower tank is $f_2(t)$. The heights f_1

[0]Much of the work in this section was first reported in Hickey and Wittenberg [HW04] presented at the 2004 Hybrid Systems: Computation and Control conference

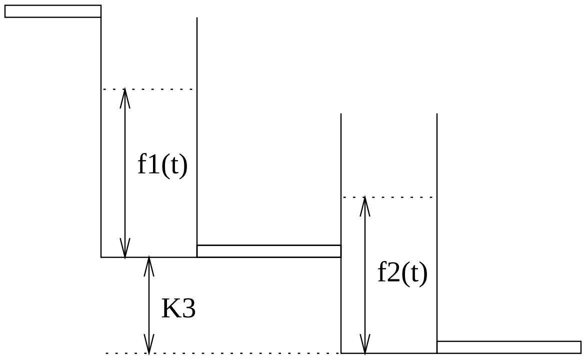

Figure 5.1: Two Tanks System

and f_2 are measured from the bottom of their respective tanks. There is a constant inflow of water into the upper tank (where the flow rate is given by a constant k_1), and a flow rate out of the bottom tank given by $k_4\sqrt{f_2}$. The bottom of the upper tank is k_3 meters above the bottom of the lower tank and there is a horizontal pipe connecting the bottom of the upper tank to the lower tank. The flow $(w(t))$ through the pipe, and the heights $f_1(t)$ and $f_2(t)$ are governed by a pair of ODEs in the constant k_2 and the water heights in the two tanks. One member of the pair holds when the water in the lower tank is below the level of the connecting pipe $(f_2(t) \leq k_3)$, the other member of the pair holds when the water level is above the connecting pipe $(f_2(t) > k_3)$. When the water level is equal to the height of the connecting pipe, the ODEs are the same, so we choose one arbitrarily. Because the initial conditions have the water level in the upper tank above the water level in the lower tank, and flow into the upper tank and out of the lower tank, the level in the lower tank never gets above the level in the upper tank $(f_1 + K_3 \geq f_2)$.

These ODEs are:

$$f_1' = \begin{cases} k_1 - k_2\sqrt{f_1 - f_2 + k_3} & f_2 > k_3 \\ k_1 - k_2\sqrt{f_1} & f_2 \leq k_3 \end{cases}$$

$$f_2' = \begin{cases} k_2\sqrt{f_1 - f_2 + k_3} - k_4\sqrt{f_2} & f_2 > k_3 \\ k_2\sqrt{f_1} - k_4\sqrt{f_2} & f_2 \leq k_3 \end{cases}$$

$$w(t) = \begin{cases} k_2\sqrt{f_1 - f_2 + k_3} & f_2 > k_3 \\ k_2\sqrt{f_1} & f_2 \leq k_3 \end{cases}$$

5.2 Rigorous Simulation of Two Tanks Problem

In this section, we give the complete CLP(F) program describing the two tanks problem, and show how it can be used to rigorously model the behavior of this system.

The program consists of two parts. The first part (Figure 5.2) describes the relation between the heights of the waters in the two tanks at two times t_0 and t_1. There are four cases considered:

- case 1: the lower tank's water level is above the pipe throughout the interval $[t_0, t_1]$

- case 2: the lower tank's water level is below the pipe throughout the interval $[t_0, t_1]$

- case 12: the lower tank's water level is above the pipe at time t_0 and stays above until some point t_2 , at which it is equal to the height of the lower pipe, and then remains below the pipe until time t_1.

- case 21: the symmetric case, where the water level in the lower tank rises from t_0 to t_1 and is equal to the height of the pipe at exactly one time t_2.

Note that there are other possible behaviours not covered by these four cases, such as when the system switches state more than once in a given time period. The code is a straightforward representation of these cases. We now walk through `case1`. The other cases are analogous. The predicate `twotank` states the following:

`twotank(Case1,X10,X20,T0,X11,X21,T1,[K1,K2,K3,K4])` is true if and only if `X10,X20,T0,X11,X21,T1` are real numbers with $\texttt{T0} \leq \texttt{T1}$, and $\texttt{X10} + \texttt{K3} \geq \texttt{X20}$, $\texttt{X11}+\texttt{K3} > \texttt{X21}$, and $\texttt{X10} > .00001$, and

- There exist two functions `X1` and `X2` defined on the domain $[T0, T1]$, (`decls([X1,X2],function(T0,T1)),`)

- the first derivative of `X1` is equal to $K2 \cdot \sqrt{X1 - X2 + K3}$ (`ddt(X1,1) = K1 - K2*psqrt(X1-X2+K3)`)

- the first derivative of `X2` is equal to $K2 \cdot \sqrt{X1 - X2 + K3} - K4 \cdot \sqrt{X2}$ (`ddt(X2,1) = K2*psqrt(X1-X2+K3) - K4*psqrt(X2)`)

- $X1(T0) = X10,\ X1(T1) = X11,\ X2(T0) = X20,\ X2(T1) = X21$ (`eval(X1,T0)=X10, eval(X1,T1)=X11, eval(X2,T0)=X20, eval(X2,T1)=X21`)

- the range of X1 is $[.0000001, 1000]$ and the range of X2 is $[K3, 1000]$ (`X1 in [E,1000], X2 in [K3,1000], E=0.0000001`)

The `{[ ]}` pair around the last five lines defines those lines as function constraints, and CLIP does as much narrowing as possible based on those constraints. The final

"." ends the assertion.

We use the range constraints `X2 in [K3,1000]` to specify that the height of the water in the second tank is always above `K3`. The lower bound is there as a condition of being in `case1`, while the upper bound is specified to be 1000, which is far above the (assumed) height of the tank. Providing a finite upper bound on a function is often necessary in CLP(F), as without it CLP(F) is unable to bound the higher derivatives, and therefore unable to start narrowing. Note that the problem of finding the transition point t_2 is automatically handled by the underlying CLP(F) system by simply adding the constraint `X2a=K3` in `case21`.

The second part of the program is an iterator (Figure 5.3) that repeatedly steps through the time domain applying the appropriate case (or when nondeterminism is present, cases) to compute the current water levels in the two tanks. The logical semantics of iterate is: `iterate(N,_DT,X10,X20,T0,X10,X20,T0,Ks)` is true if and only if the two tank system described by the parameters `Ks` will go from the state where the two tanks have depths `X10,X20` respectively at time `T0` to the state where they have depths `X11,X21` respectively at time `T1` in `N`$\geq$ 0 time steps of length `DT` where each time step can be classified as one of the four cases listed above. Note that `iterate` is false if two or more transitions take place in one time step of size `DT`. This limitation is removed in the version of the program in the next chapter.

There are three assertions for `iterate`. The first states that `iterate` for a negative number of steps fails. The second says that `iterate` for zero steps requires that the values of `X1, X2,` and `T` don't change. The third assertion is the recursive step. It states that the inductive hypothesis that a path of `N` steps exists if there is a single step followed by a path of `N` -1 steps. The `contract_vars` construct is a time

```
twotank(case1,X10,X20,T0,X11,X21,T1,[K1,K2,K3,K4]) :-
        decls([X1,X2],function(T0,T1)),
     {[  ddt(X1,1) = K1 - K2*psqrt(X1-X2+K3),
         ddt(X2,1) = K2*psqrt(X1-X2+K3) - K4*psqrt(X2),
         eval(X1,T0)=X10,   eval(X1,T1)=X11,
         eval(X2,T0)=X20,   eval(X2,T1)=X21,
         X1 in [E,1000],         X2 in [K3,1000], E=0.0000001
    ]}.

twotank(case2,X10,X20,T0,X11,X21,T1,[K1,K2,K3,K4]) :-
 decls([X1,X2],function(T0,T1)),
      {[ ddt(X1,1) = K1 - K2*psqrt(X1),
         ddt(X2,1) = K2*psqrt(X1) - K4*psqrt(X2),
         eval(X1,T0)=X10,   eval(X1,T1)=X11,
         eval(X2,T0)=X20,   eval(X2,T1)=X21,
         X1 in [E,1000],      X2 in [E,K3]
    ]}.

twotank(case12,X10,X20,T0,X11,X21,T1,Ks) :-
   {T0=<Ta, Ta<T1},Ks=[_,_,K3,_],{X2a=K3},
   twotank(case1,X10,X20,T0,X1a,X2a,Ta,Ks),
   nl,nl,print(case12(X1a,X2a,Ta)),nl,nl,
   twotank(case2,X1a,X2a,Ta,X11,X21,T1,Ks).

twotank(case21,X10,X20,T0,X11,X21,T1,Ks) :-
   {T0=<Ta, Ta<T1},   Ks=[_,_,K3,_],{X2a=K3},
   twotank(case2,X10,X20,T0,X1a,X2a,Ta,Ks),
   nl,nl,print(case21(X1a,X2a,Ta)),nl,nl,
   twotank(case1,X1a,X2a,Ta,X11,X21,T1,Ks).

   % equilibrium is at  X10=0.625,  X20=0.5625,
ks([K1,K2,K3,K4]) :-   K2=1, K4=1, % sqrt(meters)/second
                                      K3=0.5, % meters
                                      K1= 0.75. % meters/sec
```

Figure 5.2: CLP(F) code for Case1, Case2, and transitions between them

```
iterate(N,_DT,X10,X20,T0,X10,X20,T0,_Ks) :-   {N<0},fail.
iterate(N,_DT,X10,X20,T0,X10,X20,T0,_Ks) :-   {N=0}.
iterate(N,DT,X10,X20,T0,X11,X21,T1,Ks) :-
       {T1a=T0+DT,   N1=N-1},
        contract_vars([X1a,X2a,T1a],
        twotank(_Case,X10,X20,T0,X1a,X2a,T1a,Ks)),
        iterate(N1,DT,X1a,X2a,T1a,X11,X21,T1,Ks).
```

Figure 5.3: CLP(F) code for iterating to find a fixpoint

and space optimization that doesn't change the declarative semantics of the clause. `contract_vars` solves the constraint and contracts the intervals of the variables listed in its first parameter. Then it cleans the stack of all the temporary constraints used. This saves a great deal of space at the cost of losing some constraints which may have contributed to further narrowing.

This program makes the assumption that the water level does not cross the height of the pipe more than once in any DT interval. We could handle this by making the program a little more complex, but for presentation purposes we stick to this simple case for now. (We would need to use an adaptive step size when switching from case 1 to case 2 or back).

This program can now be executed by loading it into the CLP(F) interpreter and posing queries. For example, in Figure 5.4 we show the (slightly edited) output results of a query that rigorously follows the water levels over a period of two seconds with 0.1 second steps. Note that it finds the transition point from case 2 to case 1 automatically. The calculation took 53.5 seconds on a 500 MHz Pentium III running Linux.

```
| ?- reset_clip, ks(Ks),iterate(N,0.1,0.75,0.375,0,  X,Y,T, Ks).

N = 0  X = 0.75  Y = 0.375 ? ;
N = 1  X = .738726862085376...  Y = .399047107506... ? ;
N = 2  X = .7280907797217...    Y = .420650585576... ? ;
N = 3  X = .7180600004968...    Y = .44006784395... ? ;
N = 4  X = .7086039345668...    Y = .45752125423... ? ;
N = 5  X = .69969315162...      Y = .47320505137... ? ;
N = 6  X = .691299373883...     Y = .4872904076... ? ;
N = 7  X = .6833954653...       Y = .4999292640... ? ;

case21(X = .68335011672..., Y = [.49999, .50000], T = .7005915275...)

N = 8  X = .67628864233...      Y = .5109318083... ? ;
N = 9  X = .6702047371...       Y = .5202036733... ? ;
N = 10 X = .664998108...        Y = .52800756... ? ;
N = 11 X = .660542112...        Y = .534567665... ? ;
N = 12 X = .656727105...        Y = .540075051... ? ;
N = 13 X = .6534585...          Y = .54469235... ? ;
N = 14 X = .6506551...          Y = .54855781... ? ;
N = 15 X = .6482472...          Y = .5517887... ? ;
N = 16 X = .646175...           Y = .5544847... ? ;
N = 17 X = .644388...           Y = .5567299... ? ;
N = 18 X = .642844...           Y = .558595... ? ;
N = 19 X = .64150...            Y = .560142... ? ;
N = 20 X = .6403...             Y = .561420... ?
```

Figure 5.4: CLP(F) results showing transition between cases

5.3 Rigorous Analysis of the Two Tank System

Using simulation techniques, as we do, the analysis of a system can be no better than the simulation of that system. The analysis techniques we describe here are essentially applications of the simulation techniques from the previous section.

The program considered in the Section 5.2 can be used to prove properties of the two tanks system. In this section we show how to prove the following safety property, which states that if the tank levels are ever sufficiently close to an "equilibrium" point, then they stay relatively near that point forever, more precisely:

If the tank levels X_0 for the upper tank and Y_0 for the lower tank satisfy

$$0.62 \leq X_0 \leq 0.63 \quad \wedge \quad 0.558 \leq Y_0 \leq 0.567$$

at time 0, then for all times t in the future the tank levels X and Y satisfy

$$0.61922 \leq X \leq 0.63083 \quad \wedge \quad 0.55674 \leq Y \leq 0.56815$$

We prove this in two steps. First we prove that if the tank levels are in the initial interval $[0.62, 0.63] \times [0.558, 0.567]$ at time 0, then they are also in that interval at time 0.1. This implies that they are in that interval at time $N * 0.1$ for all integers N. We can use this inductive proof because the dynamics of the system depend only on the starting point, and not on the absolute time. Next we prove (Fig. 5.7) that if they start in the given interval at time 0, then they are in the second stated interval at all times t with $0 \leq t \leq 0.1$. This proves the safety property.

The first part can be proved directly by using the `solve_clip` solver which provides increasingly more precise bounds on the answer constraint as shown in Figure 5.5. This corresponds to the standard interval arithmetic ODE solving approach. `fwchk` is a "forward checking" solver, that is a divide and conquer solver which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in the list it is given, and applies the default narrowing procedure. Any boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes. On a 500 MHz Pentium III, it takes over 15 minutes to prove this directly. Of that, the N=5 case took 11 minutes.

Another approach to proving the first part is to use constraints and try to find an initial point (X, Y) such that after 0.1 seconds it is "out of the box". This is specified by the query in Figure 5.6. As can be seen, this returns with a "no" answer, which means no such point exists and hence all such (X, Y) must end up inside the "box". The calculation takes about 1.3 seconds and is more elegant than the direct approach.

The second part of the proof, involves computing the range of possible values of (X, Y) over the interval $[0, 0.1]$ assuming they start in the specified box. This is done in just under one second by making the query in Figure 5.7. By giving a range for T this query returns a constraint on the range of each variable over the entire range $[0, 0.1]$ of T, as well as whatever uncertainty the calculation includes. Again, the ubiquity of intervals adds power in unexpected ways.

```
| ?-{X0 = [0.62,0.63],Y0=[0.558,0.567]},
        ks(Ks), twotank(case1,X0,Y0,0.0,X,Y,0.1,Ks),
        solve_clip(fwchk,[X,Y],N),get_bounds(X,Lx,Hx),
        get_bounds(Y,Ly,Hy).


N = 0  X = [.61931, .63069]     Y = [.55697, .56802];
N = 1  X = [.61964, .63035]     Y = [.55758, .56741];
N = 2  X = [.61985, .63013]     Y = [.55796, .56703];
N = 3  X = [.61995, .63004]     Y = [.55812, .56687];
N = 4  X = [.62000, .62999]     Y = [.55819, .56680];
N = 5  X = [.62002, .62997]     Y = [.55822, .56677]

(929350 ms) yes
| ?-
```

Figure 5.5: IA direct proof of safety property

```
| ?- {X10 = [0.62,0.63],X20=[0.558,0.567]},
     ks(Ks), twotank(case1,X10,X20,0.0,X11,X21,0.1,Ks),
 ({X11<0.62} ; {X11>0.63}; {X21<0.558}; {X21>0.567}).

(1330 ms) no
| ?-
```

Figure 5.6: Safety Property Proof via negative answer

```
|?- {T = [0,0.1]}, {X10 = [0.62,0.63],X20=[0.558,0.567]},
    ks(Ks), twotank(case1,X10,X20,0.0,X,Y,T,Ks).

T = (0,0.10000000000000001942890293094)
X = [.61924, .63076]
Y = [.55697, .56802] ?

(900 ms) yes
| ?-
```

Figure 5.7: computation of range over $[0, 1]$ in Safety analysis

5.3.1 Multi-step Convergence

Henzinger *et al.* [HHWT98] analyzes reachability by using over approximation. Consider the space defined by the real variables. All calculations are done on rectangles in that space, such that the rectangle completely contains the region that it is modeling. By maintaining a *frontier* (a list of areas which have not been explored, but which border on the areas which are reachable) and a list of areas which are reachable, one can explore areas to determine what areas can be reachable, and not have to consider any area more than once.

When trying to determine whether a system started in a specific state $S0$ avoids certain areas, one can try to show that the system converges on a state $S1$, which has the appropriate safety property. One could keep a set of reachable states (actually an area of state space which the intervals cover) Each simulated step might add a new region to the reachable states, and one could stop adding new regions when adding a region did not increase the reachable states set. Testing to see whether a region being added actually changes the set is a computational geometry question. We have some algorithms (which we have not yet implemented) which seem fairly efficient.

Chapter 6

Scaling up to Realistic Systems

In the previous two chapters we modeled two rather small systems, and used CLP(F) to ask both "how does this system behave?", and "what parameters will cause the system to behave in a different way". In this chapter we show that we can scale up a model in CLP(F). We add complexity in several forms. The simpler is to have a larger system. We move from a system with two tanks to one with four tanks, and we add valves to the pipes connecting the tanks. The more important is the refinement of the model in several places. We use a valve model with the flow varying exponentially with the valve position over much of the valve's range, and then discontinuously as the valve is almost closed. We introduce hysteresis in our analysis to avoid an infinite loop of zero-time transitions, and we discuss why our techniques should not have trouble with "Zeno" transitions.

We start by adding valves to the model. We note that the model in Kowalewski *et al.* has the behaviour of the valve discontinuous at 0 (by 5% of full flow), and show how a broad constraint handles that. In a hybrid system, the interface between the analog and the digital part involves imperfect hardware whose description must include error bars. The models of system behaviour are often particularly imprecise near boundaries. We use intervals to handle the issue of imprecision in measurements, and use intervals in a novel way to rigorously model the behaviour of systems near boundary points.

6.1 Extending Two Tanks to N Tanks

In Section 5.1, we showed how CLP(F) could model the simplified two tanks problem. In this chapter, we show how the CLP(F) model can easily be extended to the "tank flow problem", an extension of the two tanks system to an arbitrary number of tanks, and to do so more rigorously than other methods can. Here, we consider a four tank version with valves between each pair of tanks and at the output.

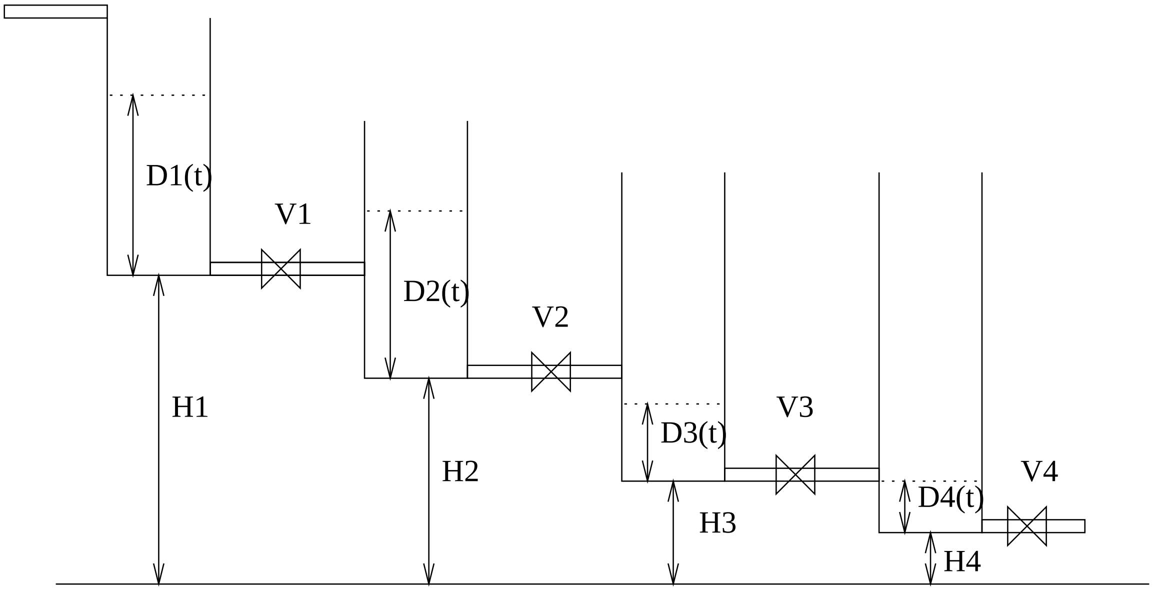

Figure 6.1: Diagram of Tank Flow system for $n = 4$

Mathematics of the Tank Flow Problem

The problem we study is diagrammed in Fig. 6.1 and the parameters and variables are shown in Table A.1. The problem can be described as follows: There are n tanks, numbered from 1 to n, with the bottom of each tank lower than the bottom of the previous tank. The depth of the water in each tank j at time t is given by $D_j(t)$ The depths D_j are measured from the bottom of their respective tanks. The height of the bottom of each tank j is H_j above an arbitrary horizontal datum, perhaps sea level. Each tank j has a horizontal pipe leaving from the bottom of the tank. The

H_j	Height of tank j above sea level
V_j	inverse of time for valve j to open or close
C_j	pipe coefficient of pipe j when the valve is fully open
E_j	exponent for describing the valve's behaviour
$P_j(t)$	position of valve j 0 is fully closed, 1 is fully open
$M_j(t)$	valve motion – closing, opening, halted
$R_j(t)$	program variable for valve regime - shut, transition, normal
$D_j(t)$	Depth of water in tank j at time t (measured from bottom of tank.)
$I_j(t)$	rate of flow through pipe j at time t

Table 6.1: Parameters and Variables

flow through that pipe is I_j, and there is a valve V_j on the pipe. There is a constant inflow of water into the upper tank (where the flow rate is given by a constant $f00$). The general equation for flow through a pipe is that the rate of flow is proportional to pipe coefficient times the square root of the height difference of the water levels at each end.

Specifically, the flow $I_j(t)$ through pipe j connecting tank j to tank $j+1$ is governed by a pair of ODEs in the resistance $R_j(t)$ to flow (which is a combination of the pipe coefficient C_j, valve coefficient E_j, and the valve position $P_j(t)$) and the pressure difference, which is proportional to the square root of the difference in water heights on each end of the pipe. The pipe coefficient C_j describes how easily water flows through the pipe when the valve is in the fully open position, while the valve coefficient E_j describes how quickly the valve shuts off the flow of water. If the water level in the lower tank is below the pipe bringing water in, there is no back pressure in the pipe, so we can ignore the water level in the lower tank. If the water level in the lower tank is higher than the input pipe, we have to include the effect of back pressure on the flow through the pipe. Therefore, we have a pair of ODEs for each pipe. One

ODE of the pair holds when the water in the lower tank $(j+1)$ is below the level of the connecting pipe $(D_{j+1}(t) \leq H_j - H_{J+1})$, the other member of the pair holds when the water level is above the connecting pipe $(D_{j+1}(t) > H_j - H_{J+1})$. When the water level is equal to the height of the connecting pipe, the ODEs are the same, so we choose one arbitrarily. Later (Section 6.3.4) we will show how to rigorously handle this point where the ODEs change, and which is therefore not analytic. Note that even if the water level is above both ends of a pipe, if the water levels (measured from sea level) are equal, the ODE is non-analytic because the square root[1] function's derivative is infinite at 0. We handle these two different reasons for an ODE to be non-analytic in exactly the same manner, described in Section 6.3.

The valve decreases the flow by a fraction which decreases exponentially with the valve position. Recall that since D_j is the depth in a tank, I_{j-1} the flow into that tank, and I_j the flow out, $D'_j = I_{j-1} - I_j$ The ODEs for middle tanks (the first and last tanks are similar) are:

$$I_j(t) =$$

$$\begin{cases} 0 & P_j(t) = 0 \\ e^{E_j \cdot (1-P_j(t))^3} \cdot C_j \sqrt{D_j(t) - D_{j+1}(t) + H_j - H_{j+1}} & D_{j+1}(t) > H_j - H_{j+1} \\ e^{E_j \cdot (1-P_j(t))^3} \cdot C_j \sqrt{D_j(t)} & D_{j+1}(t) \leq H_j - H_{j+1} \end{cases}$$

Where P_j is the position of the valve C_j is the pipe coefficient, the value under

[1]We really want a function which is the positive square root of a positive number, and the negative square root of the absolute value of a negative number to properly describe the fluid flow. This function is also not analytic at 0. Here we do not allow for later tanks to have higher water level than earlier tanks, so we do not consider "backwards" flow.

the radical is the effective difference in height between the two tank's water levels, and the exponential term is the fraction by which the valve decreases the flow.

6.2 Handling State Changes

A hybrid system of any size will have different ODEs to describe it at different times. Writing each ODE explicitly (as we did for a simpler example in Section 4) is impractical because of a combinatorial explosion in the number of ODEs. To avoid this problem, we parameterize the ODEs describing the system, so a state change is modeled by a change in some of the parameters to an ODE rather than making a different ODE active.

The ODEs governing a hybrid system can change for either of two reasons. The first is if the digital part of the system has a state change which affects the ODEs. We call this a *program control change*. The other is if the continuous system evolves in such a way as to change the ODEs, such as evolving to a point where water overflows, or the water level in a tank rises above the input pipe to that tank, causing back pressure, and decreasing the flow rate. We call these events *regime changes*. One case of regime change is when a valve that had been opening (or closing) becomes fully open (or closed). That affects the ODEs, by changing the rate at which the valve position changes, not by changing the water flow directly. A helpful feature of CLP(F) is that we can model changes in ODEs caused by program control and those caused by regime changes in exactly the same way.

Figure 6.2 is a state diagram for each valve except the last in the tank flow problem. (The last valve has no lower tank, so the level in the lower tank can't rise above the height of the pipe.) The states are described by two ternary variables, M (Valve Motion Regime) describes the motion of the valve as one of (`opening, closing, halted`), while R (Valve position regime) is one of (`shut, trans, normal`). When R takes the value `shut` it means that the valve is closed, normal means that the valve is open, and not too near the closed position. The R value `trans` means that the valve is in a transitional region and is nearly, but not quite closed. The transitional region is used to model the regime where the ODEs are not well understood, so we use a simple constraint in that regime.

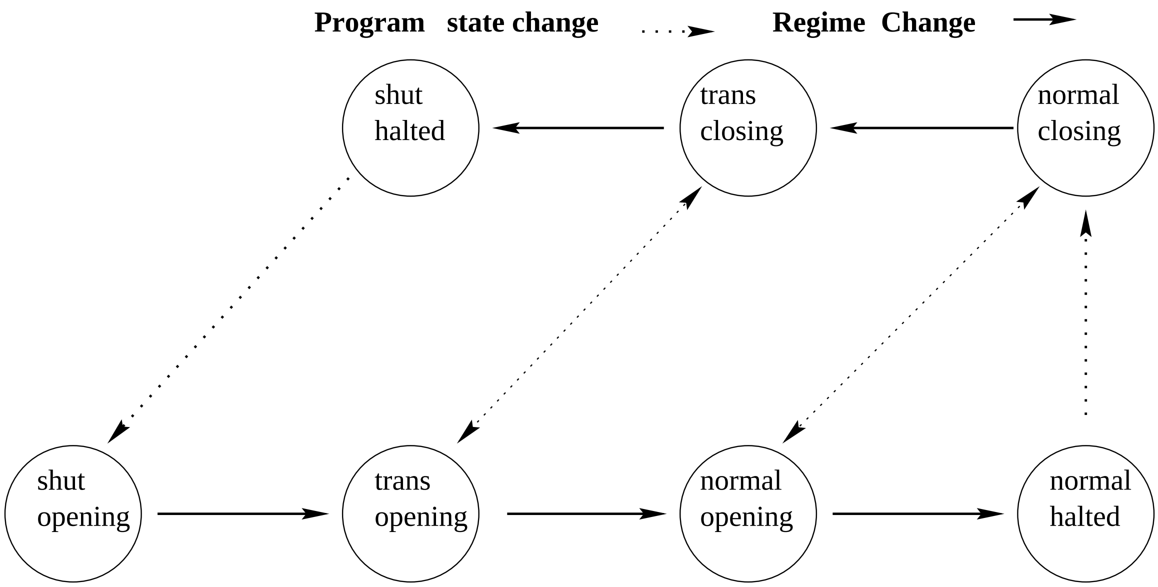

Figure 6.2: State diagram for ODEs

6.3 Handling Boundary Areas

One of the problems in rigorous modeling is that often there are areas where one's original model breaks down for some reason. This reason can be an area where the physics are unclear, a point where the defining functions are not analytic, or perhaps a function which is poorly defined at a limit point. This section deals with a single technique which allows one to deal with all of these problems.

6.3.1 Dealing with Poorly Defined Regions

In many systems, the physics in some regions is not well-understood. Most hybrid system techniques ignore this and simply assume that the ODEs which work in most areas work near boundaries as well. For example, in the tank flow problem when a tank is almost empty, the flow from it may be irregular and come in discrete drops rather than as a continuous flow. At these points, we don't claim to understand the details of the flow, but we can model them rigorously by writing constraints which clearly include any possible behaviour of the flow. We don't consider an empty tank in this case, except to constrain our description of the system to cases in which the water level in each tank is at least `E`.

A further problem is that the physics of the system may not be understood perfectly. In most cases, one measures a value (here, the flow through a valve as a function of how open the valve is) at several points, uses physical theory to decide that the curve should be an exponential, and then uses a least-squares fit to find a curve which best describes the measurements. There is, of course, error in the measurement of each point, so the coefficients for the exponential curve have some (hard to calculate) error bars. In addition, the behaviour when the valve is almost closed does not follow the exponential decay curve, and is extremely difficult to measure precisely.

For example Kowalewski *et al.* [KSF+99] describe a valve by a function $K_i(P)$

giving the pipe coefficient and valve coefficient of the valve as a measured function of how open the valve is. For the first valve, the function they give is:

$$K_1(P_1) = \begin{cases} 1.85 \cdot 10^{-4} \cdot e^{-6 \cdot 10^{-6} \cdot P_1^3} \frac{m^{5/2}}{s} & \text{if } 0 \leq P_1 < 80 \\ 0 \frac{m^{5/2}}{s} & \text{if } P_1 = 80 \end{cases}$$

and for the second valve, they give:

$$K_2(P_2) = \begin{cases} 2.26 \cdot 10^{-4} \cdot e^{-5.7 \cdot 10^{-6} \cdot P_2^3} \frac{m^{5/2}}{s} & \text{if } 0 \leq P_2 < 80 \\ 0 \frac{m^{5/2}}{s} & \text{if } P_2 = 80 \end{cases}$$

In neither case do they give error bars. Their model has 80 as a fully closed valve, and 0 for a fully open one. We describe our valves more naturally, with 0 corresponding to fully closed and 1 to fully open. Using our descriptions, we get values of $C_1 = 1.85 \cdot 10^{-4}$, $E_1 = -3.1$, $C_2 = 2.26 \cdot 10{-}4$, and $E_2 = -2.9$ for the constants describing flow. The equation for total resistance in the pipe (which they call K) is

$$R_j(t) = C_j \cdot e^{E_j \cdot (1-P_j)^3}$$

Figure 6.3 shows a graph of R vs. P for valve 1. The curve is an exponential decay, whose value when the valve is almost closed is about 5% of the flow when the valve is wide open, but they define the flow for a fully closed valve as 0. By straightforward calculation, we find that $R_1(1) \approx 1.85 \cdot 10^{-4}$, while $R_1(\varepsilon) \approx 8.570 \cdot 10^{-6}$, and $R_1(0)$ is defined to be zero. It is likely that this is not fully correct, as a discontinuity of that magnitude is not common. In any event, we can model this discontinuous point by having three constraints for three different regimes. When the valve is fully closed,

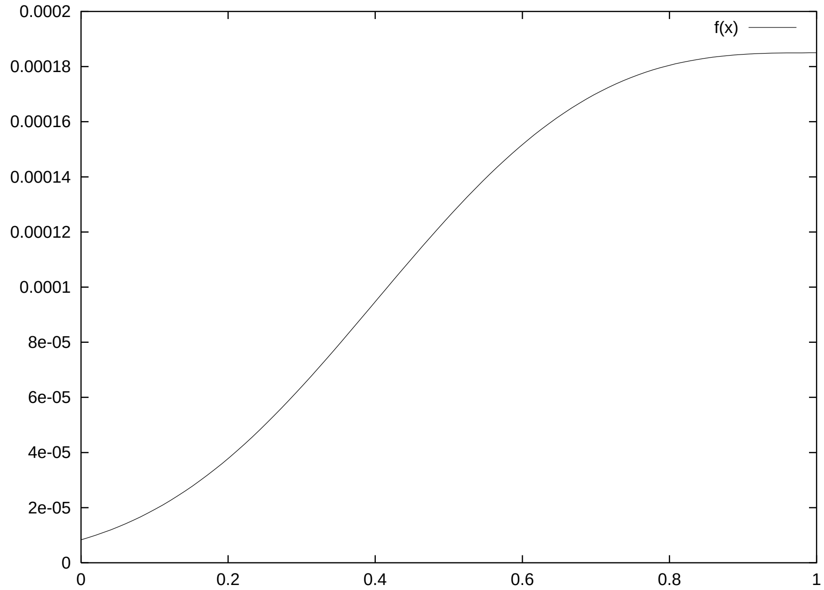

Figure 6.3: Graph of flow against valve position for valve 1

R is 0. When the valve is at position .02 or above, R is given by the ODE above. The interesting case is when the valve position is between 0.0 and 0.02. There we have a constraint which says that if the valve position is between 0 and 0.02, R is in the range $[0, 10^{-5}]$. The upper end of the range for R is chosen to be just above the calculated value of R at any point in the P's range for that region.

Figure 6.4 shows how we rigorously model this system for P near 0. (Note that this figure uses the notation of our programs, while 6.3 uses the notation of Kowalewski et al. The numbers are different, but the shapes are exactly the same.) For the part of the curve where the equations make sense, we enclose the specified curve on each side by the ODE describing the valve. Because the parameters of the ODE are intervals, the value of the curve at any point is an interval. In the area where the curve is discontinuous, we use a constraint which includes all possible values the function could take on. This introduces some uncertainty into the formal model, but

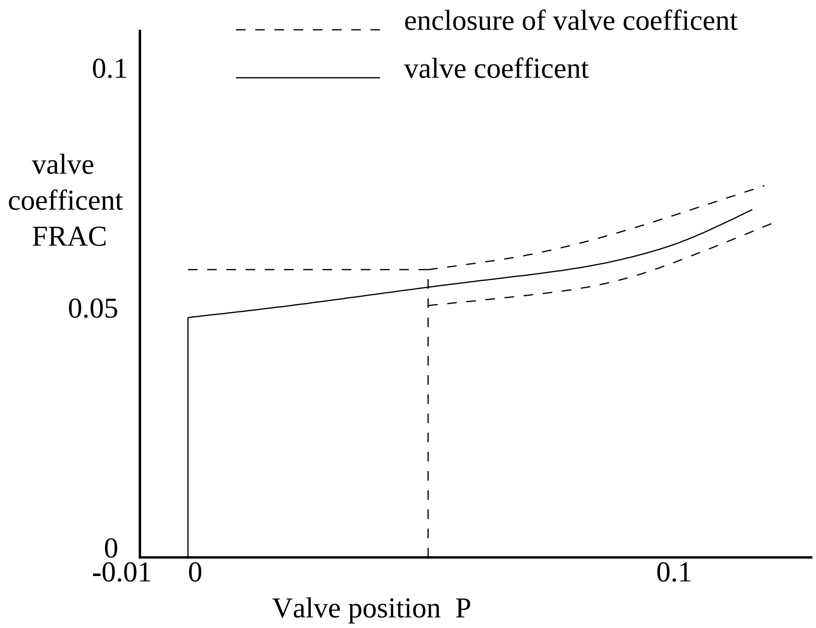

Figure 6.4: Graph of flow against valve position for valve 1 showing enclosure by simple constraint (scale greatly enlarged)

that uncertainty was already present in the description of the physical system. Using constraints makes it simple to describe that uncertainty rigorously.

6.3.2 Stepwise refinement

Ideally a modeling system allows stepwise refinement of the model. We demonstrate this in CLP(F) by adding valves to our model of the tank flow system. Adding the valves to the model was easy despite using a rather complex model of the valve's behaviour.

One problem which is rarely addressed in modeling hybrid systems is the transition regimes as a component or valve changes state. Using constraints, we can provide a rigorous answer by constraining the output of the component while it changes state to be between the output it has in one state and the output it has in the other, and keeping that constraint for however long the component takes to change state. If more precision is required, one can add a description of the components behaviour during the state transition. Since the description consists of upper and lower bounds for the component's output, one can progressively refine the bounds as one learns more about the component's behaviour.

6.3.3 Rigorous Treatment of Imprecision

An important issue in modeling hybrid systems is to realize that essentially none of the parameters are known exactly. This is true both of the parameters to the differential equations which describe the system – These parameters are often determined by curve fitting to a set of measured points or are calculated from physical models which include simplifying abstractions, and of measurements taken by sensors in the system – These are measured with some accuracy, which is often specified as an error-bar. (Note that the problem of imperfect measurement is inherent in the physical world. Heisenberg's uncertainty principle prohibits perfect measurement, and Burridan's principle [Lam86] further limits the speed at which one can usefully take measurements.) Because CLP(F) treats everything as an interval, it models these inaccuracies naturally. If a CLP(F) program shows that a system has a safety property (proves that it avoids a region), that proof is valid even when each parameter takes the worst possible value within the given error bars. To deal with this issue by sensitivity analysis [sen04, Ars, Tay97] on the inputs would be extremely difficult.

6.3.4 Dealing with Regime Change Points

One of the advantages of using CLP(F) is that one can often use one technique to handle multiple issues. In section 6.3.1 we use separate ODEs, often with rather simple-minded constraints to deal with regions where the physics is unclear. Here we use a similar system to deal with non-analytic (or even discontinuous) points in an ODE.

When the water level in the lower tank is above the input pipe (in regime `above`), one set of ODEs holds, when the level is below the input pipe (in regime `below`), another set of ODEs holds. We model this by having a regime change at that point. An obvious problem arises: Our model would allow an infinite number of transitions (each taking zero time) between the two states, and therefore never get to calculating the change in water level which would move clearly into one state (Actually, since CLPs are non-deterministic, there would be an infinite path and also a one-step path out of that condition). We handle this by creating a special artificial state `near` for water levels near the boundary. We then artificially put in hysteresis, so that on leaving that middle state, one cannot immediately re-enter it. This problem is related to the problem of Zeno automata, discussed in Section 6.5.

This example clarifies two issues, as there are two separate reasons for using `near` state between `above` and `below`. The first reason is that as the water in the lower tank reaches the level of the pipe the physics get a little unclear - what happens when the water covers half the pipe? This issue is clearer in the case where the ODEs are discontinuous, as in section 6.3.1. The second issue is that in order to model a change of ODEs, we need two regimes, with appropriate transitions between them. This issue arises even when the physics are clear, such as when one has a pipe between two tanks and the relative water levels in the two tanks is changing. At the point where the water levels are equal, the ODE is non-analytic (because the square root function is non-analytic at 0), so we would have to have a change of regimes. If the rule for a regime change was simply that the water levels were equal, when the levels became equal there would be a legal infinite path of zero-time changes from one regime to the other. One could look at the derivative to know which direction the regime change goes in, but if the water level is almost constant, the derivative will be near zero, and the same issue is still there. To avoid this case, we artificially add hysteresis to an already artificial regime change.

6.4 Overview of Code for Tank Flow

The complete program for the n=4 case of the tank-flow problem is in Appendix B. Here we discuss some of the more interesting snippets from the code.

6.4.1 Evolve and Iterate

We model a hybrid system in CLP(F) by modeling a series of steps. A step begins either at a specified initial state, or when the previous step ends, and ends when either the length of the step (amount of time simulated) reaches a maximum step size `delta`, or a change of ODEs occurs (whether caused by program control or a regime change). The following part of the program (taken from Section B.2) is the main code, which runs the system through one step, increments the state counter, and continues.

```
evolve(S0,C,N,S2) :-
```

```
  evolve0(S0,C,N,S1),
  enforce_ODEs(S1,C,S2),
  copy_discrete_state(S1,S2).
```

`evolve(S0,C,N,S2)` is true if and only if the system described by `C` can evolve to a boundary state `S1` in `N` steps and then evolve from state `S1` to state `S2`.

```
% evolve0(S0,C,N,S1) is true iff the system described by C can
% evolve in N steps from S0 to S1 *and* S1 is a boundary state
% (i.e. a program or regime changes at S1, or a maximum step length
% is achieved at S1)
evolve0(S0,_C,N,S1) :- {N=0},eqstate(S0,S1).


evolve0(S0,C,N,S2) :-
    opt_next_step(S0,C,S1),
    print(ons(S0,C,S1)),nl,nl,
    {N=M+1},
    evolve0(S1,C,M,S2).
```

A direct reading of the program is as follows: In zero steps, the system does not change state. The `evolve0` predicate says that a system can evolve from `S0` to `S2` if `S1` is the next step from `S0`, $N = M + 1$, and the system can evolve from `S1` to `S2` in `M` steps. The variable `C` in all cases is the set of constants which describe the system parameters.

```
% next_step(InitialState, ProblemConstants, FinalState)
```

```
next_step(S0,C,S1) :-
  enforce_ODEs(S0,C,S1),
  find_state_change(S0,C,S1).
```

The call to `next_step` states that the ODEs are followed (`enforce_ODEs`), and finally that system has run to an appropriate point (`find_state_change`). All the variables in the states (`S0, S1`) and the constants term (`C`) are variables over the reals. Variables over functions are used in `enforce_ODEs` to specify constraints over the real variables in `S0, C, S1`.

6.4.2 Finding State or Regime Changes

In each case, the step ends when any of the requirements becomes true. Figure 6.5, taken from section B.4, shows how `find_state_change` is defined to be true when any one of the following happen:

- One of the `find_flow_state_change` predicates becomes true because the water level in one of the tanks goes from above the input pipe in state `S0` to below in state `S1`, or vice versa (one of the tanks changes regime)
- one of the `find_valve_state_change` predicates becomes true, because the valve position is such that a change in regime occurs at state `S1`
- one of the `find_program_state_change` predicates becomes true because the program (ie. the digital part of the hybrid system) changes state at state `S1`
- `find_step_change` is true because state `S1` is `Delta` time after state `S0` and no other state changes have occurred.

```
find_state_change(S0,C,S1) :-
% TANK FLOW REGIME CHANGE
         find_flow_state_change(r1,h1,h2,d2,C,S0,S1);
         find_flow_state_change(r2,h2,h3,d3,C,S0,S1);
         find_flow_state_change(r3,h3,h4,d4,C,S0,S1);

% VALVE REGIME CHANGE
          find_valve_state_change(p1,vr1,vm1,C,S0,S1);
          find_valve_state_change(p2,vr2,vm2,C,S0,S1);
          find_valve_state_change(p3,vr3,vm3,C,S0,S1);
          find_valve_state_change(p4,vr4,vm4,C,S0,S1);

% PROGRAMMED STATE CHANGES
          find_program_state_change(v1,d2,C,S0,S1);
          find_program_state_change(v2,d3,C,S0,S1);
          find_program_state_change(v3,d4,C,S0,S1);

%  no regime or state changes before the time limit is reached
          find_step_size_change(S0,C,S1).
```

Figure 6.5: Code to Find State Changes

The CLP(F) code to check for this (excerpted in Figure 6.5 from Section B.4) looks rather repetitive. This is true only because in this example we use the same behaviour for each valve and for each tank. In a less symmetric case, this code would not grow, but there might have to be multiple versions of `find_??_state_change` to describe the different behaviours.

6.4.3 Enforcing ODEs

Section B.3 describes all of the analog parts of the system. It consists of three large assertions. The first (and largest) Section B.3.1 is purely bookkeeping. All of the ODEs are in the last two parts of `enforce_ODEs`. In order to make the lists of parameters smaller, we use lists to keep all variables of each type together. `lookup`, `evalall`, and `decls` are helper functions defined in section B.7 to deal with the lists.

The bookkeeping section states that the individual variables correspond to what the lists say they are, and constrains the domain and range of the functions. It uses `lookup` to bind the values of constants (from `C`), and conditions at the start of the step (from `S0`), and the end of the step (from `S1`) to variables. Then it uses `decls` to declare several function variables (and their domains) at once, and finally specifies which ODEs each tank should obey while in the state specified by `S0`. Figure 6.6 shows sections of the first part of `enforce_ODEs`. Much of that section is repetitive, so only representative fragments are reproduced here. We interpret the code as follows: `enforce_ODEs` is true if and only if all of the following elements are true (including, of course, those that are elided here.)

- `P` is a vector containing `P1,P2,P3,P4`

- each element of P is a function defined on $[\mathrm{T0}, \mathrm{T1max}]$
- each element of P is a function whose range is $[0, 1]$
- Ps0 is a vector containing P10,P20,P30,P40
- Ps1 is a vector containing P11,P21,P31,P41
- the values of Ps0 are as specified in C in state S0
- the values of Ps1 are as specified in C in state S1
- evalall each element in the list P evaluated at T0 gives the corresponding value from Ps0, and when evaluated at T1 gives the corresponding value from Ps1
- the value of v in the list of constants C is V
- each of the valves obeys valve_ODE given the valve position, velocity, and motion regime
- each of the tanks obeys the appropriate tank ODE

Section B.3.2 handles the flow restriction caused by the valves.

```
valve_coef(normal,FRAC,P,E) :- {[FRAC=exp(E*((1-P))**3),
                               FRAC in [0,1],  P in [0.01,1] ]}.
valve_coef(trans,FRAC,P,_) :- {[ FRAC in [0,0.06],  P in [0,0.01] ]}.
valve_coef(shut,FRAC,P,_) :- {[FRAC=0.0*FRAC, P=0*P ]}.

valve_ODE(P,_,halted) :- {[ ddt(P,1) =   0.0*P, P in [0,1] ]}.
valve_ODE(P,V,opening) :- {[ ddt(P,1) =  V+0*P, P in [0,1] ]}.
```

```
enforce_ODEs(S0,C,S1) :-

...

  % VALVE Position ODEs
  % create valve position functions on [T0,T1max]
    P=[P1,P2,P3,P4],
    decls(P,function(T0,T1max)),
  % put bounds on the range of the function
    bound_functions(P,[0,1]),
  % set their values at times T0 and T1
    Ps0=[P10,P20,P30,P40],
    Ps1=[P11,P21,P31,P41],
    lookup([p1=P10,p2=P20,p3=P30,p4=P40],S0),
    lookup([p1=P11,p2=P21,p3=P31,p4=P41],S1),
    evalall(P,T0,Ps0), evalall(P,T1,Ps1),
  % lookup the valve speed
    lookup([v=V],C),
  % add the ODE constraints
    valve_ODE(P1,V,M1),
    valve_ODE(P2,V,M2),
    valve_ODE(P3,V,M3),
    valve_ODE(P4,V,M4),
...

  % apply the ODEs corresponding to each tank
  % Ri = ode governing tank i, Di = depth in tank i,
  % Fi = flow out of tank i, Hi = height of tank i,
  % Pi = valve opening out of tank i, Ki = valve coefficient,
  % F00 = flow into tank 1

      first_tank( R1,    D1,F1,D2,  F00,H1,C1,FRAC1,H2,E),
      middle_tank(R2, F1,D2,F2,D3,      H2,C2,FRAC2,H3,E),
      middle_tank(R3, F2,D3,F3,D4,      H3,C3,FRAC3,H4,E),
      last_tank(      F3,D4,F4,            C4,FRAC4).
```

Figure 6.6: Parts of Enforce ODEs code

```
valve_ODE(P,V,closing) :- {[ ddt(P,1) = NV + 0*P, P in [0,1],
                                                     NV= - V ]}.
```

The ODE code is completely straightforward, as ODEs can be described directly in CLP(F). `FRAC`, `E`, and `P` are all functions of `T`. The first line says that in the valve regime `normal`,

$$\mathrm{FRAC} = e^{E\cdot(1-P)^3},\ \ \mathrm{FRAC} \in [0,1],\ \ P \in [0.01,1]$$

The second line says that in valve regime `trans` FRAC $\in [0, 0.06]$ and $P \in [0, 0.01]$. The third line says that flow through a shut valve is 0. The idiom `FRAC=0.0*FRAC` is a workaround used instead of `FRAC=0` because CLIP does not allow functions to be set equal to a constant. The second line of the code is needed to implement the technique of rigorously modeling discontinuous functions discussed in section 6.3.1. Observe that this procedure constrains `P` to take values inside the appropriate region (for `normal, trans, shut`).

Similarly, the last three lines specify the derivative of `P` (the valve position) to be 0 when halted, `V` for opening, and `-V` for closing.

The last assertions in the ODE section specify the flow into and out of tanks. There are seven cases (section B.3.3), as the first and last tanks have different configurations than tanks in the middle, and for all but the last tank, the ODEs differ according to which regime the tanks is (among `below`, `near`, and `above`) corresponding to whether the water level in the lower tank is above or below the pipe entering the lower tank. We consider the case of a middle tank in regime `above`, as that is the most complex.

```
middle_tank(above,F1,D2,F2,D3,H2,C2,FRAC2,H3,_E) :-
```

```
{[F2=C2*FRAC2*psqrt(D2-D3+H), ddt(D2,1)=F1-F2, H=H2-H3,
                                          D3 in [H,1000] ]}.
```

This says that given a middle tank 2 (middle tank here means that tank 2 is not the first tank, and tank 3 is not the last tank) in regime `above` with the following parameters:

`F1`	flow into the upper tank
`D2`	water height of the upper tank
`F2`	flow out of the upper tank (into the lower tank)
`D3`	water height of the lower tank
`H2`	height of the upper tank above sea level
`C2`	parameter of flow through the pipe between upper and lower tanks
`FRAC2`	fraction of the maximum flow the valve allows
`H3`	height of the lower tank above sea level
`_E`	an error term (the underscore before the `E` means ignore this term .)

then:

$$F2 = C2 \cdot \mathrm{FRAC2}\sqrt{\mathrm{D2} - \mathrm{D3} + \mathrm{H}}, \ \ \frac{d\mathrm{D2}}{d\mathrm{T}} = \mathrm{F2} - \mathrm{F1}, \ \ \mathrm{H} = \mathrm{H2} - \mathrm{H3}, \ \ \mathrm{D3} \in [\mathrm{H}, 1000]$$

Here `H2, C2, H3` and `E` are constants, and all the other variables are function variables, though that must be implied from earlier declarations. Again note that the the constraint requires the depth `D3(T)` to be in the region for the `above` case or on the boundary with another case.

6.4.4 Finding State Changes

The last section of code (Section B.6) we describe in detail determines that a regime change has occurred. Parts of this code are in Figure 6.7 and Figure 6.8. This code is called from `find_state_change` (Section B.4), which says that `find_state_change` is true if at least one of `find_flow_state_change`, `find_valve_state_change`, `find_program_state_change` or `find_step_change`, is true.

```
% Detection of regime change due to tank depth exceeding input
% pipe height  e.g. find_flow_state_change(r2,h2,h3,d3,C,S0,S1).
% note j=i+1 here and Ri in {above,near,below}

find_flow_state_change(Ri,Hi,Hj,Dj,C,S0,S1) :-
  lookup([Hi=H1,Hj=H2],C),  lookup([Dj=D],S1),
  update_discrete_state(Ri,R_before,R_after,S0,S1),
  flow_state_change(R_before,D,R_after,H1,H2).

% We use hysteresis in our analysis to avoid an infinite loop of zero
% time state changes as it goes from near to below and back again.
flow_state_change(below,D,near,H1,H2) :-
  E=0.00001,  {D = H1-H2-E}.
flow_state_change(near,D,below,H1,H2) :-
  E=0.00001,  {D = H1-H2-2*E}.
flow_state_change(above,D,near,H1,H2) :-
  E=0.00001,  {D = H1-H2+E}.
flow_state_change(near,D,below,H1,H2) :-
  E=0.00001,  {D = H1-H2+2*E}.
```

Figure 6.7: Code for Regime Change as Water Level Changes

`find_flow_state_change` (Figure 6.7) is true if and only if the two `lookup` assertions are true, `update_discrete_state` is satisfied, and `flow_state_change` is satisfied. The `lookup` assertions state that the values of constants passed to the assertion match the constants stored in `C`. `update_discrete_state` here states that the only difference in discrete variables between state `S0` and state `S1` is that in state `S0`, `Ri` has value `R_before` and in state `S1`, `Ri` has value `R_after`.

`flow_state_change` lists the four possible transitions, and the water levels which allow them. Note the hysteresis – to enter state `near` the water level has to be within `E` of the critical level (`H1 - H2`), while to leave state `near` the water level has to be `2*E` away from the critical level. This is to prevent an infinite sequence of zero-time transitions when the water level is at a critical point.

Figure 6.8 shows the code for changes in the valve's regime. `find_valve_state_change` is very similar to the code for `find_flow_state_change`, except that it twice calls `update_discrete_state` to update the two ternary variables for the two sets of regimes a valve has. One (`M`) is the valve motion regime, which can be one of `opening, halted, closing`, the other (`R`) is the valve position regime, which can be one of `shut, trans, norm`. The valve position regime is necessary because of the discontinuity in the valve ODEs at zero. `norm` means that the valve is in the regime where the standard ODE applies, `shut` means that the valve is fully closed, and there is no flow through it, and `trans` is the transition regime, where we simply apply a coarse constraint because we don't understand the physics in that regime.

6.5 Zeno hybrid systems

Johansson *et al.* [JELS99] introduce what they call a "Zeno phenomenon". This is a problem with some hybrid models in which an infinite number of steps occur in a finite amount of time. At best, this leads to calculations which never finish, while at worst, it leads to false proofs of safety properties in systems which don't have those

```
% check to see if valve n has hit a state change
% and if so, update the discrete part of S1 accordingly
% e.g. find_valve_state_change(p2,v2,C,S0,S1).
% v2 in {opening,closing,halted}, p2 in [0,1],
% note that this is a regime change, not a state change.
% Also, we have to handle the regime change from shut to transition
% to normal. The transisition to shut implies a transition to halted,
% but not vice versa.

find_valve_state_change(Pn,Rn,Mn,_C,S0,S1) :-
% use S2, as temp states to have 2 discreet vars change
  lookup([Pn=P_before],S0),lookup([Pn=P_after],S1),
  update_discrete_state(Mn,M_before,M_after,S0,S2),
  update_discrete_state(Rn,R_before,R_after,S2,S1),
  valve_state_change(M_before,R_before,P_before,M_after,
                                            R_after,P_after).


% regime change rules for valve motion (and in closing case, position)
valve_state_change(opening,normal,_P_before,halted,normal,P_after) :-
                                            {P_after=1}.
valve_state_change(closing,trans,_P_before,halted,shut,P_after) :-
                                              {P_after=0}.

% regime change rules for valves position
valve_state_change(opening,trans,_P_before, opening,normal,P_after) :-
                                                     {P_after=0.01}.
valve_state_change(opening,normal,_P_before,constant,normal,P_after):-
                                                     {P_after=1.0}.
valve_state_change(opening, shut,_P_before,  opening,trans,P_after) :-
                                                     {P_after=0.0}.
valve_state_change(closing, normal,_P_before,closing,trans,P_after) :-
                                                      {P_after=0.01}.
valve_state_change(closing,trans,_P_before,  constant,shut,P_after) :-
                                                       {P_after=0.0}.
```

Figure 6.8: Code for Valve Regime Changes

properties. The canonical examples of Zeno phenomena are a bouncing ball which with each bounce achieves some fraction of the height of the previous bounce in a fixed fraction of the time, and a water tank example discussed below. In the bouncing ball case, a simulation would have to calculate an infinite number of bounces before terminating unless the model included some handling of the idea that when the height of each bounce is less than one atom's diameter, the model must change.

Johansson *et al.* note that the Zeno phenomenon usually occurs as a result of over abstraction in the model, as happens in these cases. Real systems can have valves that chatter, but the chattering cannot involve an infinite number of state changes in a finite time. It the real system has chatter, one should model it by a constraint giving a minimum time for a valve to change state. The infinite chattering is an artifact of some models, and should be removed by the modeler. Zhang *et al.* [ZJLS01] give examples of cases where overly abstract models (with the Zeno property) of real systems (without the Zeno property) lead to incorrect proofs of safety properties. In most cases, the Zeno problem can be eliminated by a more accurate model, often by simply modeling the time a valve or switch takes to change state. In our four tanks example, we avoid Zeno phenomena because there is a lower bound on the time required for twelve consecutive state changes. This bound is implied in different ways for different sets of state changes. For example, the water flow through any pipe is proportional to the square root of the water height, and we bound the water height in each tank. That limit on the water flow limits how quickly the water level in any tank can change. The only code added to avoid Zeno phenomena is the hysteresis in Section B.6. One case in which we do not avoid Zeno phenomena is if the discrete part of a hybrid automata describes a Zeno phenomena. If, for example, the program specified that at some water level a valve would switch from open to closed and from closed to open, that behaviour would be modeled, and the simulation might never finish. There is nothing to be done here. If a user specifies a poorly-formed program, analysis may fail.

The water tanks example of Johansson *et al.* is shown in Figure 6.9. There is a flow of water i into a valve which can direct the water into either of two tanks. Each tank has a water level (h1, h2), and a required level (r1, r2). The safety property is that h1 is always above r1, and h2 is always above r2. The water flow out of each tank is proportional to the ratio of the area of the tank to the area of the output pipe, and to the square root of the water height. If the input flow i is chosen to be larger than either output flow $o1$ or $o2$, but less than their sum (when h1 and h2 are near r1 and r2), it is clear that the level in at least one of the tanks must fall below its required level. Consider the program which whenever one of tanks gets to its required level switches the flow to that tank. As the water level gets lower, the switching will happen more and more often, and the valve will switch an infinite number of times in a finite period, during which time the water level in each tank will still be at or above the required level.

6.5.1 CLP(F) and Zeno systems

How does a CLP(F) model handle a Zeno system? Consider the bouncing ball first. If the modeler does not note that the physics change for very small bounces, the simulation has to include an infinite number of vanishingly small bounces, but because everything in CLP(F) is an interval, the height of the bounce will at some point reduce to $[0, S]$, where S is the smallest number representable in the floating point system.

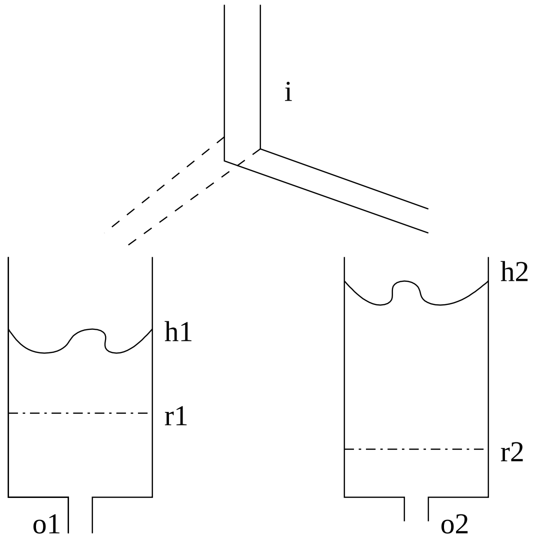

Figure 6.9: Flow system with Zeno behaviour (after Zhang *et al.* [ZJLS01])

CLP(F)'s non-determinism means that it should eventually explore the path where the bounce height is 0, and the motion ends. Because CLP(F) currently uses depth-first search, it is non-deterministic whether it will try the finite or the infinite path at each branch. If CLP(F) were to use breadth-first search, it would clearly show the possibility that the motion ended, while still modeling the Zeno execution as another possibility. This is probably the best one can hope for. If one gives a computer a model which includes a Zeno execution, the model must show that. If the model can also show that the behaviour is within measurement (or calculation) error, one hopes the user will realize that the initial model is insufficiently defined.

Chapter 7

Conclusions and Future Work

There are two different sources of uncertainty which we model. The first is the various problems which come from representing real numbers by some finite representation in the computer (often floating point representation). These errors can be handled in various ways, including the interval libraries we use, the infinite-precision intervals of Edalat and Heckman [EH02] [Eda01], or, perhaps high precision rationals which Henzinger *et al.* [HHWT97] used. One could, perhaps, also use better analysis techniques, either by finding analytic solutions to some of the ODEs, or by better numerical analysis proving acceptably tight bounds on the calculated solutions. In principle, one could reduce the uncertainty from calculation to any required non-zero limit.

The second source of uncertainty is inherent in analog systems, and that is the measurement uncertainty. While measuring instruments are constantly improving, they must always be imperfect because of Heisenberg's uncertainty principle. From Buridan's principle [Lam86], and the requirement that measurements be made in a sharply bounded time, we get further restrictions on the accuracy of the measurements. These sources of uncertainty can not be strictly bounded. A further difficulty in describing measurement errors is the form that those errors take. A realistic model of a measurement device's output is probably a normal distribution around the actual value (possibly with a spike at all 0s or all 1s output representing a hardware failure). While our method of calculating behaviour is rigorous given the assumption that the distribution has no tail outside the interval, this is unlikely to be a realistic model.

All rigorous numeric models that we know of have intervals either explicitly or implicitly, as the results often involve irrational or transcendental numbers, which cannot be expressed exactly in floating point numbers. As many of the problems hybrid systems involve can not yet be solved analytically, we are left using intervals. The two obvious ways to provide a a rigorous answer are to use intervals explicitly and ubiquitously, as we do, or to do the calculations with floating point numbers and then provide some numerical analysis to prove that the error-bars (interval widths) are sufficiently narrow. That sort of analysis is quite difficult, and it seems unlikely that a programmer approaching a hybrid system would have the training (or the time) to do it.

7.1 Benefits of our Approach

There are several advantages to using CLP(F) to model hybrid Systems:

- CLP(F) is logic based. One can infer properties of a system from imperfect descriptions of the system.
- One doesn't have to completely understand a system to prove safety properties of it. As long as one can constrain the system tightly enough, one can prove statements about the behaviour of the system. One doesn't have to have the system completely specified.
- The paradigm allows simpler proofs. We can prove convergence by splitting a region into areas, and showing that each of those areas eventually leads to a loop.
- The system can rigorously handle non-linear ODEs.
- The semantics of CLP(F) are close to the ODEs describing the problem. The problem specification is translated trivially into a program.

- By making the argument for correctness of the system simpler (because the system itself is simpler), we make it less likely that there will be an error in the proof of correctness.

- CLIP easily answers different kinds of questions about any system it models. Bhatia and Frazzoli [BF04] recently suggested that it would be helpful if instead of merely saying that a system was unsafe, the analysis tool could provide an example of the unsafe behaviour. With CLIP, one simply writes a query asking for a violation of the safety properties. Other analysis tools seem to require a great deal of re-working even though Parrilo [Par04] points out that demonstrating an unsafe point is in NP, while proving there is no such point is in co-NP, and thus harder to demonstrate.

- While CLP(F) is limited to analytic functions, it can handle points at which a function is not analytic, as long as the function is continuous (or nearly so) at all points. One simply writes one function for values above the non-analytic point and another for values below that point.

There has been much work in using CLP (Constraint Logic Programming) to analyze various aspects of hybrid systems [CF00], [Urb96], [Pod00], [GJS96] . One problem with these conventional CLP approaches to modeling hybrid systems is that they must deal with the ODEs describing the continuous part of the system using some sort of approximation, such as discretization into difference equations or restriction to ODEs that have a closed form solution. This introduces a “modeling error” so that the systems are not computationally sound. One must then reason about the modeling error outside of the CLP program. Many systems ignore these errors, and leave it up to the user to understand the numerical instabilities. For example, the SHIFT language [DGS] is very expressive, but it solves non-linear ODEs by using a fourth order Runge-Kutta algorithm without bounding the error term, and hence is not rigorous. This sort of numerical analysis [Act96] is very tricky. To require users to understand numerical analysis under pain of getting a wrong answer is to invite error.

Sensitivity analysis of hybrid systems is even more difficult than sensitivity analysis normally is. In 1999, Galán, Feehery and Barton [GFB99] gave the first technique for explicitly doing sensitivity analysis on hybrid systems. They have extended this work in later work [BL02], but it remains difficult. Because we use intervals ubiquitously, the inputs are given as intervals, so we automatically include the errors stemming from different possible values. Barton et al's techniques allow one to find out whether a wide output interval results from sensitivity to a particular input, or from performance problems in the calculation.

7.2 Limitations of CLP(F)

The primary disadvantage of the CLP(F) approach is that it is very resource intensive and hence can not currently model systems over a long modeling period.

The wrapping problem [MA85] is that in multi-dimensional interval arithmetic, the interval is always an n-dimensional rectangle (a hyper-cube). This rectangle is often much larger than the minimum volume shape to cover all possible values. This excessive over-approximation can make true statements unprovable in CLP(F). CLP(F) makes no attempt to handle the wrapping problem, other than the simple minded solution technique of dividing each rectangle into smaller pieces, exacerbating the performance problems. Wrapping is a problem for CLP(F), as it is for almost any interval arithmetic approach to ODEs. A good approach to the wrapping problem might reduce both the width of resulting intervals and the time it takes to compute them, at some cost in the complexity of the program used. There has been some work on the wrapping problem with ODEs, including Stauning's [Sta96] implementation of Lohner's [Loh87] algorithms and Deville, Janssen, and van Hentenryck's work [DJvH02] on consistency techniques for ODEs

Perhaps the main problem with using CLP(F) to model hybrid systems is that CLP(F) can be extremely computationally intensive. There are two related measures of performance for an interval based system, time and width of result intervals. One can almost always speed up a computation by allowing wider result intervals, and one can sometimes narrow the result intervals by allowing more computation. We have not yet worked on improving the efficiency of CLP(F), but from a few examples, it's clear that careful (or lucky) setting of its tuning parameters can sometimes cut the run time by a factor of two or three while giving equally narrow result intervals.

A major limitation of CLP(F) that it can only handle ODEs, and not PDEs (Partial Differential Equations). Mitchell, Bayen, and Tomlin [MBT01] and Tomlin, Lygeros and Sastry [TLS00] describe systems which can handle PDEs in hybrid system descriptions, but only very approximately.

7.3 Future Work

One of the major benefits of this approach is that the problem of analyzing the hybrid system is transformed into the problem of analyzing the corresponding CLP(F) program. In principle, one should be able to apply well understood program analysis techniques [SH90, HS91, HC88] to CLP(F) and directly infer provable properties of the corresponding hybrid system. In this thesis we describe only the simpler types of analysis that one can do by directly solving CLP(F) constraints related to the hybrid system.

Fehnker and Ivančić [FI04] recently published a set of benchmark systems to test hybrid models on. It would be worth while to see what CLP(F) does with those problems. Unfortunately, two of their three families of benchmarks are linear, and the third has only a square root as a non-linear component. Since CLIP handles rather more general ODEs, it would be nice to have a benchmark to see how well it does on them.

Like this work, Janssen, Van Hentenryck, and Deville [JvHD02] apply constraint techniques to ODEs. While we use a straightforward Interval-Constraint ODE solver in an applied way, they show that added sophistication in the solver can result in significant performance increases. Since performance is always a problem with constraint solvers, it would be worthwhile to apply their techniques to our system.

There is now work using hybrid systems in biology [BFH$^+$04, LT04, Neo04]. Most of this is attempting to model systems starting from scratch. CLIP based hybrid system analysis seems particularly well suited for work like that of Prinz, Billimoria, and Marder [PBM03] who know the connections in the system, and want to calculate the parameters. They generate a database of simulated behaviours for varying values of the parameters. A rigorous approach here might increase the reliability of the data.

Using CLP(F) to model hybrid systems currently requires a lot of knowledge about how CLP(F) works in order to know which solver to use when. Validated constraint compilation [HW99a] is a technique which allows the user to be ignorant of the different solvers, while still getting acceptable performance (both in terms of the width of result interval and in terms of computer time needed). It clearly needs more work, and integrating it with hybrid systems models has yet to be done, but both are in CLP(F), so combining them shouldn't be too difficult.

Appendix A

CLIP Documentation

This appendix contains a draft of what will become a CLIP manual [WH04]. This version is current as of the date this document was formatted (May 5, 2004). As this is still a very rough version, check to see if a later version is available.

This is a start at documenting CLIP. Eventually this will grow into a manual. CLP(I) (Constraint Logic Programming over Intervals) is a constraint logic programming (CLP) language whose domain is the real numbers (Moore's Interval Arithmetic). CLP(F) is a CLP language whose domain is analytic functions over the reals. CLP(F) is written in CLP(I). CLIP is an implementation of CLP(I). Current distributions of CLIP include the file ode.pl, which implements CLP(F) in CLP(I).

A.1 Introduction

CLP(I) is an interval-based constraint logic programming (CLP) language whose domain is the set of real numbers. The class of CLP languages (and their syntax and semantics) was introduced by Jaffar and Lassez in 1987 [JL87]. Jaffar and Maher provide an excellent survey [JM94] of the fundamental concepts of CLP. The idea of calculating over intervals of reals comes from Moore's 1966 book on Interval Arithmetic [Moo66].

The idea of combining CLP and Interval Arithmetic was first conceived by Cleary [Cle87] but the first production quality CLP(I) interpreter was the BNRProlog system developed by Older, Vellino, and Benhamou [Res88], [BO97],[OV93]. BNR Prolog was designed to be verifiably correct in the sense that the intervals it returned were mathematically guaranteed to contain all solutions to the underlying arithmetic constraints. The system however was proprietary and the underlying algorithms were never published in the scientific literature.

CLIP was originally developed as an open source implementation of CLP(I) by Qun Ju and Tim Hickey [HJ], [HJ97] and was used in Qun Ju's thesis [Ju98] as the foundation for a parallel implementation of CLP(I). CLIP has subsequently been extended by Tim Hickey, who added the CLP(F) language, which provides constraints over functions, plotting, and some other refinements. David Wittenberg has made minor changes and fixed some bugs.

CLIP is built on top of Prolog [Pro95], [DEDC96], and currently runs on GNU Prolog [Dia02] and ALS Prolog. The fundamental philosophy is to have a relatively small base of sound primitive constraint contractors which are simple enough so that one can argue convincingly, if not formally prove, that they are correct, and then build more complex solvers on top of the proven system. Since the complex solvers built on CLIP primitives are made up of sound simple solvers, they are also sound. An important feature of CLP languages is that they are theorem provers, so that each answer generated by a CLP program has a direct interpretation as a theorem about the underlying domain.

In CLIP, introducing new constraints is usually a matter of narrowing the interval corresponding to one or more variables. A poor choice of order of contractions will result in poor performance (either by taking a long time or by providing very loose bounds), but will not result in an unsound computation. Writing more complex solvers directly results in systems which are more complex, which makes it harder to construct a direct argument for their correctness.

CLIP currently runs on Linux on x386 architectures and Mac OS X on PPC architecture, using IEEE754 floating point [IEE85] using the "double" (64 bit) word length. Porting to other architectures should be simple, but using a larger floating point word would take a lot of (straightforward) work.

CLIP is an extension to Prolog, so all commands available in Prolog are available in CLIP. Prolog commands are distinguished from CLP(I) commands by the brackets used to enclose them. CLIP commands are written { *command* }. CLP(I) itself can then be extended.

The CLP(F) [Hic00a] [Hic01] language introduces function variables in addition to real variables and has proved to be a useful tool for studying systems defined partly by ODEs (Ordinary Differential Equations). The CLP(F) extensions are now a standard part of CLIP. (It is implemented as a metalevel interpreter in the CLP(I) language.) Constraints on functions and their derivatives are written {[*C1,C2,...,Cn*]}. This document describes CLIP, which implements CLP(I) and CLP(F). We attempt to label CLP(F) extensions as such. Two other extensions are also described – plotting.pl and solve.pl

CLIP can be considered to be a constraint engine over intervals and functions which interfaces to the Prolog engine (a constraint solver over general finite domains).

The CLP(F) language solves analytic constraints by soundly approximating sufficiently differentiable functions by power series with remainder terms and introducing arithmetic constraints among the Taylor coefficients of the functions at the endpoints, at points in the interval, and over the entire range.

CLIP follows the Prolog convention of using lower case identifiers for atoms and upper case for variables.

Note that CLIP follows the US convention of using a period to represent the decimal point, so that one tenth is written 0.1

This manual documents the most important features of CLIP, but currently omits a few features including vectors in CLP(F), the extended syntax for input constants in CLP(I), and the plotting libraries.

A.2 Using CLIP

In CLP(F) the constraint domain allows one to declare variables representing various analytic values including:

- *real numbers, X*
- *infinitely differentiable functions, F, on a finite interval [a,b]*
- *vectors of numbers, functions, or vectors*

The CLP(I) constraint language allows one to express any algebraic equality or inequality constraint among its variables. For example, the query:

```
| ?- {X^2=2,X>0}.
```

returns

```
X = 1.41421356237309... ? ;
no
| ?-
```

The CLIP interpreter represents the interval for X in a compact form. The ellipsis "..." indicates that all shown digits are correct and hence X must lie in the interval:

```
[1.41421356237309,
 1.41421356237310)
```

Also, note the standard Prolog feature that the user entered a semi-colon after the solution and the interpreter responded with "no" which indicates that there are no more solutions.

A.2.1 Multiple Solutions and Non-determinism

Sometimes there may be more than one solution to a given constraint. The constraint solver will indicate this by returning an interval that contains all solutions:

```
| ?- {X^2=2}.
X = [-1.41421356237309536751922678377,
      1.41421356237309536751922678377] ?
```

```
no
| ?-
```

The user must guess whether this is because the system has failed to narrow the interval around one solution or if this is a case where there are multiple solutions. Hickey and Wittenberg [HW99a] discuss methods of determining which is the case, and choosing an appropriate method to use in `solve_clip`, but this work is clearly incomplete. There is a moderately sophisticated solver `solve_clip(METHOD,VARS,N)` which allows one to specify the solving method, (See section A.7.1 for a list of supported solving methods, and a description of each of them.) the list of variables that should be solved, and a parameter N representing how much work should be done (e.g. a maximum allowed width for intervals, or a maximum depth for a divide and conquer splitting routine).

Here, to find the discrete set of solutions one must apply a divide-and-conquer approach where one divides the interval into subintervals and searches for solutions in each one. This is done using the "queue" method of the `solve_clip` solver and typing a semicolon after each solution that it finds:

```
| ?- {X^2=2},solve_clip(queue,[X],0.000001).
X = 1.41421356237309... ? ;
X = -1.41421356237309... ?
(10 ms) no
| ?-
```

The "no" answer at the end indicates that there are no more solutions to that query.

The constraint language for real variables allows any equations and inequalities constructed using the arithmetic operators and the standard mathematical functions (sin, cos, tan, asin, acos, atan, exp (e^x), log, exponentiation, integral powers X^Y, and others).

For example, the following query demonstrates the use of the forward checking solver `fwchk`, which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in `VARS`, and applies the default narrowing procedure. Any boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes. An example of the narrowing done by this method to solve $x^x = 1 + \cos(x) \ \wedge \ x > 0$ is:

```
| ?- {X**X = 1+cos(X), X>0},solve_clip(fwchk,[X],N).
N = 0  X = REAL(0,inf) ? ;
N = 1  X = 1.247504654353... ? ;
N = 2  X = 1.24750465435333... ? ;
N = 3  X = 1.24750465435333... ?
(1720 ms) yes
| ?-
```

Here `N` is the number of steps taken.

A.2.2 Analytic constraints and ODEs

CLP(F) also allows one to constrain functions by functional equations involving many of the same arithmetic operators and mathematical functions as CLP(I) supports. In addition, one can constrain a function or its derivatives to take specific values at specific points and to have a range that lies within an interval.

Consider the following mathematical constraint Q on the function variable F and real variables A and E:

$$Q(F, A, E) \equiv$$
$$(F \in \mathcal{H}([0,1]), F' = F, F([0,1]) \subseteq [-100, 100], F(0) = 1, F(A) = 2, F(1) = E)$$

Q can be represented and solved by presenting the following constraint to the CLP(F) interpreter:

```
| ?- type([F],function(0,1)), {[ ddt(F,1)=F,   F in [-100,100],
       eval(F,0)=1,eval(F,A)=2, eval(F,1)=E ]}.
```

where the `type` predicate indicates that $F \in \mathcal{H}([0,1])$, i.e., F is an analytic function in some open neighborhood of the interval $[0, 1]$. The output given by CLP(F) after 0.76 seconds on a 1 GHZ Mac TiBook is

```
A = 0.6931471... E = 2.7182818... ;
(760 ms) no
| ?-
```

which represents the following answer constraint:

$$C(F, A, E) \equiv (A \in [0.6931471, 0.6931472) \ \wedge \ E \in [2.7182818, 2.7182819))$$

The soundness of the CLIP interpreter implies that it has proven a theorem about the query and its solution constraint:

$$\forall F, A, E \quad Q(F, A, E) \Rightarrow C(F, A, E)$$

$$
\begin{aligned}
Q(a,b,c,d,k_1,k_2,k_3,k_4,t_1,t_2) \quad &\equiv \\
\exists f_1, f_2 \qquad & I = [t_0, t_1], f_1, f_2 \in \mathcal{H}([t_1, t_2]), \\
& E = 0.0000000001, k_3 < x_{20}, \\
& f_1' = k_1 - k_2\sqrt{f_1 - f_2 + k_3} \\
& f_2' = k_2\sqrt{f_1 - f_2 + k_3} - k_4\sqrt{f_2} \\
& f_1(t_0) = a, f_1(t_1) = b, f_2(t_0) = c, f_2(t_1) = d, \\
& f_1([t_1, t_2]) \subseteq [E, 1000], f_2([t_1, t_2]) \subseteq [E, 1000],
\end{aligned}
$$

Figure A.1: A complex non-linear ODE constraint

In other words, if F, A, and E represent a solution to Q, then they must satisfy the answer constraint C. Note that one cannot infer from this theorem that Q has any solutions at all. In this particular case, Q clearly does have a solution

$$F(t) = \exp(t), \quad A = \ln(2), \quad E = e$$

which of course satisfies the answer constraint C.

The function F is then constrained to be equal to its first derivative, and to take the value 1 at 0 and to take values in $[-1000, 1000]$ for all $x \in [0, 1]$. The variables A and E are not declared to be functions and hence are real constants by default. They are constrained so that $F(A) = 2$ and $F(1) = E$. The constraint solver finds A and E to 7 decimal digits of precision and also finds an interval for F not shown here, that specifies intervals for its first 10 derivatives at 0 and 1, and for the range of its first 10 derivatives over $[0, 1]$. The number of derivatives (10) can be set to any value N (but space and time complexity grows quadratically with N).

A.2.3 Complex ODEs

The CLP(F) solver can also handle very complex non-linear differential equations as it based on a "brute force" reduction of the analytic constraints into arithmetic constraints which are solved with a simple interval arithmetic constraint solver. For example, we [HW03] model a system consisting of a fluid with temperature $A(t)$ which is heated by a heating element whose temperature $B(t)$ has a non-linear component $\sin(B(t))$ in its defining ODE. This system is modeled by the following procedure:

```
ode2((T0,A0),[I,[Alpha,Beta,Gamma,Delta]],A,(T1,A1)) :-
  type([A,B],function(0,I)),
{[ ddt(A,1) = Alpha * A + Beta + Gamma*B,
   ddt(B,1) = Delta*(B + 0.1*sin(B)),
   eval(A,0)=A0,   eval(A,T)=A1,
   eval(B,0)=1,
   A in [-1.0E100,1.0E100],
   B in [-1.0E100,1.0E100],
   T=T1-T0,   T in [0,I]
]}.
```

A.3 General Theory of CLP Constraint Domains

A CLP Constraint domain D is specified by giving the syntax and semantics of its underlying constraint language.

Syntactically, constraints in a domain D are a conjunction of atomic formulas in a first order language L_D. The language is specified by giving the predicate symbols, function symbols, constant symbols, and variable symbols of the language. (In general the language may require a multi-sorted logic to cleanly handle variables of different types.)

The semantics for a CLP Constraint domain is given by a specific model ϕ_D for the language L_D, i.e., a concrete interpretation of the predicate, function, constant, and variable symbols. So ϕ_D is a map from the symbols to a corresponding set of predicates, function, and constants. The theory T_D of the domain is the set of all first order formulas in L_D which are true under the interpreteration ϕ_D.

A CLP Constraint solver is an algorithm which tests for unsatisfiability of constraints. It must be correct, but it doesn't have to be complete, i.e. for any constraint C the solver either determines that the constraint is unsatisfiable, or it makes no claim about the constraint's satisfiability. Thus, if the solver determines that a constraint $C(X)$ is unsatisfiable, then there is a proof that

$$T_D \models \neg\exists X\ C(X)$$

but the solver might not be able to detect all unsatisfiable constraints.

An interval constraint solver approximates the solution set for a constraint by assigning an interval to each variable in the constraint. For the purpose of this section, we can think of an interval for a general domain to a subset of the domain (possibly with some additional restrictions).

An interval constraint solver proves unsatisfiability by applying contraction algorithms which attempt to shrink the intervals without removing any solutions. If an interval for one of the variables is shrunk to the empty set, then the constraint is unsatisfiable. In general if an interval constraint solver is given a constraint $C(X)$ on a tuple X of variables and it contracts the variables from an initial tuple I to a subset J, then one can infer that

$$T_D \models \forall X. (C(X) \wedge X \in I) \;\Rightarrow\; X \in J$$

That is, the contraction algorithms can be viewed as mechanical theorem provers for a simple class of formulas and hence the CLP constraint solver is itself a theorem prover.

A CLP(D) program P can be interpreted as a first order theory T_P using Clark's completion semantics [Cla78]. In this semantics, each predicate in the program is replaced by a rule

$$\forall X p(X) \Leftrightarrow (\exists Y_1 q_1(X, Y_1)) \wedge \ldots$$

where each clause of the program is viewed as a conjunction q_1 of atomic formulas over the variables X from the head and some new variables Y introduced in the clause. If a CLP(D) interpreter generates m interval solutions $I_1, \ldots, I_m$ to a query $Q(X)$ then one can infer that the interpreter has produced a proof that

$$T_D \cup T_P \models \forall X.Q(X) \Rightarrow \bigvee_j X \in I_j$$

For example, let P is the following CLP(D) program over the domain D of reals:

```
p(X,Y,N) :- {N=0,Y=cos(Z),exp(Z+Y)=X}.
p(X,Y,N) :- {N>0, M=N-1, Y= exp(Y)+cos(Y1)},p(X,Y1,M).
```

then its Clark semantics is

$$
\begin{aligned}
\forall X, Y, N.p(X, Y, N) \;\; &\Leftrightarrow \;\; \exists Z(N = 0 \wedge Y = cos(Z) \wedge e^{Z+Y} = X) \\
&\vee \;\; (N > 0 \wedge M = N - 1 \wedge Y = e^{Y} + cos(Y_1) \wedge p(X, Y1, M))
\end{aligned}
$$

In the next two sections we discuss the syntax and semantics of the CLP(I) and CLP(F) languages.

A.4 How Clip Works

CLIP maintains a dequeue (double ended queue) of active constraints as well as a stack of all constraints, a stack of interval-valued variables and some other control structures to handle backtracking (a stack of choice points, a trail of bindings).

After a new constraint is added to the constraint stack and enqueued in the dequeue, the solver iteratively processes constraints in the dequeue and applies a contractor for that constraint to attempt to narrow (shrink, contract, ...) some of the intervals in the variable stack without removing any solutions to the constraint in question.

If the narrowing results in a variable X being contracted by more than some fixed amount specified by the tuning parameter `sensitivity`, then all constraints involving X are put on the dequeue. If the change to a X was more than the tuning parameter `stack_sensitivity`, the constraints are added to the front of the dequeue (i.e. pushed), otherwise they are added to the end (i.e. enqueued). CLIP does not add a constraint to the dequeue if it is already there.

CLIP continues to narrow intervals until the dequeue is empty. A naive implementation of this solver would be subject to freezing up when the solver enters a cycle of making very small changes to a set of variables with neglible progress (as measured in the decrease in the total size of the interval). To eliminate this problem (and guarantee a maximum return time for each call to the constraint solver), CLIP introduces the `max_narrow` tuning parameter. Once `max_narrow` operations occurred, CLIP continues narrowing, but only puts constraints on the dequeue if the change in the one of the variables in the constraint is more than the `insensitivity` tuning parameter. Note that `sensitivity`, `stack_sensitivity` and `insensitivity` are relative, not absolute changes, and all of these parameters can be modified inside clip using `set_clip`. Also, the precise semantics of these parameters is tricky since they must apply to bounded intervals as well as those where one endpoint is infinite.

A.5 The CLP(I) Constraint Domain

The CLP(I) constraint language defines a constraint as a conjunction of atomic formulas in the language specified below. Constraints in CLP(I) are enclosed with curly braces to indicate that they are to be processed by the constraint solver and not the usual Prolog engine. Thus, all CLP(I) constraints have the form:

$$\{C_1, C_2, \ldots, C_n\}$$

where the C_i are atomic constraints.

The CLP(I) constants include the standard representations of decimal numbers and the semantics of a CLP(I) constant is somewhat complex. If the decimal number constant c is in fact exactly representable by a floating point number f, then the the

CLP(I) semantics assigns the symbol c to the value f. For decimal number symbols c without an exact floating point representation, there are two floating point numbers f_0 and f_1 such that $f_0 < c < f_1$ and the semantics of CLP(I) maps c to some unspecified number strictly between f_0 and f_1. Note that in this case, syntactic constants are actually represented by slightly constrained variables.

CLP(I) variables are represented by capitalized identifiers (as in Prolog) and correspond to real numbers under the CLP(I) model.

The integers are naturally embedded in the reals and we identify the booleans with the subset $\{0, 1\}$ with 0 being false and 1 being true. Thus, $integer(X)$ is true iff X is an integer and $boolean(X)$ is true if and only if X is 0 or 1. We could actually define these in terms of the other constraints:

$$integer(X) \equiv sin2\pi(X/2) = 0 \quad boolean(X) \equiv integer(X) \wedge X \in [0, 1]$$

CLP(I) has a very rich language of functions which include those shown in Table A.2.

The atomic constraints CLP(I) constraints are constructed from the predicates in Table A.1. These represent the usual predicates on reals, with one exception – the "in" predicate, which we discuss in more detail below.

The usual use of the "in" predicate is X `in` $[A, B]$ which has the meaning $A \leq X$ and $X \leq B$ One can get a reasonable semantics for X `in` Y where Y is a real variable or expression by using the domain of real-valued functions on an unspecified set S instead of the domain of real numbers. The real numbers a are embedded in this domain as constant functions f_a with $\forall t.f_a(t) = a$ and the F in G relation is interpreted as range inclusion $F(S) \subseteq G(S)$. The real operators are extended to functions using a pointwise semantics: $(F \circ G)(t) = F(t) \circ G(t)$. Since the reals are embedded in this function space, one can introduce a type system with both real and function variables. This semantics plays a central role in the CLP(F) constraint domain described in the next section.

Symbol	Arrity	Pos	Description
$S = T$	2	In	equals
$S < T$	2	In	less than
$S =< T$	2	In	less than or equal
$S >= T$	2	In	greater than or equal
S`\==`T	2	In	not equals
`integer`(T)	1		Integer
`boolean`(T)	1		Boolean
S `in` T	2	In	Containment

Table A.1: CLP(I) constraint predicates where S and T are CLP(I) constraint terms Pos is "In" for infix operators.

Note that the $<, \leq, =$ are functions as well as predicates. As function they return 0 or 1 and so you can write

```
(X<1) + (Y< 1) + (Z<1) = 2
```

to represent the constraint that exactly two of the three variables X, Y, Z are smaller than 1.

Symbol	Arrity	Pos	Description
$S + T$	2	In	addition
$S * T$	2	In	multiplication
$S - T$	1		unary minus
$\texttt{exp}(T)$	1		e^T
$\texttt{sq}(T)$	1		square
$\texttt{abs}(T)$	1		absolute value
$\texttt{sgn}(T)$	1		sign of argument, -1, 0, or 1
$\texttt{max}(S, T)$	2		Maximum
$\texttt{min}(S, T)$	2		Minimum
$\texttt{floor}(T)$	1		Floor
$\texttt{ceil}(T)$	1		Ceiling
S `or` T	2	In	Logical OR (and S,T are boolean, i.e. in {0,1}
S `and` T	2	In	Logical AND (and S,T are boolean)
S `xor` T	2	In	Logical eXclusive OR (and S,T are boolean)
S `not` T	1		Logical Negation (and T is in {0,1}
$\texttt{sin}(T)$	1		Sine
$\texttt{cos}(T)$	1		Cosine
$\texttt{tan}(T)$	1		Tangent
$S < T$	2		Less than function mapping to {0,1}
$S \leq T$	2		Less than or equals function
$S = T$	2		Equals function
$\texttt{sin2pi}(T)$	1		returns $\sin(2 \cdot \pi \cdot X)$
$\texttt{cos2pi}(T)$	1		returns $\cos(2 \cdot \pi \cdot X)$
$\texttt{tan2pi}(T)$	1		returns $\tan(2 \cdot \pi \cdot X)$
$\texttt{evenpow}(S, T)$	2		if $S > 0, S^T$; if $S = 0, 0$; if $S < 0, (-S)^T$
$\texttt{oddpow}(S, T)$	2		if $S > 0, S^T$; if $S = 00$; if $S < 0, -(-S)^T$
`psqrt`(T)	2		the positive squareroot of T

Table A.2: Clip functions

A.6 The CLP(F) Constraint Domain

Constraints in CLP(F) all have the form

$$\{[C_1, C_2, \ldots, C_n]\}$$

where the C_i are atomic CLP(F) constraints. The variables that appear in CLP(F) constraints are of two types. Either they are real variables (as in CLP(I) constraints) which represent real numbers or they are function variables which represent infinitely differentiable functions defined on a finite interval $[a, b]$.

The function variables must be declared outside of the CLP(F) constraint. They are declared used using a CLP(F) declaration as follows:

```
decls([F1,...,Fn],function(A,B))
```

This states the the Fi are infinitely differentiable functions in an open set containing the interval $[A, B]$.

Atomic CLP(F) constraints include all CLP(I) constraints on real variables and in addition include the following, where F, G are expressions of type function and S, T are expressions of type real. The list of functions available in CLP(F) includes all CLP(I) functions on reals and add the operators in Table A.4. Note that when functions and reals are combined (as in $F + X$) the result is a function obtained by interpreting the real value as a constant function.

There are also a few tuning parameters and printing procedures CLP(F) adds:

`set_degree(N)` sets to N (default 10) the degree of the Taylor approximation polynomial used for functions.

Symbol	Arrity	Pos	Description
$F = G$	2	In	equality of functions
$F = T$	2	In	function F is the constant function T
$T = F$	2	In	function F is the constant function T
$S = T$	2	In	real numbers S and T are equal
`identity`(F)	1		F is the identity function
$F \leq G$	2	In	$F(t) \leq G(t)$ for all $t \in [A, B]$
$F\ in\ T$	2	In	$F([A, B]) \subseteq T$
$F\ in\ G$	2	In	$F([A, B]) \subseteq G([A, B])$

Table A.3: CLP(I) constraint predicates where F, G are functions and S, T are reals

Symbol	Arrity	Pos	Description
$F + G, F + T, T + F$	2	In	addition
$F * G, F * T, T * F$	2	In	multiplication
$F - G, F - T, T - F$	1		subtraction
$F/G, F/T, T/F$	1		division
`exp`(F)	1		$e^{F(t)}$
`log`(F)	1		$\log(F(t))$
`sin`(F)	1		$\sin(F(t))$
`cos`(F)	1		$\cos(F(t))$
`tan`(F)	1		$\tan(F(t))$
`ddt`(F, n)	2		nth derivative of F, n an integer constant
`eval`(F, T)	2		$F(T)$

Table A.4: CLP(F) functions

`print_ps_clip/1` prints the power series of its argument (defined in ode.pl)

A.7 CLIP Predicates and Commands

CLIP is an extension of GNU Prolog in which several new predicates have been added to the base system. The most important of these new predicates are the constraint predicates for CLP(I) – { } and {[ ]} and `decls`. The other predicates are mostly commands that allow one to interact with the constraint engine – getting/setting tuning parameters and other information stored in the constraint engine. They don't affect the logical semantics of the program, but they may affect the performance.

The basic predicates and commands in CLIP are described in section A.10.

A.7.1 Command Descriptions

'$INT'(I) refers to the interval variable whose index in the constraint engine is I.

- `{ }/1` takes the included list of constraints and does as much narrowing as it can.
- `{[ ]}/1` takes the included list of function constraints and adds them to the current constraints (defined in ode.pl)
- `help_clip/0` prints a list of available commands.
- `set_cp_clip/0` sets a choice point in the constraint engine.
- `print_clip/1` prints its argument as an interval rather than as an index into the constraint engines variable stack.

- `reset_clip/0` sets clip to its original state. In particular, it clears the dequeue. Dequeue usage can be a problem on certain calculations. Note that `reset_clip` does *not* change the values which have been set with `set_clip`.

- `get_bounds_clip/3` (Var, Lo, Hi) returns the bounds of interval Var in Lo and Hi. This only works on variables which are bound to an interval. For variables which are not bound, use '$INT'(Num) where Num is the index in the constraint engine of the interval variable.

- `get_hex_bounds_clip/3` gets the bounds as two hex numbers

- `get_bit_bounds_clip/3` gets the bounds as two bit strings (represented as 4 unsigned shorts).

- `dump_clip/1` provides information on CLIP's state. `dump_clip(stats)` gives statistics: number of choice points, number of constraints, value of maxcon, number of variables, value of maxvar, value of conbot, value of trail, var_dep, narrows, changes, and big changes.

  `dump_clip(all)` lists all the state information it has, which includes stats, and also gives information on each variable, including its type, range, and the last change to it.

  It lists the active constraints and variables (Note that constants are variables whose numbers start at 99999 and go down, variables start at 0 and go up) and choice points.

  It then provides statistics on CLIP's state: the number of choice points (essentially the number of vertices in the tree of execution), the number of constraints active (con), the maximum number of constraints (maxcon) at any one time, the bottom of the constant storage area (conbot), the number of narrowing operations performed so far (narrows), the number of primitive narrowing operations (prim), the number of times an interval changed (changes), and the number of times an interval changed by more than 10% (big changes).

!!! what other arguments does it take??

The values described by `dump_clip` can all be reset by `reset_clip`.

- `set_clip/2 (Var, Val)` is used to set the control variable `Var` to `Val`. Note that variables that can take non-integer numbers must have at least one digit before and at least one digit after the decimal point. Values which can be set are
  - `accuracy` (default: 0.0001) is the minimum interval size. A variable whose size is less than `accuracy` is not requeued for more narrowing.
  - `sensitivity` (default: 0.05) The minimum change to an interval which will cause CLIP to put that interval back on the stack of intervals to narrow further (until `max_narrow` narrowings have occured). `sensitivity` can safely be set to 0.0 so that any change in an interval will cause the interval to be requeued. Note that `sensitivity` is a fraction, not an absolute number.
  - `max_narrow` default: (10 000) - the maximum number of narrowings using `sensitivity` to determine whether to continue narrowing (used to prevent infinite loops). After `max_narrow` steps, CLIP will only add an interval to the stack of work remaining if the change was greater than `insensitivity`.

insensitivity should be significantly greater than sensitivity. Note that sensitivity is a fraction, not an absolute number.

- insensitivity (default: 0.125) The minimum change necessary to requeue an interval after max_narrow narrowings have been done. Note that insensitivity is a fraction, not an absolute number.
- narrow_debug boolean. (default: false) causes CLIP to show the value of the interval being narrowed before and after each narrowing operation. This generates an enormous amount of data, so it's only useful for debugging relatively small programs.
- finite_bounds boolean. (default: false) determines whether new variables should have finite or [− maxreal, + maxreal] bounds.
- stack_sensitivity (default: 0.25) If an interval is narrowed by more than stack_sensitivity, then it is put on the front of the dequeue to be renarrowed immediately. If the fraction the interval was narrowed by was by less than stack_sensitivity, but more than sensitivity, it is put on the end of the dequeue. If the change is less than sensitivity, it is not requeued. Note that stack_sensitivity is a fraction, not an absolute number.

- get_clip/2 is used to read the control values which set_clip can set as well as some state description. A useful idiom is `get_clip (A,B), write(A=B),nl, fail.` This gets all the parameters, writes the name and value of one, writes a new line, then gets the next value. Some values which can be checked with get_clip, but not set with set_clip are:

  - `var_top` initial value: 1
  - `constraint_top` initial value: 0
  - `constant_top` initial value: 99 999
  - `max_vars` default 100 000

- `plot2_solve(M,[X,Y],E,F)` (defined in plot.pl)
- `fplot([X,Y],E,F)` generates GNUPLOT dataset from 2D constraints
- `fplot([X,Y,Z],E,F)`
- `plot3_solve(M,[X,Y,Z],E,F)` generates 3dvplot datasets from 2D constraints.
- `plot_param([X,Y,Z],Params,E,F)`
- `narrow_all_clip(N)` Adds all constraints to the dequeue, and them performs N narrowing operations.
- `absolve(L,S,F)` Tries to narrow an interval by checking to see if there are any solutions near the bottom of the interval. If not, it removes the bottom section of the interval. This is repeated until removing a piece of that size from the bottom of the interval would remove an area in which there are solutions. It then does the same for the top of the interval. L is a list of intervals to test, (S, F) give the starting and ending sizes of the piece to check. If F is less than S, it does nothing. Absolve starts by checking for solutions in a piece of size $1/S$ times the size of the original interval and reduces the size of the piece by half until it reaches one of size $1/F$. By convention S and F are powers of 2.

`absolve` prints characters to show its progress. "-" means that it succesfully removed a piece from the low end of the interval "," that it failed to remove a piece from the low end of the interval "+" that it removed a piece from the high end of the interval ";" that it failed to remove a piece from the high end of the interval Whenever it finishes absolving 1 interval, it prints "*"

- `contract_vars(Vs, G)` solves the given constraints (Vs) for the goal G, and then cleans the stack of all the temporary constraints used. All the variables in Vs must be intervals. Returns the union of the solutions. Defined in contract.pl
- `solve_clip(Solver, Term, Eps)` Legal solvers are:
  - `queue` Splits each variable which is too large, then continues going through the list splitting each variable once per time until all the variables are small enough.
  - `fwchk` (Forward Checking Solver) A divide and conquer solver which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in `VARS`, and applies the default narrowing procedure. Any boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes.
  - `seq` this traverse a search space by putting the variables into a queue and sequentially splitting the top variable in the queue and the moving in to the back of the queue
  - `incr` – this does a breadth first search (like fwchk) of the solution space, but after each breadth-first splitting it reports on each connected component

of the solution space before going deeper.

  - `bf`

  Term is the set of variables to work on, and Eps is a work parameter, whose meaning varies according to which solver is used.

- `solve_clip(Term, Eps)` Calls `solve_clip/3` using queue as the solver.

- `decls/2`

- `print_ps/1` prints the power series of its argument by showing the endpoints a, b of the interval $I = [a, b]$ is defined on as well as the values of its first n derivatives at a at b and on the interval I.

A.8 Implementation

In this section we give a brief overview of how the CLP(I) and CLP(F) constraint solvers are implemented.

A.8.1 CLP(I) constraint solving

The CLP(I) constraint solver is based on a relatively simple model. At any point in time the set of constraints that have been encountered in the current branch of the Prolog search tree are stored in a stack of primitive constraints called the constraint store. Each new constraint that is added to the system is decomposed into a conjunction of primitive constraints and these constraints are added to the constraint store and the variables that appear in the constraint are pushed onto a stack of interval variables. These new primitive constraints are also put into a queue of active constraints.

The constraint solver then attempts to contract the intervals in the interval variable stack by processing the constraints in the constraint queue. If one of the constraints in the queue becomes unsolvable (i.e. an interval contracts to the empty set), then backtracking is triggered as the solver has proved that the current set of constraints is unsatisfiable.

On backtracking the constraints pushed on the stack since the last choice point are popped off, as are the new constraint variables, and any contractions of interval variables are also replaced with their original values using a binding trail stack.

The decomposition of constraints into primitive constraints is a simple process of introducing temporary variables for each subexpression, e.g.

$$X^2 + Y^2 = 25$$

would be mapped to the following conjuction of primitive constraints

$$X^2 = T_1, Y^2 = T_2, T_1 + T_2 = T_3, T_3 = 25$$

where the constant 25 has been compiled into a constraint variable with a constant value.

The constraint solving algorithm simply processes the constraint in the queue by

- taking off the first primitive constraint $C(X_1, X_2, X_3)$ in the queue,

- applying a contraction algorithm for that constraint on its arguments X_1, X_2, X_3,

- if any of the variables X_i are contracted by a sufficiently large amount, then all constraints that depend on them (except for C itself), are added to the queue

- the constraint solving continues until either a maximum number of contractions has been performed (`max_narrow` — a tuning parameter), or a constraint has been found to be unsatisfiable.

A.8.2 Primitive Contraction

The contraction algorithms for the primitive constraints have been implemented very carefully making full use of the directed rounding capabilities of the underlying processors so as to contract their intervals without removing any solutions. Some of these contraction algorithms have published correctness proofs, others have been proved correct only by the authors, but have not been peer reviewed. Also, the programs implementing the algorithms have not been formally proved correct.

As a simple example of contraction, consider the contraction operator for the addition constraint $X+Y = Z$, then a correct contraction algorithm for this constraint is

$$\begin{aligned} X &\leftarrow X \cap (Y + Z) \\ Y &\leftarrow Y \cap (Z - X) \\ Z &\leftarrow Z \cap (Z - Y) \end{aligned}$$

where

```
Y+Z := [addLO(ylo,zlo),addHI(yhi,zhi)]
Z-X := [subLO(zlo,xhi),subHI(zhi,xlo)]
Z-Y := [subLO(zlo,yhi),subHI(zhi,ylo)]
```

where `addLO, addHI, subLO, subHI` are operations that add or subtract two floating point numbers and the round to the nearest floating point below (for LO) or above (for HI) the actual sum or difference.

A similar approach is used for multiplication, division, and the exponential and trigonometric constraints (although these require that the math libraries for computing exp, sin, etc be rewritten to return intervals that are certain to contain the actual value).

The `integer(X)` constraint is even easier to implement:

```
X := [ceiling(xlo),floor(xhi)]
```

Note that for soundness all we need to verify is that the contractors do not remove any solutions from the constraint. They do not have to remove all non-solutions!

The library of contractors which clip uses is the C library `smathlib` which is an opensource project available from `interval.sourceforge.net` `smathlib` has been tested under Linux on x86 and Mac OS X on Power PC.

A.8.3 CLP(F) constraint solving

The CLP(F) constraint solver is implemented in CLP(I). The key idea is to soundly approximate infinitely differentiable functions defined on an interval $I = [A, B]$ by providing intervals for the values of the first n derivatives at the end points, as well as intervals containing the range of their derivatives over the entire interval I.

More precisely, we approximate a functions by a set of $3n+5$ intervals where n is a tuning parameter (default value of 10). The idea is to use an abstraction operator $\gamma_n(f)$ that maps a function f on $I=[a,b]$ to the following tuple

$$\begin{array}{l} a, b, \\ f(a), f'(a), f''(a), \ldots, f^{(n)}(a) \\ f(I), f'(I), f''(I), \ldots, f^{(n)}(I) \\ f(b), f'(b), f''(b), \ldots, f^{(n)}(b) \end{array}$$

in $\mathbf{R}^2 \times \mathbf{R}^{2n+2} \times \mathbf{S}^{n+1}$ where S is the set of real intervals and $f^{(n)}(I)$ denotes the range of the nth derivative of f on the interval I. We can then approximate a set U of functions by a tuple Z of $3n+5$ intervals such that $\gamma_n(f) \in Z$ for each $f \in U$. We can then compute approximations of functional expressions by computing operations on their γ_n approximations. If F is a tuple of intervals such that $\gamma_n(f) \in F$ we say, by abuse of language, that $f \in F$. Thus, if $f \in F$ and $g \in G$, then $f+g \in F+G$ where $F+G$ is obtained by adding the corresponding intervals in F and G.

Thus, the decls predicate

```
decls([F],function(A,B))
```

is implemented by binding F to a tuple of interval variables:

$$\begin{array}{rcl} F & = & \\ & & [[A, (F00, F01, F02, ..., F0n)], \\ & & [I, (R0, R1, R2, ..., Rn)], \\ & & [B, (F10, F11, F12, ..., F1n)]] \end{array}$$

where I is the interval $[A, B]$. Since it is assumed that F is infinitely differentiable, we also add Taylor constraints which relate the values of F and its derivatives at A to the Taylor coefficients at B. The remainder terms are computed from the intervals bounding the ranges of the derivatives on $[A, B]$. For example, we add the following constraints:

$$T = B - A,$$

$$F10 = F00 + T * Z1, Z1 in R1,$$

$$Z1 = F01 + T/2 * Z2, Z2 in R2,$$

$$Z2 = F02 + T/3 * Z3, Z3 in R3,$$

$$\ldots$$

$$I = [A, B],$$

$$R0 in F00 + I * R1,$$

$$R1 in F01 + I/2 * R2,$$

$$R2 in F02 + I/3 * R3,$$

$$\ldots$$

and we also add constraints expressing the F0j in terms of the F1j, and vice versa. These constraints allow information about the function at one end point to be propogated to the remainder terms and to the other endpoint. The $R0$ term can be bounded using the syntax:

```
F in [L,H]
```

which adds the constraints $L \leq R$ and $R \leq H$. The value of the function at any point `X` in $[A, B]$ can be expressed using the eval function:

```
eval(F,X) = Y
```

which adds the Taylor constraints for expressing Y in terms of (X-A), the derivatives at X, and the remainder terms (Rj), as well as (X-B), the derivatives at Y, and the remainder terms (Rj).

The constraint $ddt(F, 1) = F$ is converted into two primitive constraints $ddt(F, 1) = G$ and $G = F$. The primitive constraint $ddt(F, 1) = G$ is converted into the obvious set of corresponding constraints

$$F10 = G00,$$

$$F20 = G10, ...$$

$$R1 = S0,$$

$$R2 = S1, ...$$

$$F11 = G01,$$

$$F21 = G11, ...$$

Other function operators are treated similarly, for example $F * G = H$ generates constraints on the Fij, Gij and Hij as well as their remainder terms expressing the corresponding relations among the derivative functions:

$$\begin{aligned} H00 &= F00 * G00 \\ H01 &= F00 * G01 + F01 * G00 \\ H02 &= F00 * G02 + F01 * G01 + F02 * G00 \\ \ldots &\quad \ldots \end{aligned}$$

The exponential and trigonometric functions are handled by reducing them to other functional equations, e.g.

$$F = exp(G) \Rightarrow F' = G' * F \ \wedge \ F(A) = e^A \ \wedge \ F(B) = e^B \ \wedge \ F([A,B]) = e^{[A,B]}$$

and so the constraint $F = exp(G)$ is converted to

```
ddt(F,1)=G*F
eval(F,A)=exp(A)
eval(F,B)=exp(B)
F in exp([A,B])
```

The equality constraint among functions is soundly approximated by setting the corresponding intervals in their γ_n approximation equal and the inequality constraint $F < G$ is converted to inequalities

```
eval(F,A) < eval(G,A)
eval(F,B) < eval(G,B)
eval(F,Z) < eval(G,Z), Z in [A,B]
```

Note that the constraint solver only needs to be sound not complete (i.e. contractions need only keep all solutions, they don't have to remove all non-solutions). Thus, we are free to transform any constraint C(X) into a weaker constraint D(X) as long as $C(X) \Rightarrow D(X)$, because if $D(a)$ if proved to be false (and hence a is eliminated), then we will know that $C(a)$ is likewise false and can be eliminated.

A.9 Known Bugs and Issues

A.9.1 Unification and Constraint Solving

Probably the most aggravating of the problems with the CLIP implementation is that it currently does not support a tight integration with GNU Prolog's unification. Thus, two interval variables can only be unified by an explicit constraint {X=Y}. In particular, the following query will succeed:

```
?- {X=X, Y=Y, X=Y}.
X=Y
?-
```

But if we move the last equality out of the constraints, it fails:

```
?- {X=X, Y=Y}, X=Y.
no
?-
```

More subtle is that head unification is converted into an implicit equation which will fail. Thus if p is a procedure defined by:

$$p(X, X).$$

then the following query will also fail.

```
?- {X=X, Y=Y}, p(X,Y).
no
?-
```

This bug can be fixed by using some recent extensions to GNU Prolog that allow one to add unification hooks, but for now it requires a careful (manual) separation of interval variables from tree variables which guarantees no interval variables are implicitly unified in the head of any rule.

A.9.2 Other bugs and issues

- `help_clip` fails to write out the constraint operators. This may be a result of porting from Sicstus Prolog and using $\wedge$, but GNU Prolog does define $\wedge$ in what seems to be the correct way.

- `absolve` fails when the interval it's working on is very small. It appears that when the interval it's checking is very small (perhaps 1 ULP??) it goes into an infinite loop.

- It would be nice if $>$ worked on functions. For now, we use `F in [0,_]` for $F \geq 0$.

- Related to the previous item, one cannot have a function f equal a constant c. A workaround is to say f = c + 0 * f

- `psqrt` is non-analytic at 0. Currently we simply define `psqrt` only for values greater than some small ϵ, but that's not rigorous.

- When reading non-integer numbers, even if the number (say 0.5) has an fp representation, the interval read in is at least 2 ULPs wide, 1 on each side of the correct value.

- Not really a bug, but when you increase Prolog's stack size, Clip does not take advantage of the extra memory. Perhaps we should re-compile clip with larger arrays to take advantage of the larger memories that have become standard. As of Jan. 2004, clip's total static memory usage is about 20 MB. - we could easily up all the static declarations by a factor of 5 or 10. April 2004 - I increased all of the array sizes by a factor of 10, so we're using 200MB plus what Prolog uses. Should we increase them further?

- If you don't specify a range for a function, all sorts of intervals grow much larger than they should. Perhaps a warning would help??

- Prolog warns about singleton variables, but if you get too enthusiastic about putting underscores in front of variables, there is no warning that you have two variables with the same name _X and they are treated as separate variables.

- Clip is much more sensitive than it should be to the order in which constraints are listed. This may be a bug, or it may be a place for improvement.

- When reading integers which are too large, clip sometimes reads positive integers as negative. Presumably it is writing an unsigned int, which is then read as a signed int.

```
| ?- {T=1000000000}.


T = -73741824 ?
```

A.10 Quick Reference

A.10.1 CLIP commands

`{ }/1`

`help_clip/0`

`set_cp_clip/0`

`reset_clip/0`

`get_bounds_clip/3`

`dump_clip/1`

`set_clip/2`

`get_clip/2`

`narrow_all_clip/1`

ode.pl adds:

`{[ ]}/1`

`set_degree/1`

`decls/2`

`print_ps_clip/1`

solvers.pl adds:

`print_clip/1`

`solve_clip/3`

`solve_clip/2`

plotting.pl adds:

`plot2_solve/4`

`fplot/3`

contract.pl adds:

`contract_vars/2`

A.10.2 CLP(I) predicates and functions

Symbol	Arrity	Pos	Description
$S = T$	2	In	equals
$S < T$	2	In	less than
$S =< T$	2	In	less than or equal
$S >= T$	2	In	greater than or equal
S\==T	2	In	not equals
integer(T)	1		Integer
boolean(T)	1		Boolean
S in T	2	In	Containment

Symbol	Arrity	Pos	Description
$S+T$	2	In	addition
$S*T$	2	In	multiplication
$S-T$	1		unary minus
`exp`(T)	1		e^T
`sq`(T)	1		square
`abs`(T)	1		absolute value
`sgn`(T)	1		sign of argument, -1, 0, or 1
`max`(S,T)	2		Maximum
`min`(S,T)	2		Minimum
`floor`(T)	1		Floor
`ceil`(T)	1		Ceiling
S `or` T	2	In	Logical OR (and S,T are boolean, i.e. in {0,1}
S `and` T	2	In	Logical AND (and S,T are boolean)
S `xor` T	2	In	Logical eXclusive OR (and S,T are boolean)
S `not` T	1		Logical Negation (and T is in {0,1}
`sin`(T)	1		Sine
`cos`(T)	1		Cosine
`tan`(T)	1		Tangent
$S<T$	2		Less than function mapping to {0,1}
$S\leq T$	2		Less than or equals function
$S=T$	2		Equals function
`sin2pi`(T)	1		returns $\sin(2\cdot\pi\cdot X)$
`cos2pi`(T)	1		returns $\cos(2\cdot\pi\cdot X)$
`tan2pi`(T)	1		returns $\tan(2\cdot\pi\cdot X)$
`evenpow`(S,T)	2		if $S>0, S^T$; if $S=00$; if $S<0,(-S)^T$
`oddpow`(S,T)	2		if $S>0, S^T$; if $S=0,0$; if $S<0,-(-S)^T$

A.10.3 CLP(F) predicates and functions

Symbol	Arrity	Pos	Description
$F = G$	2	In	equality of functions
$F = T$	2	In	function F is the constant function T
$T = F$	2	In	function F is the constant function T
$S = T$	2	In	real numbers S and T are equal
`identity`(F)	1		F is the identity function
$F \leq G$	2	In	$F(t) \leq G(t)$ for all $t \in [A, B]$
$F\ in\ T$	2	In	$F([A, B]) \subseteq T$
$F\ in\ G$	2	In	$F([A, B]) \subseteq G([A, B])$

Symbol	Arrity	Pos	Description
$F+G, F+T, T+F$	2	In	addition
$F*G, F*T, T*F$	2	In	multiplication
$F-G, F-T, T-F$	1		subtraction
$F/G, F/T, T/F$	1		division
$\texttt{exp}(F)$	1		$e^{F(t)}$
$\texttt{log}(F)$	1		$\log(F(t))$
$\texttt{sin}(F)$	1		$\sin(F(t))$
$\texttt{cos}(F)$	1		$\cos(F(t))$
$\texttt{tan}(F)$	1		$\tan(F(t))$
$\texttt{ddt}(F, n)$	2		nth derivative of F, n an integer constant
$\texttt{ddt}(n, F)$	2		nth derivative of F, n an integer constant
$\texttt{eval}(F, T)$	2		$F(T)$

Appendix B

Code for n-tanks system

Naming scheme: Tanks are numbered from 1 to n. A pipe's number is the number of the tank above it. A valve's number is the number of the pipe it is in. Fi is the flow out of tank i, in pipe i

Variables

t = Time

Discrete Variables (regime or program state)

r1 r2 r3 r4 = regime of tank i in above, below

m1 m2 m3 m4 = regime of valve motion i in opening, closing, halted

r1 r2 r3 r4 = regime of valve position i in shut, trans, normal

Continuous Variables

d1 d2 d3 d4 = depth of water in tank i in $[0, \infty]$

p1 p2 p3 p4 = valve position for valve out of tank i in $[0, 1]$

f1 f2 f3 f4 = flow out of tank i

Parameters

h1 h2 h3 h4 = height of tank i above base

c1 c2 c3 c4 = flow constants for pipe out of tank i

e1 e2 e3 e4 = exponent for valve out of tank i

v = speed at which valves open - could be parameterized later

f00 = initial flow into tank 1

delta = maximum time step for problem

B.1 TEST examples

This section sets up a version of the problem by specifying values for all the parameters, and starting values for all the variables.

```
ex1(Initial_state,Problem_constants,N,Final_state) :-
 set_clip(sensitivity,0.0), set_clip(max_narrow,100000),

 Problem_constants =
[delta=0.05,             % maximum time step
        h1=10,h2=9,h3=8,h4=7,   % tank heights

                         % exponent for decay of flow as f(position)
     ke1= -3.1, ke2= -3.1, ke3= -3.1, ke4= -3.1,

                  %   flow constant through pipe - valve fully open
     kc1= 0.000185, kc2=0.000185,kc3=0.000185,kc4=0.000185,
     vt=1.0,              % speed at which valve size changes
     f00=0.1,               % initial inflow rate
     epsilon=0.00001],      % a small value

 Initial_state =
          [t=0,
    % ODE regime for tank i
           r1=below,r2=below,r3=below,r4=below,
```

```
    % valve motion status out of tank i
           vm1=opening,vm2=opening,vm3=opening,vm4=opening,
    % valve position status out of tank i
           vr1=normal,vr2=trans,vr3=trans,vr4=trans,
    % water depth in tank i
           d1=0.1, d2=0.1, d3=0.1, d4= 0.1,
    % valve opening out of tank i
           p1=0.5, p2=0.005, p3=0.0, p4=0.0],



 evolve(Initial_state, Problem_constants,N,Final_state).




ex2(Initial_state,Problem_constants,N,Final_state) :-
 set_clip(sensitivity,0.0), set_clip(max_narrow,100000),

 Problem_constants =
[delta=0.0512312312321223123212 31,             % maximum time step
        h1=10,h2=9,h3=8,h4=7,   % tank heights

                          % exponent for decay of flow as f(position)
     ke1= -3.1, ke2= -3.1, ke3= -3.1, ke4= -3.1,
```

```
                %   flow constant through pipe - valve fully open
    kc1= 0.000185, kc2=0.000185,kc3=0.000185,kc4=0.000185,
    vt=1.0,                % speed at which valve size changes
    f00=0.1,                 % initial inflow rate
    epsilon=0.00001],        % a small value

Initial_state =
         [t=0,
   % ODE regime for tank i
          r1=below,r2=below,r3=below,r4=below,
  % valve motion status out of tank i
          vm1=opening,vm2=opening,vm3=opening,vm4=opening,
  % valve position status out of tank i
          vr1=normal,vr2=trans,vr3=trans,vr4=trans,
  % water depth in tank i
          d1=0.1, d2=0.1, d3=0.1, d4= 0.1,
  % valve opening out of tank i
          p1=0.5, p2=0.005, p3=0.0, p4=0.0],


lookup([t=1.0],Final_state),
evolve(Initial_state, Problem_constants,N,Final_state).
```

B.2 Evolving a System

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% evolving a hybrid system
% evolve(Initial_state, Problem_constants, Num_steps, Final_state)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  We use contract_vars_BT here to figure out why the current step
%  ends, and then reclaim the stack space used for that.  If
%  we were trying to analyze the system, and ask what parameters
%  had what effect,  we would have to use just   next_step(S0,C,S1),
%   but that  doesn't reuse memory



% evolve(S0,C,N,S2) is true iff the system described by C can evolve
% in N steps from  state S0 to state S1.
evolve(S0,C,N,S2) :-
  evolve0(S0,C,N,S1),
  enforce_ODEs(S1,C,S2),
  copy_discrete_state(S1,S2).
```

```
% evolve0(S0,C,N,S1) is true iff the system described by C can
% evolve in N steps from S0 to S1 *and* S1 is a boundary state
% (i.e. a program or regime changes at S1, or a maximum step length
% is achieved at S1)
evolve0(S0,_C,N,S1) :- {N=0},eqstate(S0,S1).

evolve0(S0,C,N,S2) :-
    opt_next_step(S0,C,S1),
    {N=M+1},
    evolve0(S1,C,M,S2).

% test if two continuous states are equal
eqstate([],[]).
eqstate([A=X|As],[A=Y|Bs]) :-
  eq(X,Y),eqstate(As,Bs).



  eq(T0,T1),eq(A0,A1),eq(Z0,Z1).

eq(A,B) :- (var(A);var(B)),!,A=B.
eq('$INT'(N),'$INT'(M)) :-
  {'$INT'(N) = '$INT'(M)}.
```

```
% next_step(InitialState, ProblemConstants, FinalState)
next_step(S0,C,S1) :-
  enforce_ODEs(S0,C,S1),
  find_state_change(S0,C,S1).

% optimized version that save space and time
opt_next_step(S0,C,S1) :-
    contract_vars_BT([S1],next_step(S0,C,S1)).
```

B.3 ODE Code

B.3.1 Looking up Constants and Current Values of Variables

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ODE code -- note, this uses ODEs on function variables
% to generate constraints on the interval variables in S0 and S1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
enforce_ODEs(S0,C,S1) :-
  is_state(S0),is_state(S1),
  % lookup maximum time step and time at S0 and S1
  lookup([delta=Delta],C),  lookup([t=T0],S0),  lookup([t=T1],S1),
```

```
% and set upper bound on next step
{T1max=T1, T1 < T0+Delta},



% get discrete state at T0 which will determine the ODEs to use
% ri is ODEregime, vmi is valvemotion regime, vri is valve position
% regime
% tank 4's output pipe does not go into another tank, so it can't
% change regime, hence the r4=_R4

lookup([r1=R1,r2=R2,r3=R3,r4=_R4,vm1=VM1,vm2=VM2,vm3=VM3,vm4=VM4,
             vr1=VR1,vr2=VR2,vr3=VR3,vr4=VR4],S0),
% lookup relevant constants for ODEs
lookup([epsilon=E,f00=F00],C),

% VALVE Position ODEs
% create valve position functions on [T0,T1max]
  P=[P1,P2,P3,P4],
  decls(P,function(T0,T1max)),
% put bounds on the range of the function
  bound_functions(P,[0,1]),
% set their values at times T0 and T1
  Ps0=[P10,P20,P30,P40],
  Ps1=[P11,P21,P31,P41],
```

```
    lookup([p1=P10,p2=P20,p3=P30,p4=P40],S0),
    lookup([p1=P11,p2=P21,p3=P31,p4=P41],S1),
    evalall(P,T0,Ps0), evalall(P,T1,Ps1),
  % lookup the valve speed
    lookup([vt=VT],C),
  % add the ODE constraints
    valve_ODE(P1,VT,VM1),
    valve_ODE(P2,VT,VM2),
    valve_ODE(P3,VT,VM3),
    valve_ODE(P4,VT,VM4),


                        % DEPTH and FLOW ODEs
% create depth and flow function variables on [T0,T1max] for the ODEs
D=[D1,D2,D3,D4],
        F=[F1,F2,F3,F4],
        FR=[FRAC1,FRAC2,FRAC3,FRAC4],
    % and constrain them to be analytic functions on [T0,T1max]
        decls(D,function(T0,T1max)),
        decls(F,function(T0,T1max)),
        decls(FR,function(T0,T1max)),
                    % put bounds on their ranges
        bound_functions(D,[0,1000]),
```

```
        bound_functions(F,[0,1000]),  % Note: assumes no backward flow
        bound_functions(FR,[0,1]),
                      % bind values of depth functions at T0 and T1
        Ds0=[D10,D20,D30,D40],
        Ds1=[D11,D21,D31,D41],
        lookup([d1=D10,d2=D20,d3=D30,d4=D40],S0),
        lookup([d1=D11,d2=D21,d3=D31,d4=D41],S1),
        evalall(D,T0,Ds0), evalall(D,T1,Ds1),
                    % lookup static values for tanks
        lookup([h1=H1,h2=H2,h3=H3,h4=H4,
          kc1=KC1,kc2=KC2,kc3=KC3,kc4=KC4,ke1=KE1,ke2=KE2,
                 ke3=KE3,ke4=KE4],C),
  % calculate flow through valves,
% VR - valve regime, FRAC- fraction of full flow through valve, P -
                    %valve position, KE - exponent for valve
        valve_coef(VR1,FRAC1,P1,KE1),
        valve_coef(VR2,FRAC2,P2,KE2),
        valve_coef(VR3,FRAC3,P3,KE3),
        valve_coef(VR4,FRAC4,P4,KE4),

  % apply the ODEs corresponding to each tank
  % Ri = ode governing tank i, Di = depth in tank i,
  % Fi = flow out of tank i, Hi = height of tank i,
  % Pi = valve opening out of tank i, Ki = valve coefficient,
```

```
  % F00 = flow into tank 1


        first_tank( R1,    D1,F1,D2,  F00,H1,KC1,FRAC1,H2,E),
        middle_tank(R2, F1,D2,F2,D3,      H2,KC2,FRAC2,H3,E),
        middle_tank(R3, F2,D3,F3,D4,      H3,KC3,FRAC3,H4,E),
        last_tank(      F3,D4,F4,            KC4,FRAC4).
```

B.3.2 Valve Flow

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Here we describe flow through the valves
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The next 3 lines define the fraction of full flow allowed
% through a valve  (FRAC), as a
% function of the regime the valve is in (normal - mainly open,
% transition - nearly but not quite closed; shut);
% the position of the valve  % (P) - 0 is closed, 1 is open;
%  and the exponent (KE) of the valve description.
% (note: KE is negative)

valve_coef(normal,FRAC,P,KE) :- {[FRAC=exp(KE*((1-P))**3),
```

```
                                                FRAC in [0,1],
                                                 P in [0.01,1] ]}.
valve_coef(trans,FRAC,P,_) :- {[ FRAC in [0,0.05],  P in [0,0.01] ]}.
valve_coef(shut,FRAC,0,_) :- {[FRAC=0.0*FRAC ]}.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Here we describe the valve motions by giving its ODEs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
valve_ODE(P,_VT,halted) :- {[ ddt(P,1) =   0.0*P, P in [0,1] ]}.
valve_ODE(P,VT,opening) :- {[ ddt(P,1) =  VT+0*P, P in [0,1] ]}.
valve_ODE(P,VT,closing) :- {[ ddt(P,1) = NVT + 0*P, P in [0,1],
                                                     NVT= - VT ]}.
```

B.3.3 Tank Descriptions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Here we describe each of the tanks, by giving its ODEs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% D1,D2 water depths, F1 flow from D1->D2, F00 input flow,
%  H1, H2 heights, KC flow constant for pipe,
% FRAC - fraction of full flow valve allows
% P - valve position, E-epsilon for error function
%
% The ODEs change when the water in the lower tank goes from above
% the height of the pipe to below the height of the pipe.  Because the
% ODEs change there, this point is not analytic.  We handle this by
% considering 3 regimes, above, near, below where the water height is
% respectively above, near, or below the in-flow pipe.  Since the
% behaviour here is well understood, Epsilon can be quite small.

% Since square root is non-analytic near zero, we can't use the
% standard % version of the formula for water flow at the point
% where the heights are almost equal.
% For now, we ignore the  case when the lower tank has water height
% nearly  equal to the upper tank's, but we'd have to use this
% technique there.

first_tank(above,D1,F1,D2,F00,H1,KC1,FRAC1,H2,_E) :-
 {[F1=KC1*FRAC1*psqrt(D1-D2+H), ddt(D1,1)=F00-F1, H=H1-H2,
                                       D2 in [H,1000] ]}.

% Here EF is an error function whose range is [0,2*E]
```

```
% and thats all we know
first_tank(near,D1,F1,D2,F00,H1,KC1,FRAC1,H2,E) :-
 decls([EF],function(_,_)),
 {[F1=KC1*FRAC1*psqrt(D1-EF), ddt(D1,1)=F00-F1, H=H1-H2,
                        D2 in [-2*E+H,2*E+H], EF in [0,2*E]  ]}.

first_tank(below,D1,F1,D2,F00,H1,KC1,FRAC1,H2,E) :-
 {[F1=KC1*FRAC1*psqrt(D1), ddt(D1,1)=F00-F1, H=H1-H2, D2 in [E,H]]}.

middle_tank(above,F1,D2,F2,D3,H2,KC2,FRAC2,H3,_E) :-
 {[F2=KC2*FRAC2*psqrt(D2-D3+H), ddt(D2,1)=F1-F2, H=H2-H3,
                                       D3 in [H,1000] ]}.

middle_tank(near,F1,D2,F2,D3,H2,KC2,FRAC2,H3,E) :-
        decls([EF],function(_,_)),
        {[F2=KC2*FRAC2*psqrt(D2-EF), ddt(D2,1)=F1-F2,
        H=H2-H3, D3 in [-2*E+H,2*E+H], EF in [0,2*E] ]}.

middle_tank(below,F1,D2,F2,D3,H2,KC2,FRAC2,H3,E) :-
 {[F2=KC2*FRAC2*psqrt(D2), ddt(D2,1)=F1-F2, H=H2-H3, D3 in [E,H]]}.

last_tank(F3,D4,F4,KC4,FRAC4) :-
 {[F4=KC4*FRAC4*psqrt(D4), ddt(D4,1)=F3-F4 ]}.
```

B.4 Specifying Transitions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Specification of Transitions
%
%  find_state_change(S0,C,S1)
%
% Find state change -- here we look for regime/state change
% constraints being satisfied
%
% Key idea -- for this to work well, we need to ensure that a
% change is detected at time t
% only if no earlier change has occurred at some time before
% t.... We don't need this for soundness, but we do need it to
% rule out spurious paths.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% one of the tanks changes regime
find_state_change(S0,C,S1) :-
           find_flow_state_change(r1,h1,h2,d2,C,S0,S1).
find_state_change(S0,C,S1) :-
           find_flow_state_change(r2,h2,h3,d3,C,S0,S1).
```

```
find_state_change(S0,C,S1) :-
           find_flow_state_change(r3,h3,h4,d4,C,S0,S1).


% one of the valves finishes opening or closing, or changes regime
% among shut, transition, normal
find_state_change(S0,C,S1) :-
                 find_valve_state_change(p1,vr1,vm1,C,S0,S1).
find_state_change(S0,C,S1) :-
                  find_valve_state_change(p2,vr2,vm2,C,S0,S1).
find_state_change(S0,C,S1) :-
                  find_valve_state_change(p3,vr3,vm3,C,S0,S1).
find_state_change(S0,C,S1) :-
                  find_valve_state_change(p4,vr4,vm4,C,S0,S1).



% PROGRAMMED STATE CHANGES
find_state_change(S0,C,S1) :-
                      find_program_state_change(v1,d2,C,S0,S1).
find_state_change(S0,C,S1) :-
                       find_program_state_change(v2,d3,C,S0,S1).
find_state_change(S0,C,S1) :-
                       find_program_state_change(v3,d4,C,S0,S1).


%  no regime or state changes before the time limit is reached
```

```
find_state_change(S0,C,S1) :- find_step_size_change(S0,C,S1).


% succeeds if the maximum step size has been reached...

%  no regime or state changes before the time limit is reached
% no change in discreet state - update discrete state called
% with no change in r1 so no discrete changes at all
find_step_size_change(S0,C,S1) :-
  lookup([t=T0],S0),lookup([t=T1],S1),lookup([delta=Delta],C),
  update_discrete_state(vr1,VR10,VR10,S0,S1),
  {T1= T0+Delta}.  %  Only continuous state changes
```

B.5 User Program

```
% if the user's program starts a valve moving, this handles
% any valve position regime change if needed
%       Valve#, WaterDepth, constants, old state, new state
find_program_state_change(Vi,Dj,_C,S0,S1) :-
  lookup([Dj=D],S1),
```

```
  update_discrete_state(Vi,_VM_before,VM_after,S0,S2),
  update_discrete_state(Vi,VR_before,VR_after,S2,S1),
  ((VR_before = shut, VM_after = opening, VR_after = trans);
       ( VR_before \= shut, VR_before = VR_after)),
  regulate_valve1(D,VM_after).

% Program for regulating depth in tank j using valve i (with j=i+1)
%  if depth in tank j > 1.1 and then close valve i
%  if depth in tank j < 0.9 and then open valve i
% e.g. find_program_state_change(v2,d3,C,S0,S1).

% Here D is the analog value of the the depth of water in the tank
% This allows a phantom state change from closing->closing
% and from opening->opening
% This is the user's program, it can do anything it wants...
% We have to try to prove it safe or show it is unsafe, or claim
% ignorance -- authoritatively.
regulate_valve1(D,VM_after) :- VM_after=closing, {D = 1.1}.
regulate_valve1(D,VM_after) :- VM_after=opening, {D = 0.9}.
```

B.6 Finding Regime Changes

```
% Detection of regime change due to tank depth exceeding input
% pipe height % e.g. find_flow_state_change(r2,h2,h3,d3,C,S0,S1).
% note j=i+1 here and Ri in {above,near,below}

find_flow_state_change(Ri,Hi,Hj,Dj,C,S0,S1) :-
  lookup([Hi=H1,Hj=H2],C),  lookup([Dj=D],S1),
  update_discrete_state(Ri,R_before,R_after,S0,S1),
  flow_state_change(R_before,D,R_after,H1,H2).



% We use hysteresis in our analysis to avoid an infinite loop of zero
% time state changes as it goes from near to below and back again.
% We'll probably need to do this in several cases of regime change
flow_state_change(below,D,near,H1,H2) :-
  E=0.00001,  {D = H1-H2-E}.
flow_state_change(near,D,below,H1,H2) :-
  E=0.00001,  {D = H1-H2-2*E}.
flow_state_change(above,D,near,H1,H2) :-
  E=0.00001,  {D = H1-H2-E}.
flow_state_change(near,D,below,H1,H2) :-
  E=0.00001,  {D = H1-H2+2*E}.

% check to see if valve n has hit a state change
% and if so, update the discrete part of S1 accordingly
```

```
% e.g. find_valve_state_change(p2,v2,C,S0,S1).
% v2 in {opening,closing,halted}, p2 in [0,1],
% note that this is a regime change, not a state change.
% Also, we have to handle the regime change from shut to transition
% to normal The transisition to shut implies a transition to halted,
% but not vice versa.

% We use S2 as a temporary state to allow changes to 2
% discrete variables  in one change

find_valve_state_change(Pn,VRn,VMn,_C,S0,S1) :-
  lookup([Pn=P_before],S0),lookup([Pn=P_after],S1),
  update_discrete_state(VMn,VM_before,VM_after,S0,S2),
  update_discrete_state(VRn,VR_before,VR_after,S2,S1),
  valve_state_change(VM_before,VR_before,P_before,VM_after,
                                            VR_after,P_after).



% regime change rules for valve motion (and in closing case, position)
valve_state_change(opening,normal,_P_before,halted,normal,P_after) :-
                                            {P_after=1}.
valve_state_change(closing,trans,_P_before,halted,shut,P_after) :-
                                            {P_after=0}.
```

```
% regime change rules for valves position
valve_state_change(opening,trans,_P_before, opening,normal,P_after)
                                                       :-  {P_after=0.01}.
valve_state_change(opening,normal,_P_before, constant,normal,P_after)
                                                       :- {P_after=1.0}.
valve_state_change(opening, shut,_P_before,  opening,trans,P_after)
                                                       :- {P_after=0.0}.
valve_state_change(closing, normal,_P_before, closing,trans,P_after)
                                                       :- {P_after=0.01}.
valve_state_change(closing,trans,_P_before,   constant,shut,P_after)
                                                       :-  {P_after=0.0}.
```

B.7 Helper Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% helper functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lookup([],_).
lookup([A|As],R) :- lookup_one(A,R),lookup(As,R).
    lookup_one(A=V,[A=W|_]) :- eq(V,W).
lookup_one(A=V,[B=_|R])  :- A\==B, lookup_one(A=V,R).
```

```
copy_discrete_state(S0,S1) :-
  update_discrete_state(vr1,V,V,S1,S2).

% for statename V, unify all discrete states in S1 to corresponding
% ones in S0
% except for the V state. Its value in S0 is V0 and in S1 is V1.
update_discrete_state(V,V0,V1,S0,S1) :-
  is_state(S0),is_state(S1),
  update_ds(S0,S1,V,V0,V1), % get the value of Vn in S0 and in S1
  Dvals=[r1,r2,r3,r4, vm1,vm2,vm3,vm4, vr1,vr2,vr3,vr4],
  remove(V,Dvals,CopyVals),
  copy_ds(S0,S1,CopyVals).
% make sure S0,S1 have same values for other discrete variables

% makes sure that V has value V0 in S0 and V1 is S1,
% no other constraints.
update_ds([V=V0|_Cs],[V=V1|_Ds],V,V0,V1).
update_ds([A=_|Cs],[A=_|Ds],V,V0,V1) :- A\==V,!,
  update_ds(Cs,Ds,V,V0,V1).

copy_ds([],[],_).
copy_ds([V=V0|Cs],[V=V0|Ds],CopyVals) :- member(V,CopyVals),!,
```

```
  copy_ds(Cs,Ds,CopyVals).
copy_ds([_|Cs],[_|Ds],CopyVals) :-
  copy_ds(Cs,Ds,CopyVals).

remove(_,[],[]).
remove(X,[X|Ys],Ys).
remove(X,[A|As],[A|Bs]) :- X\==A,remove(X,As,Bs).



print_the_list([]).
print_the_list([A|As]) :- print(A),nl,print_the_list(As).



evalall([],_,[]).
evalall([F|Fs],T,[C|Cs]) :- {[eval(F,T)=C]}, evalall(Fs,T,Cs).

bound_functions([],_).
bound_functions([F|Fs],I) :- {[ F in I ]}, bound_functions(Fs,I).

% is_state could be expanded to check if state is legal -
% are values and  regimes compatible?
is_state( [t=_,
       r1=_,r2=_,r3=_,r4=_,             % ODE regime for tank i
       vm1=_,vm2=_,vm3=_,vm4=_, % valve motion status out of tank i
```

```
vr1=_,vr2=_,vr3=_,vr4=_,      % valve i position status
d1=_, d2=_, d3=_, d4= _,        % water depth in tank i
p1=_, p2=_, p3=_, p4=_]).     % valve opening out of tank i
```

Bibliography

[ACH$^+$95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Grossman et al. [GNRR93], pages 209–229.

[Act96] Forman S. Acton. *Real computing made real: Preventing Errors in Scientific and Engineering calculations.* Princeton University Press, Princeton, New Jersey, 1996.

[AD91] Rajeev Alur and David Dill. The theory of timed automata. In de Bakker et al. [dBHdRR91], pages 45–73.

[AH83] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[AHS96] Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors. *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[AK93] E. Adams and U. Kulisch, editors. *Scientific Computing with Automatic Result Verification*. Academic Press, 1993.

[AKL+99] Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems V*, volume 1567 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[AKNS95] Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[AKNS97] Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems IV*, volume 1273 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[AL91] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. In de Bakker et al. [dBHdRR91], pages 1–27.

[AP04] Rajeev Alur and George J. Pappas, editors. *Hybrid Systems: Communication and Control, HSCC 2004*, volume 2993 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[Ars] Hossein Arsham. Sensitivity analysis. A collection of recent developments on sensitivity analysis in several fields, http://ubmail.ubalt.edu/~harsham/senanaly/SenAnaly.htm.

[BBB+00] Andrea Balluchi, Luca Benvenuti, Maria D. Di Benedetto, Guido M. Miconi, Ugo Pozzi, Tiziano Villa, Howard Wong-Toi, and Alberto L. Sangiovanni-Vincentelli. Maximal safe set computation for idle speed control of an automotive engine. In Lynch and Krogh [LK00], pages 32–44.

[Ben95] Frédéric Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 1995.

[BF04] Amit Bhatia and Emilio Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In Alur and Pappas [AP04], pages 142–156.

[BFH+04] Călin Belta, Peter Finin, Luc C.G.J.M. Habets, Ádám M. Halász, Marcin Imielinski, R. Vijay Kumar, and Harvey Rubin. Understanding the bacterial stringent response using reachability analysis of hybrid systems. In Alur and Pappas [AP04], pages 111–125.

[BL91] Richard Berber and Insup Lee. Specification and analysis of resoruce-bound real-time systems. In de Bakker et al. [dBHdRR91], pages 371–396.

[BL02] Paul I. Barton and Cha Kun Lee. Modeling, simulation, sensitivity analysis and optimization of hybrid systems. *ACM Transactions on Modeling and Computer Simulation*, 12(4):256–289, Oct 2002.

[BMvH94] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proc. 1994 International Symposium*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press.

[BO97] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, Jul 1997.

[Boh96] Gerd Bohlender. Literature on enclosure methods and related topics. Technical report, Institut für Angewandte Matematik, Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, Sep 1996. http://www.uni-karlsruhe.de/∼Gerd.Bohlender/litlist.html an earlier version appeared in [AK93].

[BSV01] Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors. *Hybrid Systems: Communication and Control, HSCC 2001*, volume 2034 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[Bur24] J. C. Burkill. Functions of intervals. *Proceedings of the London Mathematical Society*, 22:375–446, 1924.

[CF00] Angelo E.M. Ciarlini and Thom Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *ACM SIGSIM Conference on AI, Simulation and Planning (AIS '2000)*, Mar 2000.

[Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, NY, 1978.

[Cle87] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.

[CW98] Dmitri Chiriaev and G. William Walster. Interval arithmetic specification. www.mscs.mu.edu/∼globsol/Papers/spec.ps, May 1998.

[CWA+91] Mats Carlsson, Johan Widén, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, PO Box 1263; S-16128 KISTA, Sweden, Oct 1991.

[dBHdRR91] J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice. REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, Mook, The Netherlands, Jun 1991. REX (Research and Education in Concurrent Systems, Springer Verlag.

[Dec00] Rina Dechter, editor. *Principles and Practice of Constraint Programming - CP2000*, volume 1894 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[DEDC96] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: The Standard; Reference Manual*. Springer Verlag, 1996.

[DGS] Akash Deshpande, Aleks Göllü, and Luigi Semenzato. *The SHIFT Programming Language and Run-time System for Dynamic Networks of Hybrid Automata*. Department of Electrical Engineering and Computer Sciences; University of California at Berkeley, Berkeley, CA 94720. http://www.path.berkeley.edu/shift/doc/ieeshift.ps.gz.

[Dia02] Daniel Diaz. *GNU Prolog Manual*, 1.7 edition, Sep 2002.

[DJvH02] Yves Deville, Micha Janssen, and Pascal van Hentenryck. Consistency techniques in ordinary differential equations. *Constraints*, 7(3):289–315, Jul 2002.

[DL97] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In Oded Maler, editor, *International Workshop on Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 154–170. Springer Verlag, 1997.

[DN00] J.M. Davoren and Anil Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88(7):985–1010, Jul 2000.

[DP99] Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In Rance Cleaveland, editor, *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer Verlag, 1999.

[DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3), 2001.

[Eda01] Abbas Edalat. Exact real number computation using linear fractional transformations. http://www.doc.ic.ac.uk/exact-computation/exactarithmeticfinal.ps.gz, 2001. Final report on EPSRC grant GR/L43077/01.

[EH02] Abbas Edalat and Reinhold Heckmann. Computing with real numbers: (i)LFT approach to real computation, (ii) domain-theoretic model of computational geometry. In Gilles Barthe, Peter Dybjer, Luís Pinto, and Joao Saraiva, editors, *Applied Semantics: International summer school, APPSEM 2000*, volume 2395 of *Lecture Notes in Computer Science*, pages 193–267. Springer Verlag, 2002.

[Fah70] David Arthur Fahrland. Combined discrete event continuous systems simulation. *Simulation*, 14(2):61–72, Feb 1970.

[FI04] Ansgar Fehnker and Franjo Ivančić. Benchmarks for hybrid systems verification. In Alur and Pappas [AP04], pages 326–341.

[Fre96] Eugene C. Freuder, editor. *Principles and Practice of Constraint Programming - CP96*, volume 1118 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[GFB99] Santos Galán, William F. Feehery, and Paul I. Barton. Parametric sensitivity functions for hybrid discrete/continuous systems. *Applied Numerical Mathematics*, 31(1):17–47, 1999.

[GJEK87] G. Günther-Jürgens, P. Endebrock, and R. Klatte. RESI: practical experience in computer arithmetic. In Kaucher et al. [KKU87], pages 186–198.

[GJS96] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. `Hybrid cc`, hybrid automata and program verification. In Alur et al. [AHS96], pages 52–63.

[GJSB95] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel G. Bobrow. Programming in hybrid constraint languages. In Antsaklis et al. [AKNS95], pages 226–251.

[GNRR93] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Gol91] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar 1991.

[Hay03] Brian Hayes. A lucid interval. *American Scientist*, 91(6):484–488, Nov-Dec 2003.

[HC88] Timothy J. Hickey and Jacques Cohen. Automating program analysis. *JACM*, 35(1):185–220, 1988.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Symposium on Logic in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Society Press, 1996.

[Hen02] Pascal Van Hentenryck, editor. *Principles and Practice of Constraint Programming - CP2002*, volume 2470 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[HG91] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In de Bakker et al. [dBHdRR91], pages 291–314.

[HHMWT00] Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HyTech: Hybrid systems analyis using interval numerical methods. In Lynch and Krogh [LK00], pages 130–144.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(?):110–122, 1997.

[HHWT98] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.

[Hic94] Timothy J. Hickey. CLP(F) and constrained ODEs. In *Proceedings of the Workshop on Constraints and Modelling*, 1994.

[Hic00a] Timothy J. Hickey. Analytic constraint solving and interval arithmetic. In *POPL'00 ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, pages 338–351, 2000. published as vol. 27 of SIGPLAN notices.

[Hic00b] Timothy J. Hickey. CLIP: A CLP(Intervals) dialect for metalevel constraint solving. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages: PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 200–214. Springer Verlag, 2000. A later version is [Hic01].

[Hic01] Timothy J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001. http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2001/S01-02/JFLP-A01-07.pdf.

[HJ] Timothy J. Hickey and Qun Ju. clip 1.0 a CLP(Intervals) interpreter, based on Sicstus Prolog. interval.sourceforge.net/interval/prolog/clip.

[HJ97] Timothy J. Hickey and Qun Ju. Efficient implementation of interval arithmetic narrowing using IEEE arithmetic. Technical report, Brandeis University CS Dept, April 1997. www.cs.brandeis.edu/~tim/narrow_multiply/paper.ps.

[HJvE01] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: from principles to implementation. *JACM*, 48(5):1038–1068, Sep 2001.

[HMP91] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In de Bakker et al. [dBHdRR91], pages 226–252.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol95] C. Holzbaur. *OFAI CLP(Q,R) Manual*. Austrian Research Institute for Artificial Intelligence, Vienna, 1.3.3 edition, 1995. TR-95-05.

[HQvE00] Timothy J. Hickey, Zhi Qiu, and Maarten H. van Emden. Interval constraint plotting for interactive visual exploration of implicitly defined relations. *Reliable Computing*, 6(1):81–92, 2000.

[HR94] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797, October 1994.

[HS81] E.R. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT*, 21:203–211, 1981.

[HS91] Timothy J. Hickey and Donald A. Smith. Toward the partial evaluation of CLP languages. In *Proceedings of the 1991 SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 43–51. ACM SIGPLAN, 1991. Vol. 26, Issue 9 of ACM SIGPLAN notices.

[HvEW98] Timothy J. Hickey, Maarten H. van Emden, and Huan Wu. A unified framework for interval constraints and interval arithmetic. In Michael

Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 1998.

[HW99a] Timothy J. Hickey and David K. Wittenberg. Validated constraint compilation. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 482–483, 1999. A longer version is [HW99b].

[HW99b] Timothy J. Hickey and David K. Wittenberg. Validated constraint compilation. Technical Report CS-99-201, Computer Science Department, Brandeis University, April 1999. URL: www.cs.brandeis.edu/~tim/Papers/cs99201.ps.gz.

[HW03] Timothy J. Hickey and David K. Wittenberg. Using analytic CLP to model and analyze hybrid systems. Technical Report CS-03-240, Brandeis University, 2003. http://www.cs.brandeis.edu/~dkw/papers/cs03-240.pdf, Accepted to FLAIRS 04.

[HW04] Timothy J. Hickey and David K. Wittenberg. Rigorous modeling of hybrid systems using interval arithmetic constraints. In Alur and Pappas [AP04], pages 402–416.

[Hyv89] E. Hyvönen. Constraint reasoning based on interval arithmetic. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1193–1198, Detroit, USA, 1989.

[IEE85] IEEE. IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN*, 22(2):9–25, 1985.

[Jaf99] Joxan Jaffar, editor. *Principles and Practice of Constraint Programming - CP99*, volume 1713 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[JELS99] Karl Henrik Johansson, Magnus Egerstedt, John Lygeros, and Shankar Sastry. On the regularization of zeno hybrid automata. *System and Control Letters*, 38:141–150, 1999.

[JKDEW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer Verlag, 2001.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on the Principles of Programming Languages*, pages 111–119, 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programing Languages and Systems*, 14(3):339–395, 1992.

[Ju98] Qun Ju. *A Sound Interval Constraint Logic Programming System*. PhD thesis, Brandeis University, May 1998.

[JvHD02] Micha Janssen, Pascal van Hentenryck, and Yves Deville. A constraint satisfaction approach for enclosing solutions to parametric ordinary differential equations. *SIAM Journal on Numberical Analysis*, 40(5):1896–1939, 2002.

[Kah96] William Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Technical report, EECS, University of California, Berkeley, 1996.

[Kea96] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[KKU87] Edgar Kaucher, Ulrich Kulisch, and Christian Ullrich, editors. *Computerarithmetic: Scientific Computation and Programming Languages*. B. G. Teubner, Stuttgart, 1987.

[KSF+99] Stefan Kowalewski, Olaf Stursberg, M. Fritz, H. Graf, Ingo Hoffman, Jorg Preußig, M. Remelhe, S. Simon, and H. Treseler. A case study in tool-aided analysis of discretely controlled continuous systems: The two tanks problem. In Antsaklis et al. [AKL+99], pages 163–185.

[Kui93] Benjamin J. Kuipers. Qualitative simulation: Then and now. *Artificial Intelligence*, 59:133–140, 1993.

[KZB+90] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings, 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–19. IEEE, 1990.

[KZB+91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, november 1991.

[LA90] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In Cynthia Dwork, editor, *ACM Symposium on Principles of Distributed Computing, PODC'90*, pages 265–280. ACM, 1990.

[Lam86] Leslie Lamport. Buridan's principle. Revised from a version of Oct. 1984; http://research.microsoft.com/users/lamport/pubs/buridan.ps, Jan 1986.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. Also available as DEC SRC Research Report 79.

[LGS96] John Lygeros, Datta N. Godbole, and Shankar S. Sastry. A game theoretic approach to hybrid system design. In Alur et al. [AHS96], pages 1–12.

[LK00] Nancy Lynch and Bruce H. Krogh, editors. *Hybrid Systems: Communication and Control, HSCC 2000*, volume 1790 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[LL98] John Lygeros and Nancy Lynch. Strings of vehicles: Modeling and safety conditions. In Sastry and Henzinger [SH98], pages 273–288.

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming*. Springer Verlag, second, expanded edition, 1987.

[Loh87] Rudolf J. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In Kaucher et al. [KKU87], pages 255–286.

[LSV01] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata revisited. In Benedetto and Sangiovanni-Vincentelli [BSV01], pages 403–417.

[LSVW99] Nancy Lynch, Roberto Segala, Frits W. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. Technical Report CSI-R9907, Computing Science Institue Nijmegen; Faculty of Mathematics and Informatics; Catholic University of Nijmegen, Toernooivveld 1; 6525 ED Nijmegen; The Netherlands, Apr 1999.

[LT88] Nancy Lynch and Mark Tuttle. An introduction to Input/Output Automata. Technical Report MIT/LCS/TM373, MIT LCS, 1988.

[LT04] Patrick Lincoln and Ashish Tiwari. Symbolic systems biology: Hybrid modeling and analysis of biological networks. In Alur and Pappas [AP04], pages 660–672.

[LV91] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In de Bakker et al. [dBHdRR91], pages 397–446.

[LvG00] Michael Lerch and Jürgen Wolff von Gudenberg. `fi_lib++`: Specification, implementation, and test of a library for extended interval arithmetic. In *Proceedings of the Fourth Conference on Real Numbers and Computers*, 2000. http://www.imada.sdu.dk/˜kornerup/RNC4/papers/p03.ps.

[MA85] S. Markov and R. Angelov. An interval method for systems of ODE. In Karl Nickel, editor, *Interval Mathematics 1985*, volume 212 of *Lecture Notes in Computer Science*, pages 103–108. Springer Verlag, 1985.

[MBT01] Ian M. Mitchell, Alexandre M. Bayen, and Claire J. Tomlin. Validating a Hamilton-Jacobi approximation to hybrid system reachable sets. In Benedetto and Sangiovanni-Vincentelli [BSV01], pages 418–432.

[Mik91] Solomon G. Mikhlin. *Error Analysis in Numerical Processes*. John Wiley & Sons, 1991. Translated by Reinhard Lehmann.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[MMP91] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In de Bakker et al. [dBHdRR91], pages 447–484.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Mos99] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In Vaandrager and van Schuppen [VvS99], pages 165–177.

[MP98] Michael Maher and Jean-Francois Puget, editors. *Principles and Practice of Constraint Programming - CP98*, volume 1520 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[MP03] Oded Maler and Amir Pnueli, editors. *Hybrid Systems: Communication and Control, HSCC 2003*, volume 2623 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[MR95] Ugo Montanari and Francesca Rossi, editors. *Principles and Practice of Constraint Programming - CP95*, volume 976 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[Neo04] Natasha A. Neogi. Dynamic partitioning of large discrete event biolgical systems for hybrid simulation and analysis. In Alur and Pappas [AP04], pages 463–476.

[Neu90] Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

[Neu01] Arnold Neumaier. *Introduction to Numerical Analysis*. Cambridge, 2001.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In Grossman et al. [GNRR93], pages 149–178.

[NS91] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [dBHdRR91], pages 526–548.

[NSY91] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [dBHdRR91], pages 549–572.

[ora85] Department of defense trusted computer system evaluation criteria. DOD 5200.28 STD, Department of Defense, Washington DC, Dec 1985. Usually referred to as the "orange book".

[OV93] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[Par04] Pablo Parrilo. Sum of squares relaxation for system analysis. Special Session on Robustness, Abstraction, and Computation part of HSCC04 [AP04], 2004.

[PBM03] Astrid A. Prinz, Cyrus P. Billimoria, and Eve Marder. An alternative to hand-tuning conductance-based models: Construction and analysis of data bases of model neurons. *Journal of Neurophysiology*, 90:3998–4015, Dec 2003.

[Pod00] Andreas Podelski. Model checking as constraint solving. In Jens Palsberg, editor, *Proceedings of SAS'2000: Static Analysis Symposium*, Jun 2000.

[Pri97] Doug Priest. Differences among IEEE 754 implementations. Appendix D to [Gol91], 1997. available at `http://www.validlab.com/goldberg/addendum.html`.

[Pro95] ISO Prolog standard iso/iec 13211-1, information technology — programming languages — prolog — part 1: General core. available from www.iso.org/iso/en/, 1995.

[Res88] Bell Northern Research. *BNR Prolog user guide and reference manual*. Bell Northern Research, 1988.

[Ros03] Francesca Rossi, editor. *Principles and Practice of Constraint Programming - CP2003*, volume 2833 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[SDJ+91] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In de Bakker et al. [dBHdRR91], pages 640–675. Work of the Oxford University Timed CSP Group.

[sen04] *Fourth International Conference on Sensitivity Analysis of Model Output*, Santa Fe, New Mexico, 2004. http://www.samo2004.org.

[SH90] Donald A. Smith and Timothy J. Hickey. Partial evaluation of a CLP language. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference in Logic Programming*, pages 119–138, 1990.

[SH98] Shankar Sastry and Thomas A. Henzinger, editors. *Hybrid Systems: Computation and Control - HSCC'98*, volume 1386 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, Sep 1993.

[SKHP97] Olaf Stursberg, Stefan Kowalewski, Ingo Hoffman, and Jorg Preußig. Comparing timed and hybrid automata as approximations of continuous systems. In Antsaklis et al. [AKNS97], pages 361–377.

[Smo97] Gert Smolka, editor. *Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[Sta96] Ole Stauning. Enclosing solutions of ordinary differential equations: with applications. Technical Report IMM-REP-1996-18, Technical University of Denmark, 1996. http://www.imm.dtu.dk/documents/ftp/tr96/tr18_96.abstract.html.

[Sys96] Applied Logic Systems. *CLP(RI) reference manual*. Applied Logic Systems, 1996.

[Tay97] John R. Taylor. *An introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, second edition, 1997.

[TG02] Claire J. Tomlin and Mark R. Greenstreet, editors. *Hybrid Systems: Communication and Control, HSCC 2002*, volume 2289 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[TLS99] Claire Tomlin, John Lygeros, and Shankar Sastry. Computing controllers for nonlinear hybrid systems. In Vaandrager and van Schuppen [VvS99], pages 238–255.

[TLS00] Claire J. Tomlin, John Lygeros, and S. Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proceedings of the IEEE*, 88(7):949–970, 2000.

[Urb96] Luis Urbina. Analysis of hybrid systems in CLP($\mathcal{R}$). In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, volume 1118 of *Lecture Notes in Computer Science*, pages 451–467. Springer Verlag, Aug 1996.

[vE97a] Maarten H. van Emden. Canonical extensions as common basis for interval constraints and interval arithmetic. In *Proceedings of the Sixth French Conference on Logic and Constraint Programming*, Orléans, France, 1997.

[vE97b] Maarten H. van Emden. Value constraints in the CLP Scheme. *Constraints*, 2:163–183, 1997.

[vH89] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[vHMB98] Pascal van Hentenryck, Laurent Michel, and Frédéric Benhamou. Newton: Constraint programming over nonlinear constraints. *Science of Computer Programming*, 30(1–2):83–118, 1998.

[vHMD97] Pascal van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.

[vNG47] J. von Neumann and H. H. Goldstine. Numerical inverting of matrices of high order. *Bulletin of the American Mathematical Society*, 53:1021–1099, 1947.

[VvS99] Frits W. Vaandrager and Jan H. van Schuppen, editors. *Hybrid Systems: Communication and Control, HSCC 1999*, volume 1569 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Wal01] Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP2001*, volume 2239 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[WH04] David K. Wittenberg and Timothy J. Hickey. *Notes on using CLIP*, 2004. currently a very early draft.

[Wit03] David K. Wittenberg. On unverifiable facts. *Brandeis Graduate Journal*, 1(1), 2003. http://www.brandeis.edu/gsa/gradjournal/2003/pdf/wittenberg.pdf.

[Woo91] William G. Wood. A specification of the cat and mouse problem. In de Bakker et al. [dBHdRR91], pages 676–686.

[You31] R. C. Young. The algebra of many-valued quantities. *Mathematische Annalen*, 104:260–290, 1931.

[ZJLS01] Jun Zhang, Karl Henrik Johansson, John Lygeros, and Shankar Sastry. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control*, 11(5):435–451, 2001.