VERIFICATION OF ANALOG AND MIXED-SIGNAL CIRCUITS USING SYMBOLIC METHODS

by

David C. Walter

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2007

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

David C. Walter

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	Priyank Kalla
	Ganesh Gopalakrishnan
	Konrad Slind
	Reid R. Harrison

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of David C. Walter in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

With the rapidly increasing complexity of hardware, traditional validation techniques are becoming insufficient. This has led to a substantial interest in the formal verification of digital components. There has been relatively little research, however, into the application of formal verification methods to the analog/mixed-signal domain. Therefore, the overall goal of this work is to provide a system for efficient and meaningful analysis of analog/mixed-signal circuits. This encompasses two major efforts: modeling and symbolic analysis.

The continuous nature of analog circuits requires a modeling method that is capable of representing continuous behavior and the discrete nature of digital circuits requires a modeling method that is capable of representing discrete behavior. This dual requirement necessitates a hybrid model—a model that can simultaneously represent continuous and discrete behavior. This work details the development of a specialized hybrid Petri net model with capabilities similar to hybrid automata.

Analysis is greatly complicated by the addition of continuous behavior to the model. To help alleviate this, infinite numbers of states are often grouped into equivalence classes represented by symbolic structures. The analysis methods described here represent ranges of continuous variables using groups of inequalities which are then either mapped to Binary Decision Diagram variables so that necessary operations can be performed efficiently, or handed over to an advanced Satisfiability Modulo Theories solver for analysis.

After describing the verification system in detail, experiences applying the techniques to several case studies are described and performance results are provided.

To my family and friends.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation is the culmination of six years of graduate research and eleven years as a student at the University of Utah. Of course, none of this would have been possible without the support of so many people and organizations.

I must first thank my research advisor, Chris Myers, for his several years of patient guidance and support. It is truly difficult to express my gratitude to Chris for his invaluable guidance of this research and reliable advice in navigating the academic world. I have learned many lessons throughout the years with Chris as an advisor. Thank you also to Ching and John Myers for their friendship.

Life in the office was always kept interesting thanks to my fellow students Scott Little, Nicholas Seegmiller, Vijay Durairaj, Sivaram Gopalakrishnan, Nathan Barker, Nam Nguyen, Chris Condrat, and Namrata Shekhar, Robert Thacker, Kevin Jones, Curt Nelson, and Yanyi Zhao. They have been great resources for sharing ideas and getting assistance. Special thanks also go to Scott Little, Nicholas Seegmiller, and Kevin Jones for their significant contributions to this research.

My family and friends have been a constant source of support and motivation over the years. Particularly my parents, Deborah and Dieter Walter, who have provided wise guidance. I must also thank my brothers, Derek and Daniel Walter—Derek for his pessimism and Daniel for his optimism; and thank you to Ena Ladi and Kate Walter for providing some balance. Thank you to my close friends Paul Alexander, Kosta Damevski, David Goldberg, and Jason Waltman for their understanding during frustrating times.

The University of Utah has been very good to me over the years. Thank you to the staff and faculty in the School of Computing for providing an excellent learning environment. In particular, I'd like to thank Sandy Hiskey and Karen Feinauer who have proven to be irreplaceable resources. I'm very proud to be associated with the University of Utah.

Thank you to Tomohiro Yoneda and Robert Kurshan, and my committee members Reid Harrison, Priyank Kalla, Konrad Slind, and Ganesh Gopalakirshnan for there contributions to this research. Finally, thank you to the Semiconductor Research Corporation for their many years of support under contracts 2002-TJ-1024 and 2005-TJ-1357.

CHAPTER 1

INTRODUCTION

1.1 Context

Society increasingly relies on computing devices for common tasks. Often these tasks are safety-critical. Motor vehicles contain up to a hundred computing devices for controlling everything from anti-lock break systems to the windshield wipers. Medical devices for monitoring patients as well as performing life-saving procedures rely on computing devices and specialized sensors. With this dependence on technology, it is increasingly important that the computing devices that we rely on for so many tasks operate according to specification.

The approaches to ensuring that the computing devices operate correctly vary depending on the type of device. Computing devices come in a range of flavors including complex microprocessors with millions of transistors performing mainly binary calculations, embedded systems that are capable of performing digital or analog operations and run specialized software, and field programmable gate arrays (FPGAs) which can be programmed on the fly to perform specific tasks. Analog and digital circuits are increasingly integrated within computer systems allowing for processing of analog signals using digital circuits. In fact, advances in technology have resulted in the integration of digital and analog circuits on the same *integrated circuit* (IC). Additionally, circuits on these ICs can be both analog and digital. These circuits are known as *mixed-signal circuits*. This dissertation focuses on a methodology for validating analog and mixed-signal (AMS) circuits.

Currently, to remove flaws that are introduced during the design process, simulation and testing methods are used to validate circuit behavior. In simulation, theoretical paths through the model are executed to approximate the actual behavior. In testing, specially designed test cases are applied to the actual system under appropriate environmental conditions to ensure proper behavior. Analog circuit validation is typically performed using SPICE-level simulations, and mixed-signal validation can be done using VHDL-

AMS or Verilog-AMS simulation though it is often done in a more ad hoc way. While simulation and testing are often sufficient, they rarely test the full functionality and interactions of the system instead focusing on particular corner cases.

In the AMS circuit domain, there are additional reasons for unexpected behavior. These errors are primarily due to the fact that in ICs individual devices built on the semiconductor chip can vary widely in value chip to chip depending on the manufacturing process. These variances are uncontrollable by the designer, whereas during board level design, components are tested and binned according to value. Furthermore, depending on doping gradients, device properties can vary across each individual IC. Specialized design techniques like using ratios of resistors, using devices with matched sizes, and using large devices so that the statistical variations become insignificant can help to account for these issues.

These additional sources of error further necessitate the need for techniques to ensure that AMS circuits adhere to design specifications. To attempt to account for variations introduced during the manufacturing process, Monte Carlo or random simulations are often performed. However, it has been cited that digital and analog circuits are integrated in about 75 percent of chips designed today and that 50 percent of the errors in these chips are due to problems in the analog portion [52]. Therefore, further improvements in AMS circuit validation methodology are very important.

1.2 Formal Verification

There are two main categories of formal verification: logical inference and model checking. Logical inference relies on formal mathematical proofs to reason about the system. Theorem proving software tools are often used to help automate the process [23], but logical inference is a time-consuming process that normally requires a verification expert with considerable mathematical experience. In model checking, a property is tested against a model of a system [27]. Construction of the system models and properties can be a manual process but is often and ideally automated. Model checking utilizes nondeterminism and state space exploration methods (calculation of all possible reachable states) to validate designs over a range of parameters and initial conditions simultaneously. Nondeterminism in the models used for model checking even allows conditions to change over the course of the execution. These techniques, therefore, provide a promising mechanism to validate designs in the face of noise and uncertain parameters.

Model checking often encounters the state space explosion problem—so many states are found that the resources of the computer (specifically memory) are fully depleted. To avoid this, specialized model checking methods have been developed that utilize *symbolic execution* and/or *abstraction*. In symbolic analysis methods, arbitrary symbols are used to represent ranges of values with the goal of making state space exploration more efficient by reducing the size of the state representation. In the digital circuit verification domain, Boolean methods such as *binary decision diagrams* (BDDs) [22] and SAT solvers [35, 36] are often used to compactly and symbolically represent Boolean functions and perform Boolean operations. Abstraction is a general technique that groups several similar states into a single state by allowing additional behavior. For example, in Figure 1.1, the irregularly shaped light gray polygon over the variables x and y can be much more efficiently represented by encapsulating it in the dark gray polygon. This sometimes introduces false negatives, i.e., errors that cannot really occur. Therefore, it is necessary to have methods that determine if an error is false and to adjust the abstraction as necessary to prevent the false error from occurring allowing for verification to proceed. This process is known as *abstraction refinement*.

1.3 Hybrid System Verification

Hybrid systems are systems that contain both continuous and discrete behavior. AMS circuits fall into this category where the discrete behavior corresponds to the digital components of the circuit and the continuous behavior corresponds to the analog portions of the circuit. Significant research has been done in the modeling and verification of

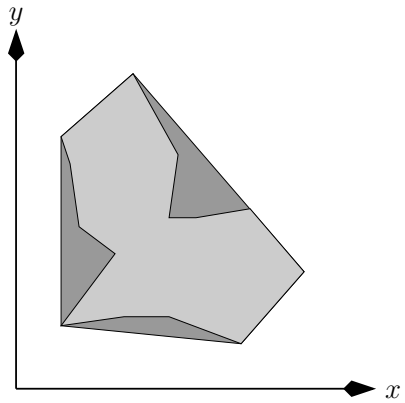


Figure 1.1. Intuition behind abstraction.

general hybrid systems. Much of the work described in this dissertation is based on previous work done in the hybrid system domain. A large array of tools has been developed for exploring the state space of hybrid systems; however, they differ widely in input models, capabilities, and methods used. Table 1.1 provides a listing of several of these tools [67]. Some of the tools use an exact reachability method while others use symbolic methods. Additionally, the tools can be differentiated by their conservativeness or exactness. Conservative tools find additional behavior that actually do not occur in the hybrid system, whereas exact tools perform state space explorations without introducing additional behavior.

The tool d/dt [31, 13, 14] relies on a method called *face-lifting* to create over approximate collections of *orthogonal polyhedra* representing the linear dynamics of the system. An efficient data structure is used for representing and operating on the polyhedra. Reachability is performed by iteratively performing face-lifting on the current set of states for some time step ΔT. The tool verifies that the set of reachable states does not intersect with a set of user specified bad states. Additionally, this tool supports systems with nondeterminism in the initial conditions and/or dynamics.

Another tool that can analyze hybrid systems is RED [70]. This tool was originally designed to analyze timed automata using *clock-restrictions diagrams* (CRDs) but has

Table 1.1. Hybrid analysis tool features.

Tool	Model Type	Data Structures	Precision	Property Specification
d/dt	LDHA	Polyhedra	Conservative	Safety
RED	LHA	HRDs	Exact	Safety
CheckMate	SimuLink/LHA	Polyhedra	Conservative	ACTL
TReX	CA, TA	SREs, CPDBMs	Exact	Safety
ATACS-DBM	LHPNs	DBMs	Conservative	ACTL
HyTech	LHA	Polyhedra	Exact	Safety
PHAVer	LHA	Polyhedra	Exact/Cons.	Safety
TMV	TA	BDDs	Exact	TCTL/$\mathbf{T}\mu$
ATACS-BDD	LHPNs	BDDs	Conservative	TCTL/$\mathbf{T}\mu$
ATACS-SMT	LHPNs	SMT	Exact	Safety

LHA = linear hybrid automata, LDHA = linear dynamical hybrid automata, CA = counter automata, TA = timed automata, LHPN = labeled hybrid Petri net, DBM = difference bound matrix, BDD = binary decision diagram, HRD = hybrid restriction diagram, SRE = simple regular expression, CPDBM = constrained parametric DBM

now been extended to analyze *hybrid automata* (HA) using an algorithm similar to other tools. This tool mainly contributes a new BDD-like data structure called *Hybrid Restriction Diagrams* (HRDs) to represent sets of convex polyhedra. HRDs are different from BDDs because the nodes contain LH-expressions and the arcs are labeled with LH-upperbounds. LH-expressions are of the form $\sum_i a_i x_i$ and LH-upperbounds are of the form $(\sim, c)$ where $\sim \in \{\leq, <\}$ and c is a rational number or ∞. A mapping from LH-expressions to LH-upperbounds defines a convex polyhedron.

CheckMate [25, 26, 66] is another hybrid analysis tool dealing specifically with *threshold event driven hybrid systems* (TEDHSs), a class of linear hybrid systems. A model is constructed using MATLAB Simulink and then CheckMate constructs an equivalent polyhedral invariant hybrid automaton that is analyzed and verification properties are specified using ACTL.

TReX [2, 9, 10] is an extensible symbolic analysis tool that operates on any symbolic data structure that has a symbolic successor/predecessor function and extrapolation procedure. Currently, the tool includes *simple regular expressions* (SREs) for modeling unbounded lossy FIFO-channels and constrained *Parametric Difference Bound Matrices* (PDBMs) for representing counter/clock automata.

ATACS-DBM [54] is an analysis tool developed specifically for the *labeled hybrid Petri net* (LHPN) modeling method. This tool is based on work in [19] and uses *difference-bound matrices* (DBMs) containing integer values to represent the continuous portion of the state space and relationships among clocks. In order to handle continuous behavior, a new method of warping DBMs based on rates of continuous variables is applied. However, this warping method and the use of integers introduce approximation into the algorithm. Additionally, it has the limitation of supporting only single rates on continuous variables.

The theoretical foundation for the analysis method that is described in this thesis is the work of Henzinger et al. [50] for timed automata. This work introduced *timed μ-calculus* ($\mathbf{T}\mu$) for specifying properties of real-time systems, represented states using *separation logic* (SL) which are Boolean combinations of Boolean variables and predicates, and performed a symbolic analysis by quantifying over SL and computing fix points using state predicates. These methods were later extended for use with *linear hybrid automata* (LHA) in the tool HyTech [51, 7].

The current state of the art tool for analyzing linear hybrid automata is PHAVer [40] which uses similar methods as those used by HyTech. PHAVer allows for unlimited

precision values and relies on the Parma Polyhedra Library which allows for exact computations with nonconvex polyhedra for its state space representation [15]. This tool also introduces the use of novel abstraction and approximation methods such as limiting the number of bits in the arithmetic representation and limiting the number of constraints.

The work of Henzinger is expanded upon by Seshia et al. [64, 65] in the development of the tool TMV. TMV relies on BDDs to represent the state space when verifying timed automata. As the analysis proceeds, predicates are created on the fly and mapped to BDD variables resulting in a Boolean encoding of SL formulas. Thus, the problem of quantifying over separation logic formulas with real variables has been reduced to quantifying over Boolean variables. When compared to other tools for analyzing timed automata, TMV performs quite well because of the use of BDDs. When analyzing larger systems, however, the creation of a large number of BDD variables requires significant amounts of memory.

ATACS-BDD and ATACS-SMT are the tools described in this dissertation. The methods used by ATACS-BDD and ATACS-SMT extend existing methods by supporting continuous variables that can change at any rate within a range in order to allow for the symbolic model checking of AMS circuits.

1.4 Analog Circuit Verification

To date, there has been relatively little research in the formal verification of AMS circuits. Perhaps the first work in this area is from Kurshan and McMillan in which analog circuits are represented as finite state models [53]. Hartong et al. verify analog circuits by dividing the continuous state space into regions that are represented in a Boolean manner [45]. This allows them to perform verification using standard Boolean-based approaches though with some loss of accuracy.

Tools for verifying hybrid systems have also been adapted to verify AMS circuits. Gupta et al. utilize their verification tool, CheckMate, to verify analog circuits such as a tunnel diode oscillator and a delta-sigma modulator [44]. In [30], Dang et al. use their tool, d/dt, to verify a biquad low-pass filter. These last two methods are very accurate but also very computationally complex. In [41], Frehse et al. use PHAVer to verify analog oscillator circuits. These approaches, however, require a user to describe an AMS circuit using a *hybrid automaton* which is unfamiliar to most AMS circuit designers. In [55], Little et al. adapt use the ATACS-DBM tool which uses a zone-based algorithm, for

the verification of AMS circuits. This method, however, supports only constant rates of change for the continuous variables and conservatively abstracts the continuous state space.

This dissertation describes two new model checking algorithms for the verification of AMS circuits. The model checkers have been developed to operate on the LHPN model which is then converted into a symbolic representation referred to as the *symbolic model* for analysis. The first model checker, ATACS-BDD, uses Boolean variables to represent the state space symbolically and performs a conservative state space exploration. The second model checker, ATACS-SMT, maps the symbolic model into a *Satisfiability Modulo Theories* (SMT) checker to perform a bounded state space exploration.

1.5 Contributions

Boolean methods can be used for the efficient formal verification of AMS circuits by mapping inequalities over real variables to Boolean variables and representing the state space as conjunctions and disjunctions of those Boolean variables.

The research described in this dissertation results in new methods and tools necessary for the formal verification of AMS circuits. Specifically, there are five main contributions: a hybrid modeling method for AMS circuits, generation of a symbolic model suitable for analysis, a Boolean based analysis method and implementation that uses BDDS, an SMT based analysis method, and demonstration of these methods on a several case studies.

The first contribution is the development of a hybrid system modeling method that is specially suited for modeling of AMS circuits. By developing a specialized modeling method for AMS circuits, our goal is to provide designers with an automated method of modeling based on their traditional design methodologies. Additionally, by customizing the model to the application, we can ensure that the model is expressive enough to analyze interesting properties and not too expressive to reduce the size and complexity of the systems that are analyzable. Specifically, the syntax and semantics for the LHPN model are formally introduced and methods for generating LHPNs are described. Specifically, approaches for generating LHPNs from VHDL-AMS and LHA are presented. Additionally, methods for approximating differential equations using LHPNs are described.

The second contribution is the method for generating a *symbolic model* from LHPNs. This step is required so that the two model checking algorithms described in this dissertation can be applied to LHPN models.

The third contribution is the development of a BDD based model checker. The model checking algorithm described performs a state space exploration of the provided models and determines if given properties are satisfied in every state. The algorithm maps inequalities over real variables to BDD variables and then uses BDD operations in combination with constraint generation to perform the state space exploration. The presented algorithm is novel because of its capability to allow for real variables to change at ranges of rates while still relying primarily on BDD operations. However, the necessity to frequently create constraints to maintain exactness results in poor performance. Therefore, significant effort was devoted to applying constraints in an efficient manner, resulting in an approximate algorithm with improved performanced. Additionally, more efficient methods were developed to to reduce the numbers of BDD variables that are created and thus the potential maximum BDD sizes.

The fourth contribution is the development of an SMT based bounded model checker. The model checking algorithm described utilizes elements of the BDD based model checker to construct SMT assertion statements which are then used to perform analysis over a specified set of iterations.

The fifth contribution is the development of AMS benchmarks and the initial application of the LHPN modeling and model checking methods to these benchmarks. The application of these methods demonstrates the usefulness and necessity of these methods.

1.6 Overview

This dissertation is divided into six chapters with each chapter building upon the preceding chapters. A chapter is devoted to each central contribution with a final concluding chapter. Figure 1.2 presents a flowchart of the steps in the AMS verification process described in this dissertation.

Chapter 2 describes the approach for modeling AMS systems. Background references related to modeling methods are provided. The remainder of Chapter 2 is devoted to the modeling portion of the verification flow in Figure 1.2. In this portion of the flow, the designer can specify the circuit in a number of ways including VHDL-AMS. By allowing designers to specify the model in a language that is familiar to them, it is hoped that they are encouraged to accept formal verification methodologies. The VHDL-AMS description is automatically compiled into a LHPN which includes Boolean signals to represent digital circuitry and continuous variables to model voltages and currents in the analog circuitry.

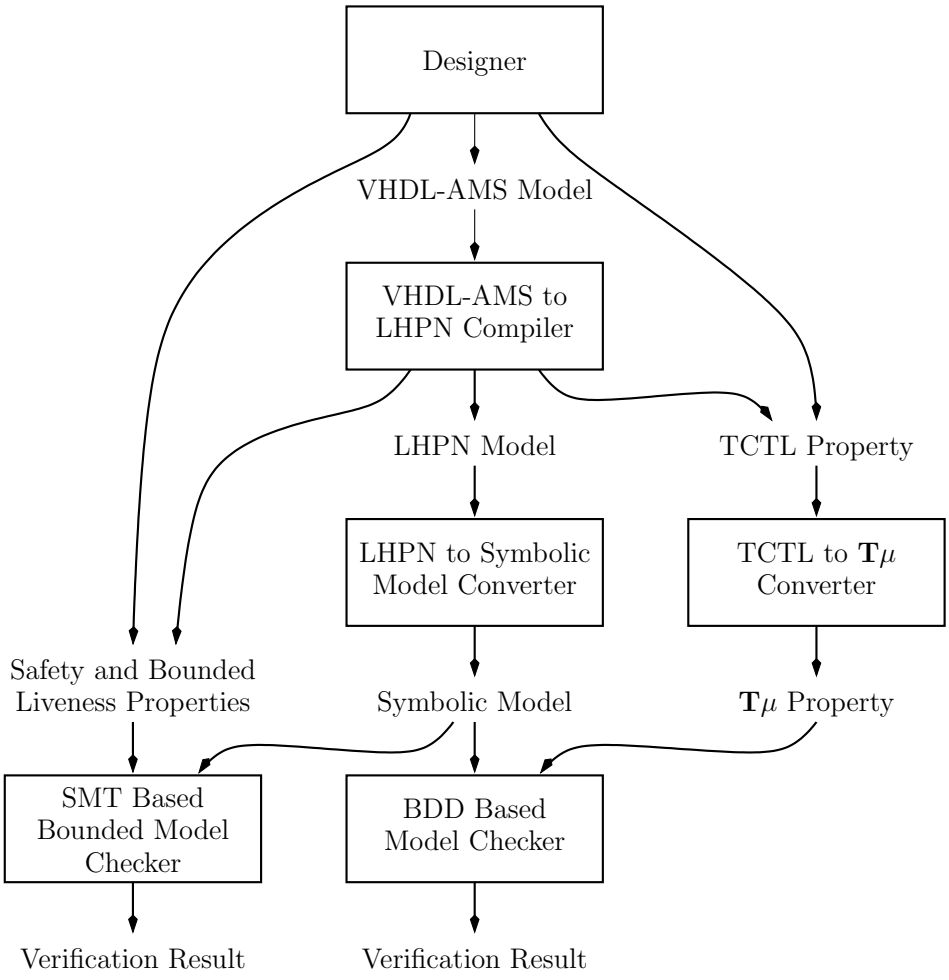


Figure 1.2. AMS circuit verification tool flow.

The LHPN model provides a formalism for reasoning about the system being analyzed.

Chapter 3 focuses on the symbolic verification algorithm. After providing background material related to the verification of hybrid systems, a detailed description of a new symbolic verification algorithm is provided. This chapter focuses on the property specification and LHPN to symbolic model conversion portions of the verification flow illustrated in Figure 1.2. System properties are specified as temporal logic formulas using *timed CTL* (TCTL). TCTL can be automatically generated from **assert** statements in VHDL-AMS or more complicated properties can be specified by the designer. In preparation for the application of the Boolean model checking method, an LHPN is converted to a Boolean symbolic model and a TCTL property is converted into a *timed* μ ($\mathbf{T}\mu$) property.

Chapter 4 explains the method for representing the state space in a Boolean manner.

In [65], Seshia and Bryant describe a symbolic model checking procedure for real-time systems based on the one described in [50]. Their method maps separation predicates to Boolean variables so that analysis can be performed using BDD operations. Since this work is only for real-time systems, all continuous variables can change only with a rate of one. Therefore, this dissertation extends this work to support continuous variables that can change at any rate within a range in order to allow for the symbolic modeling checking of AMS circuits with BDDs. The Boolean representation relies on a canonical representation of a restricted form of inequalities which is also described. Based on the Boolean representation, each component of the algorithm is described in detail.

Chapter 5 provides details and algorithms for implementing an LHPN model checker using *Satisfiability Modulo Theories* (SMT) tools. The SMT problem is a generalization of *Boolean Satisfiability* (SAT) where Boolean variables are replaced by predicates from various backbground theories including linear real and integer arithmetic. This support of additional theories lends itself well to the application of model checking hybrid systems. However, a necessity of creating state variables for each iteration of the state space exploration means that an SMT model checker is bounded.

Chapter 6 discusses the results of running several examples through the software developed from the implementation of these algorithms. Comparisons to other hybrid verification tools are also provided. Results demonstrating the necessity for abstraction methods are shown.

Chapter 7 summarizes the results of this work. The successes and limitations of this approach to verifying AMS circuits are discussed. Finally, several ideas for extending this research are presented as future work including a method of using the BDD based model checker in combination with the SMT based model checker to exploit each method's strengths for greater utility.

CHAPTER 2

LABELED HYBRID PETRI NETS

2.1 Related Work

Several methods have been developed for modeling hybrid systems. Some of these models resemble programming languages. One notable example is CHARON [6]. CHARON is a modular hierarchical language where *agents* are the outermost building blocks containing *modes* that describe the flow of control. Modes can contain additional modes demonstrating the hierarchical nature of CHARON. Additionally, agents and modes support concurrency and reuse. The goal of languages like CHARON is to be formal enough to support analysis while allowing for a higher-level reasoning about hybrid systems.

A dominant class of hybrid system models is *hybrid automata* [4, 5]. They combine automaton transitions for capturing discrete change, like digital circuit behavior with differential equations for capturing continuous change, like analog circuit behavior. Hybrid automata can model virtually any hybrid system, but this expressiveness makes analysis extremely expensive and complicated. Therefore, the analysis approach described in this dissertation is applicable only to a less expressive subset of hybrid automata known as *linear hybrid automata* (LHA) where the real variables change at rates within a bounded range and system properties are expressed in terms of linear constraints.

Another class of models is based on *Petri nets*. Petri nets consist of *places*, which can contain *tokens*. Tokens move through the net via *transition firings*, which consume tokens from the transition's incoming places and produce tokens in transition's outgoing places. Historically, Petri nets have been used to model discrete systems but various extensions have been proposed to develop hybrid Petri net models [33, 34]. One example is the Fluid Stochastic Petri Nets (FSPN) proposed by Tuffen et al. [69]. FSPNs represent continuous state with fluid places and are used with stochastic analysis methods. Another Petri net model is the Hybrid Net Condition/Event System (HNCES) model proposed by Chen et al. [24]. This model consists of discrete Petri nets and continuous Petri nets interacting with each other through condition and event signals.

LHA and other hybrid Petri net formalisms were considered; however, they each had certain limitations. Transition firings of LHA are forced via invariants—a construct not naturally present in AMS circuit specifications. In fact, the timing constraints of timed Petri nets were found to be more well suited for forcing transition firings. However, hybrid Petri net forms that contain places and transitions that are explicitly used for the flow of continuous quantities did not lend themselves to the modeling of AMS circuits either. The concept of having a resource that continuously flows from one place to another was not necessary. Rather the idea of a value that can change at varying rates was more important. Additionally, the connectivity between continuous places and transitions, and discrete places and transitions was found to be cumbersome. The remainder of this chapter focuses on the labeled hybrid Petri net (LHPN) model—a Petri net that is annotated with assignments, enabling conditions, and invariants over continuous variables and Boolean signals. The LHPN model is specially designed with analog and mixed-signal circuits in mind. The LHPN model augments discrete Petri nets with labels that can operate over continuous variables. This chapter describes the LHPN model and its semantics in detail. This chapter also describes methods for generating LHPNs from various other forms.

2.2 LHPN Definition

An LHPN is defined as a directed graph with labels on places and transitions. An LHPN is a tuple $N = \langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$:

- P : is a finite set of places;
- T : is a finite set of transitions;
- B : is a finite set of Boolean signals;
- V : is a finite set of continuous variables;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- L : is a tuple of labels defined below;
- $M_0 \subseteq P$ is the set of initially marked places;
- S_0 : is the set of initial Boolean signal values;
- Q_0 : is the set of initial ranges of values for each continuous variable and;

- R_0 : is the set of initial ranges of rates for each continuous variable.

The *preset* of a transition t, denoted as $\bullet t$, represents the set of places feeding t (i.e., $\{p \mid (p,t) \in F\}$). The *postset* of a transition t, denoted as $t\bullet$, represents the set of places that t feeds (i.e., $\{p \mid (t,p) \in F\}$). Similarly, the preset of a place p ($\bullet p$) is the set of transitions feeding the place, and the postset of a place p ($p\bullet$) is the set of transitions feeding that place.

A key component of LHPNs is labels. Some labels contain *hybrid separation logic* (HSL) formulas which are a Boolean combination of Boolean variables and separation predicates. HSL is an extension of *separation logic* [50, 64] (sometimes referred to as *difference logic*) that allows for non-unit slopes on the separation predicates. These formulas satisfy the following grammar:

$$\phi \quad ::= \quad \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid c_1 x_1 \geq c_2 x_2 + c_3$$

where b_i are Boolean variables, x_1 and x_2 are continuous variables, and c_1, c_2, and c_3 are rational constants in $\mathbb{Q}$. Note that any inequality between two real variables can be formed with $\geq$ inequalities and inverses of $\geq$ inequalities. The notation $boolPortion(\phi)$ is used to represent the Boolean portion of ϕ, i.e., ϕ with each of the inequalities existentially abstracted. Similarly, the notation $realPortion(\phi)$ represents the real portion of ϕ where each Boolean variable appearing in ϕ has been existentially abstracted. Given this separation logic, each transition $t \in T$ and place $p \in P$ is labeled using the functions defined in $L = \langle Inv, En, D, BA, VA, RA \rangle$:

- $Inv : P \rightarrow \phi$ labels each place $p \in P$ with an invariant;
- $En : T \rightarrow \phi$ labels each transition $t \in T$ with an enabling condition;
- $D : T \rightarrow |\mathbb{Q}| \times (|\mathbb{Q}| \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper bound $[d_{\text{l}}, d_{\text{u}}]$ on the delay for t to fire;
- $BA : T \rightarrow 2^{(B \times \{0,1\})}$ labels each transition $t \in T$ with Boolean assignments made when t fires;
- $VA : T \rightarrow 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a range of continuous variable assignments, consisting of a lower and upper bound $[a_{\text{l}}, a_{\text{u}}]$, that are made when t fires;

- $RA : T \rightarrow 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a range of rate assignments, consisting of a lower and upper bound $[r_l, r_u]$, that are made when t fires.

2.3 LHPN Example

To illustrate the LHPN model, the switched capacitor integrator shown in Figure 2.1 is used as a running example throughout this chapter. This circuit takes as input a 5 *kHz* square wave that varies from -1 V to 1 V and generates a triangle wave as output representing the integral of the input voltage. A simulation of this circuit under ideal conditions is shown in Figure 2.2. Discrete-time integrators typically utilize switched capacitor circuits to accumulate charge, which can cause gain errors in the integrator due to capacitor mismatch. Therefore, the output voltage in our model is allowed to have a slew rate anywhere between 18 to 22 $mV/\mu s$ to represent a ± 10 percent variance in circuit parameters. A random simulation allowing for the variance in the output slew rate is shown in Figure 2.3. The verification goal is to ensure that *Vout* never saturates (i.e., it is always between -2000 mV and 2000 mV). An experienced analog circuit designer may realize the potential of this circuit to fail. However, a very specific SPICE simulation is required to demonstrate this failure as shown in Figure 2.4 where the output voltage always increases at a faster rate than it decreases. Furthermore, it is highly unlikely that a simulation allowing for random uncertainty in the system variables would reveal the error [58]. Therefore, a formal verification approach is beneficial.

Figure 2.5 shows an LHPN model of the switched capacitor circuit in Figure 2.1. This example is used to intuitively describe LHPN semantics before they are formally

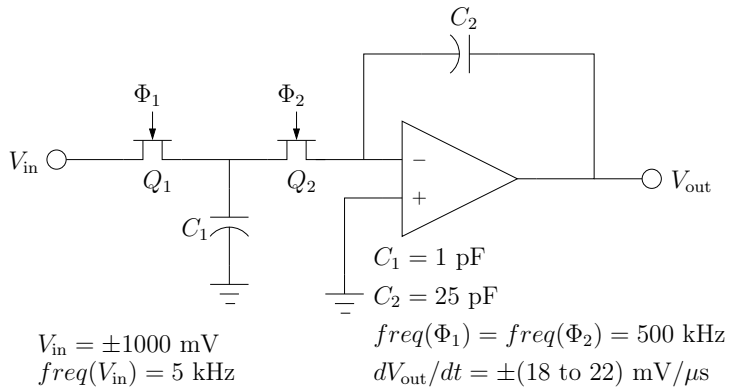


Figure 2.1. Circuit diagram of a switched capacitor integrator.

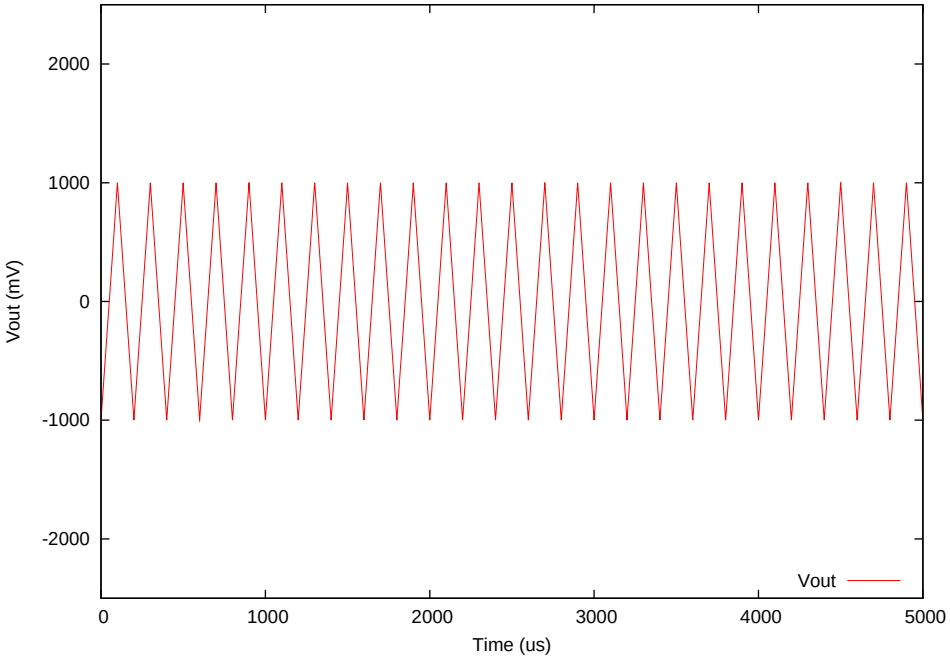


Figure 2.2. Basic simulation of integrator under ideal conditions.

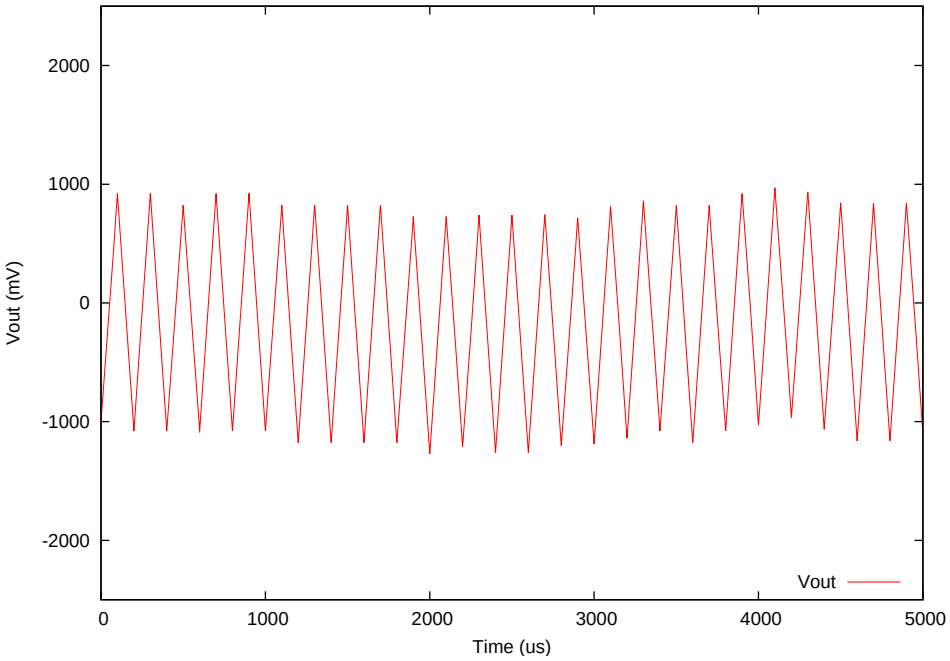


Figure 2.3. Random simulation of integrator with variance in circuit parameters.

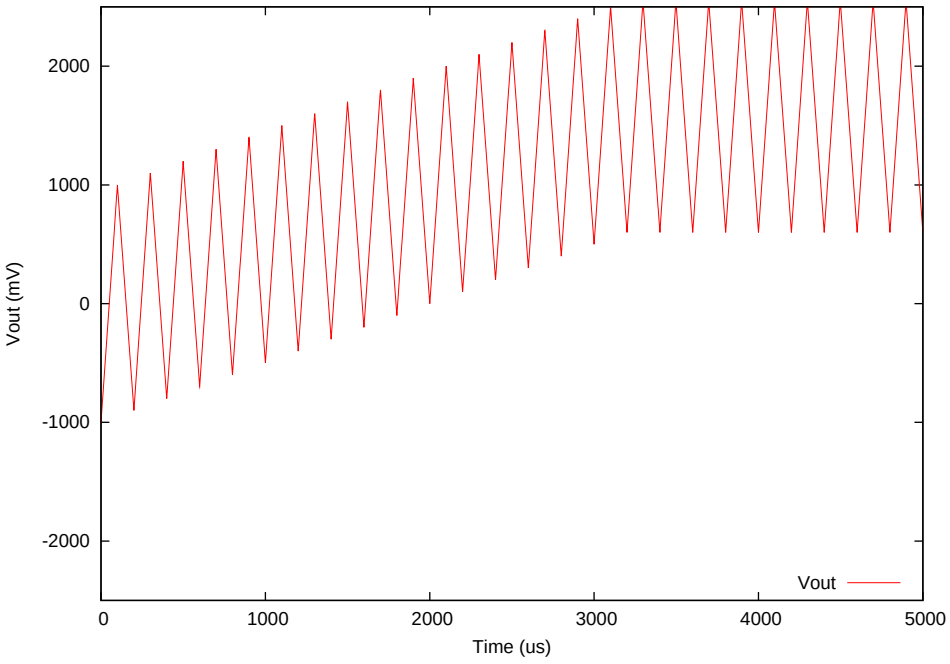
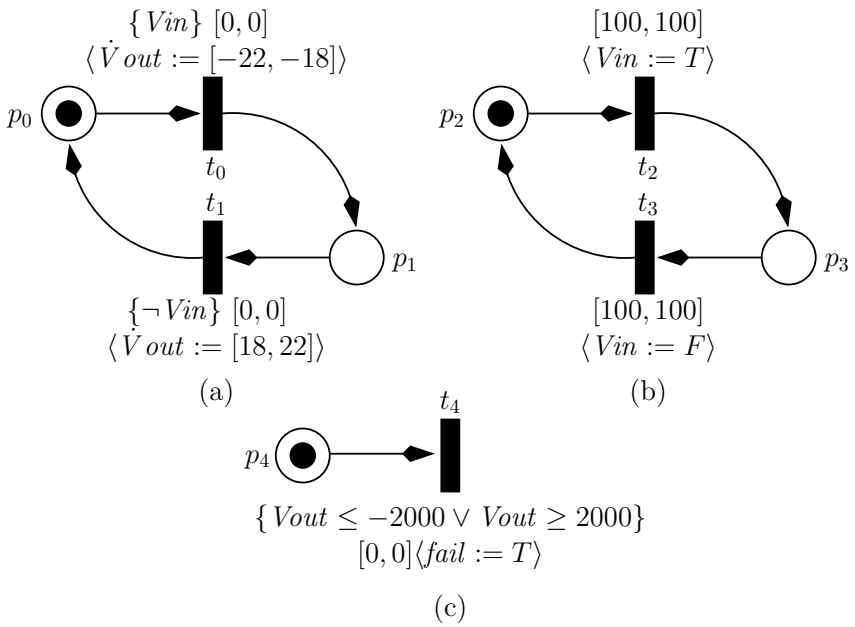


Figure 2.4. Worst case simulation of integrator with variance in circuit parameters.



$$Q_0 = \{\, Vout = -1000\} \quad R_0 = \{\, \dot{V}\, out = [18, 22]\} \quad S_0 = \{\neg Vin, \neg fail\}$$

Figure 2.5. LHPN model of the switched capacitor integrator.

described. Transitions in LHPNs are controlled by enabling conditions and timing constraints. When the enabling condition becomes satisfied, the clock on the transition begins and the transition fires sometime during which the clock is above its lower bound and below its upper bound. Upon firing, the discrete marking is updated by removing tokens from the preset places of the transition and placing tokens in the postset places of the transition. Additionally, assignments are made to continuous variables, rates of continuous variables, and Boolean variables. For the LHPN in Figure 2.5, the marking is initially $\{p_0, p_2, p_4\}$, the Boolean variables *Vin* and *fail* are **false**, the continuous variable *Vout* is -1000, and *Vout* is increasing at a rate of 18 to 22 mV/μs. After 100 μs, t_2 is required to fire resulting in p_2 becoming unmarked, p_3 becoming marked, and the assignment of **true** to *Vin*. This assignment causes the enabling and immediate firing of t_0 and thus the assignment of -22 to -18 mV/μs to the rate for *Vout*. After an additional 100 time units, transition t_3 fires, causing *Vin* to be assigned **false** which results in t_1 becoming enabled. Immediately, t_1 fires, and the process continues. If at any time, the output voltage *Vout* falls below -2000 mV or increases about 2000 mV, transition t_4 will fire causing *fail* to be set to **true**. The assignment of **true** to *fail* indicates a failure.

2.4 LHPN Semantics

The formal semantics of LHPNs is considerably complex, specifically with regard to range assignments to values and rates of continuous variables. Consider an LHPN example with a single continuous variable, y, that is assigned the value range of 2 to 3 and a rate range of 1 to 2 at time 0. Figure 2.6a shows all the possible values that y could have over time. Figure 2.6b shows possible traces through this range assuming that at the time of assignment, a random value and rate within the specified ranges are selected for y. This interpretation of the semantics, however, does not allow for traces where the rate of y can change at any time as in Figure 2.6c. Similarly, the trace shown in Figure 2.6d, where the piece-wise representation has such a fine granularity that it looks like a continuous curve, would not be allowed.

Furthermore, consider the case where y is assigned an initial value range of 2 to 3 and a constant rate of one at time 0. Figure 2.7a shows all possible values of y in this instance and Figure 2.7b shows a possible trace. However, the traces in Figure 2.7c and Figure 2.7d could also be considered reasonable traces if the initial value of y could change within its specified assignment range at any time. In Figure 2.7d, y changes by limited amounts

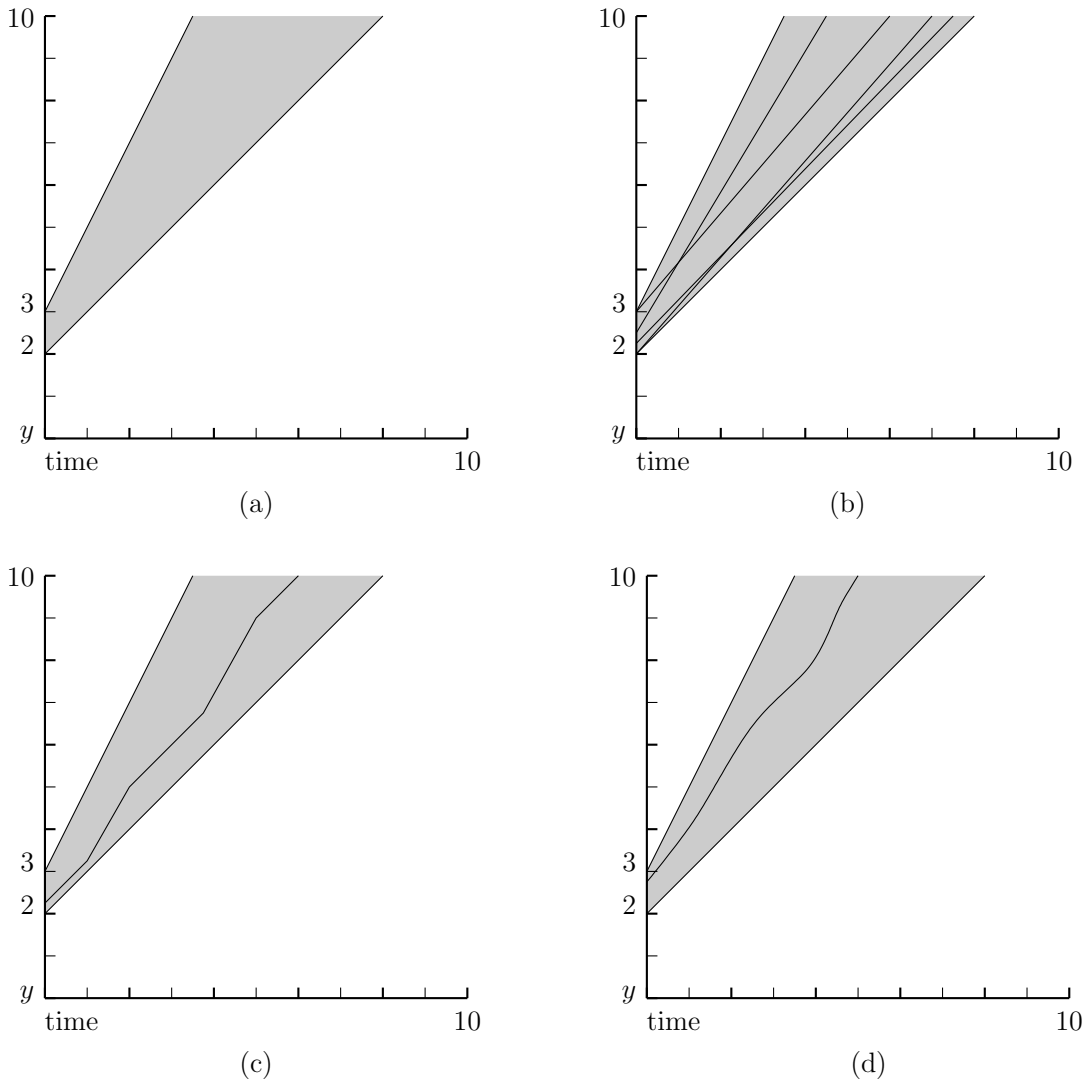


Figure 2.6. Range assignment semantics of LHPNs. The shaded region encompasses all the possible values of y and the lines represent possible traces through the region under different semantic conditions.

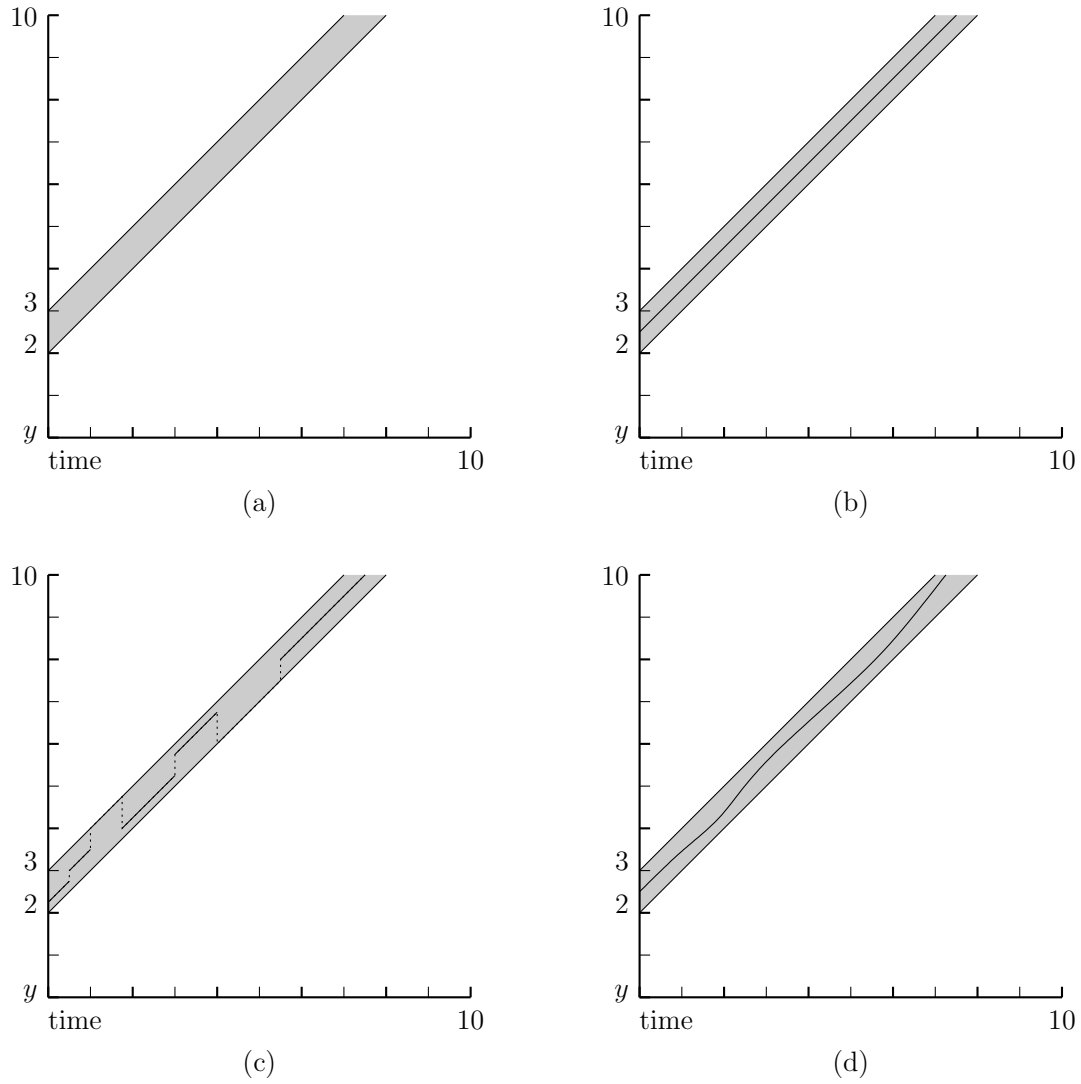


Figure 2.7. Range assignment semantics of LHPNs where the rate is a single value. The shaded region encompasses all possible values of y and the lines represent possible traces through the region under different semantic conditions.

very frequently resulting in a curve that looks, but is not, continuous. This models the idea that as y evolves, there is uncertainty about its value. With these considerations in mind, a semantics for LHPNs allowing all of these described behaviors, follows.

The state of an LHPN is defined using a 6-tuple of the form $\psi = \langle M, S, Q, R, C, QL, QR, RR \rangle$ where:

- $M \subseteq P$ is the set of marked places;
- $S : B \rightarrow \{0, 1\}$ is the value of the Boolean signals;
- $Q : V \rightarrow \mathbb{Q}$ is the value of the continuous variables;
- $R : V \rightarrow \mathbb{Q}$ is the rate of each continuous variable;
- $C : T \rightarrow \mathbb{Q}$ is the value of the transition clocks;
- $QL : V \rightarrow \mathbb{Q}$ is the last value that each variable is set to via a transition or adjustment;
- $QR : V \rightarrow \mathbb{Q} \times \mathbb{Q}$ is the range that is used in the most recent assignment to a particular value;
- $RR : V \rightarrow \mathbb{Q} \times \mathbb{Q}$ is the current range of acceptable rates for each continuous variable;

The first five elements of the state (M, S, Q, R, and C) define the essential state information and the last three elements of the state (QL, QR, and RR) provide additional information that is necessary for calculating next states. The current state of an LHPN can change via a transition firing, time advancement, or by adjustments to a variable's value and/or rate.

From a state ψ, a new state $\psi' = \langle M', S', Q', R', C', QL', QR', RR' \rangle$ can be reached by firing an *enabled* transition t_i (denoted as $\psi \xrightarrow{t_i} \psi'$). A transition t_i is enabled when all of the places in its preset are marked (i.e., $\bullet t_i \subseteq M$) and the enabling condition on t_i evaluates to true (i.e., $Eval(En(t_i), S, Q)$ where the function $Eval : \phi \times S \times Q \rightarrow \{0, 1\}$ evaluates an HSL formula for given values of the Boolean signals and continuous variables). The set of enabled transitions is calculated by the function $\mathcal{E}(M, S, Q)$ given a state marking, set of signal values, and set of variable values as follows:

$$\mathcal{E}(M, S, Q) = \{t_i \in T | \bullet t_i \subseteq M \wedge Eval(En(t_i), S, Q)\}$$

When firing a transition t, the next state is calculated as follows:

- $M' = (M - {\bullet t}) \cup t{\bullet}$
- $\forall b_i \in B.S'(b_i) = \begin{cases} s & \text{if } \exists (b_i, s) \in BA(t) \\ S(b_i) & \text{otherwise} \end{cases}$
- $\forall v_i \in V.Q'(v_i) = \begin{cases} a_l & \text{if } \exists (v_i, a_l, a_u) \in VA(t) \\ Q(v_i) & \text{otherwise} \end{cases}$
- $\forall v_i \in V.R'(v_i) = \begin{cases} r_l & \text{if } \exists (v_i, r_l, r_u) \in RA(t) \\ R(v_i) & \text{otherwise} \end{cases}$
- $\forall t_i \in T.C'(t_i) = \begin{cases} 0 & \text{if } t_i \in \mathcal{E}(M', S', Q') \wedge t_i \notin \mathcal{E}(M, S, Q) \\ C(t_i) & \text{otherwise} \end{cases}$
- $\forall v_i \in V.QL'(v_i) = \begin{cases} a_l & \text{if } \exists (v_i, a_l, a_u) \in VA(t) \\ QL(v_i) & \text{otherwise} \end{cases}$
- $\forall v_i \in V.QR'(v_i) = \begin{cases} (a_l, a_u) & \text{if } \exists (v_i, a_l, a_u) \in VA(t) \\ QR(v_i) & \text{otherwise} \end{cases}$
- $\forall v_i \in V.RR'(v_i) = \begin{cases} (r_l, r_u) & \text{if } \exists (v_i, r_l, r_u) \in RA(t) \\ RR(v_i) & \text{otherwise} \end{cases}$

In other words, the marking is updated by removing the places in the preset of t_i and adding the places in the postset of t_i. Additionally, the Boolean assignments, the continuous assignments, and the rate assignments associated with transition t_i are executed. In the cases of continuous assignments and rate assignments, the lowest value in the range of assignments is selected initially. Next, the clocks associated with any newly enabled transitions are set to zero. When a continuous variable assignment is performed, its corresponding last assignment value is set to the same value to keep track of the most recent assignment to that variable. This value is then used during state adjustments to calculate the amount by which the variable has changed since its previous assignment. Finally, the ranges of the continuous variable and rate assignments are stored in the state so that continuous values and rates can periodically be adjusted for each variable.

Not all enabled transitions can fire. In order to fire, the transition must be *admissible*. When a transition becomes enabled, its clock begins incrementing from zero. It is admissible to fire a transition t_i at any time after the transition's clock satisfies its lower delay bound and before it exceeds its upper delay bound (i.e., $d_l(t_i) \leq C(t_i) \leq d_u(t_i)$) as long as it remains continuously enabled. Furthermore, the state that would result from firing t_i must not violate any place's invariant (i.e., $\forall p_i \in M'.Eval(Inv(p_i), S', Q')$).

The function $\mathcal{A}(\psi, t_i)$ determines if transition t_i is admissible given a current state. It is defined as follows:

$$\begin{aligned}\mathcal{A}(\psi, t_i) \quad = \quad & \{t_i \in \mathcal{E}(M,S,Q) \wedge d_l(t_i) \leq C(t_i) \leq d_u(t_i) \wedge \\ & \forall p_i \in M'.Eval(Inv(p_i), S', Q') \text{ where } \psi \xrightarrow{t_i} \psi'\}\end{aligned}$$

From a state ψ, a new state ψ' can be reached by elapsing τ time units (denoted as $\psi \xrightarrow{\tau} \psi'$). The resulting state is calculated as follows:

- $\forall v_i \in V.Q'(v_i) = Q(v_i) + R(v_i) \cdot \tau$
- $\forall t_i \in T.C'(t_i) = \begin{cases} 0 & \text{if } t_i \in \mathcal{E}(M,S,Q') \wedge t_i \notin \mathcal{E}(M,S,Q) \\ C(t_i) + \tau & \text{otherwise} \end{cases}$

In other words, when calculating the new state, values of each continuous variable are updated based on the current rate and the amount of time that has elapsed. Additionally, active clocks are incremented by τ time units and clocks on newly enabled transitions are set to zero. Transitions can become enabled via time elapse because enabling conditions can become satisfied as variables change value. All other components of the state remain the same.

Time can potentially advance by any value τ which is less than $\tau_{\max}(\psi)$. The value of $\tau_{\max}(\psi)$ is the largest amount of time that may pass before a transition is forced to fire (i.e., the clock associated with it exceeds its upper bound) or an inequality changes its Boolean value based on the current values and rates of the variables. The maximum possible time advancement, $\tau_{\max}(\psi)$, is calculated as:

$$\tau_{\max}(\psi) = min \begin{cases} C(t_i) - d_u(t_i) & \forall t_i \in \mathcal{E}(M,S,Q) \\ \tau'' & \begin{array}{l} \mathcal{E}(M'',S'',Q'') \neq \mathcal{E}(M,S,Q) \wedge \\ \forall \tau' < \tau''.\mathcal{E}(M',S',Q') = \mathcal{E}(M,S,Q) \\ \text{where } \psi \xrightarrow{\tau'} \psi' \text{ and } \psi \xrightarrow{\tau''} \psi'' \end{array} \end{cases}$$

It is admissible to elapse time by $\tau \leq \tau_{\max}(\psi)$ time units as long as the invariants on the marked places remain satisfied at all times less than τ. Formally, τ is admissible if the following holds in state ψ:

$$\mathcal{A}(\psi, \tau) = \tau \leq \tau_{\max}(\psi) \wedge \forall \tau' \leq \tau. \bigwedge_{p_i \in M} Eval(Inv(p_i), S', Q')) \text{ where } \psi \xrightarrow{\tau'} \psi'.$$

At random times, a new state ψ' can be reached from a state ψ, by adjusting a continuous variable's value (denoted as $\psi \xrightarrow{Q(v_i) \leftarrow q} \psi'$) or by adjusting a continuous variable's rate (denoted as $\psi \xrightarrow{R(v_i) \leftarrow r} \psi'$). When an adjustment to a variable's value occurs, the new state is calculated as follows where $v_i \in V$ and $q \in QR(v_i)$:

- $Q'(v_i) = (Q(v_i) - QL(v_i)) + q$
- $QL'(v_i) = q$

The adjustment is applied by assigning a new value, q, from the most recent range of assignments to the variable v_i and adding it to the amount by which v_i had already changed since the last assignment (i.e., $Q(v_i) - QL(v_i)$). Additionally, the variable's last assignment value is updated to the new value. All other components of the state remain the same. It is admissible to adjust a variable v_i's value using q if the following holds:

$$\mathcal{A}(\psi, v_i, q) = \forall q' \in ([QL(v_i), q] \cup [q, QL(v_i)]).(\mathcal{E}(M, S, Q) = \mathcal{E}(M', S', Q') \wedge \forall p_i \in M'.Eval(Inv(p_i), S', Q') \text{ where } \psi \overset{Q(v_i) \leftarrow q'}{\rightarrow} \psi'$$

In other words, in order to perform a value adjustment, the set of enabled transitions must remain the same for all values between the variable's previous value and the variable's new value, and the invariants for all enabled transitions must remain satisfied for all values between the variable's previous value and the variable's new value.

An adjustment to a variable's rate, which is always admissible, results in a new state calculated as follows where $v_i \in V$ and $r \in RR(v_i)$:

- $R'(v_i) = r$

In this case, the new rate assignment is simply updated in the state. Adjustments to a variable's value or rate can occur at any time on any variable.

2.5 LHPN Execution

The execution of an LHPN is a finite or infinite sequence $\psi_0 \rightarrow \psi_1 \rightarrow \psi_2 \rightarrow ...$ of states such that:

1. The initial state, ψ_0 is $\langle M_0, S_0, lowers(Q_0), lowers(R_0), \mathbf{C} := 0, lowers(Q_0), Q_0, R_0 \rangle$ where *lowers* forms a set containing the lower bound values for each variable in the range sets;

2. The next state is reached by a transition firing ($\overset{t_i}{\rightarrow}$), a time progression ($\overset{\tau}{\rightarrow}$), a continuous value adjustment ($\overset{Q(v_i) \leftarrow q}{\rightarrow}$), or a continuous rate adjustment ($\overset{R(v_i) \leftarrow r}{\rightarrow}$);

3. If the next state is reached via a transition firing ($\psi_i \overset{t_i}{\rightarrow} \psi_{i+1}$), then the transition t_i must be admissible, i.e., $\mathcal{A}(\psi_i, t_i)$ must be satisfied;

4. If the next state is reached via time progression ($\psi_i \xrightarrow{\tau} \psi_{i+1}$), then the amount of time elapsed, τ, is admissible, i.e., $\mathcal{A}(\psi_i, \tau)$;

5. If the next state is reached via a continuous value adjustment ($\psi_i \overset{Q(v_i)\leftarrow q}{\rightarrow} \psi_{i+1}$), then it must be admissible as determined by $\mathcal{A}(\psi, v_i, q)$ where $v_i \in V$ and $q \in QR(v_i)$;

6. If the next state is reached via a continuous rate adjustment ($\psi_i \overset{R(v_i)\leftarrow r}{\rightarrow} \psi_{i+1}$), then $v_i \in V$ and $r \in RR(v_i)$.

The language accepted by an LHPN, N, is the collection of all possible executions of the LHPN and it is denoted as $\mathcal{L}(N)$.

2.6 LHPN Simulation

An algorithm for simulating LHPNs is shown in Figure 2.8. The algorithm operates by constructing the initial state from the model using the function *lowers* which returns a set containing the lower bound values for each variable in the range sets. This algorithm requires that a minimum time step δ be specified by which time is elapsed at each iteration, if possible. This simulation approach is necessary because the calculation of $\tau_{\max}(\psi)$ requires the determination that at all times below $\tau_{\max}(\psi)$ that the set of enabled transitions remain unchanged. This is difficult to ensure given the infinite number of possible times below $\tau_{\max}(\psi)$. The algorithm operates by firing any events that are queued to occur at the current time. Alternatively, if no events are ready to occur, a time step of δ is elapsed, if possible. To determine if it is possible to elapse time by δ time units, a time progression is applied where τ equals δ, resulting in a new state ψ'. Next, the invariants are checked based on ψ' to ensure that application of the time step did not violate them. If none of the invariants are violated, the time step is allowed, otherwise it is rolled back. After either applying a transition or moving time forward by δ time units, a continuous value adjustment is performed by randomly selecting a new value and testing if the set of enabled transitions is impacted. If not, the continuous value adjustment is applied. Additionally, continuous rate adjustments are performed by randomly selecting new values from the rate ranges. Next, the set of currently enabled transitions is calculated based on the current state. Any events that are no longer enabled are removed from the event queue and newly enabled transitions are scheduled for firing by randomly selecting a time within the transition's timing bounds. Finally, this process repeats itself forever.

```
LHPNSim (⟨P, T, B, V, F, L, M_0, S_0, Q_0, R_0⟩, δ)
   ψ = ⟨M_0, S_0, lowers(Q_0), lowers(R_0), C := 0, lowers(Q_0), Q_0, R_0⟩
   eventQueue = ∅ // Pairs of times and transitions ordered by times.
   time = 0
   while true
      // Perform the next scheduled discrete transition if possible
      if eventQueue.front().time() == time
         t_i = eventQueue.pop().transition()
         if A(ψ, t_i) then
            ψ --t_i--> ψ'
            ψ = ψ'
         end if
      // If no transitions were fired, elapse time by δ if possible.
      else
         ψ --δ--> ψ'
         if ⋀_{p_i ∈ M} Eval(Inv(p_i), S', Q') then
            ψ = ψ'
            time = time + δ
         end if
      end if
      // Adjust value and rates
      for each v_i ∈ Q
         select q ∈ QR(v_i)
         ψ --Q(v_i)←q--> ψ'
         if E(M, S, Q) == E(M', S', Q') ∧ ⋀_{p_i ∈ M} Eval(Inv(p_i), S', Q') then
            ψ = ψ'
         end if
         select r ∈ RR(v_i)
         ψ --R(v_i)←r--> ψ'
         ψ = ψ'
      end for
      // Update event queue.
      currentE = E(M, S, Q)
      // Remove events that are no longer enabled.
      for each e ∈ eventQueue
         if e.transition() ∉ currentE then
            eventQueue.remove(e)
         end if
      end for
      // Insert newly enabled events and pick a random firing time.
      for each t_i ∈ currentE
         if ⟨*, t_i⟩ ∉ eventQueue then
            // Calculate next time and round to nearest δ
            rtime = time + (random(lower(t_i), upper(t_i))/δ) * δ
            eventQueue.insert(⟨rtime, t_i⟩)
         end if
      end for
   end while
end
```

Figure 2.8. Algorithm for simulating LHPNs that is approximate.

Note that this simulation algorithm is approximate in that the set of executions that this simulation approach allows is not equivalent to $\mathcal{L}(N)$. This is due to the fact that events are scheduled to occur on delta time step boundaries which may in fact be slightly before or slightly after when the event is supposed to occur. As a result, the simulator can potentially allow traces that are not in $\mathcal{L}(N)$ and potentially disallow traces that are in $\mathcal{L}(N)$.

2.7 Generating LHPNs from LHAs

Linear hybrid automata provide many of the same capabilities as LHPNs with the exception that concurrency is arguable easier to represent in LHPNs. An LHA model of the switched capacitor integrator from Figure 2.1 is shown in Figure 2.9 , minus the fail transition, and referred to throughout the formal definition of LHA. This definition is based on that in [7]. An LHA is defined as a tuple, $H = \langle X, O, E, inv, dif, act, syn \rangle$, such that:

X : is a finite ordered set of real-valued *data variables* $X = \{x_1, x_2, \ldots, x_n\}$. Note that $\dot{X} = \{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n\}$ is a finite ordered set of real-valued variables where $\dot{x}_i$ refers to the first derivative of x_i with respect to time. X_0 contains the initial values of the variables in X. In Figure 2.9, $X = \{V_{out}, clk\}$; $\dot{X} = \{\dot{V}_{out}, \dot{clk}\}$; and the initial values for V_{out} and clk are -1000 and 0, respectively.

O : is a finite set of vertices called *control locations*. Figure 2.9 has two control locations labeled **low** and **high**. The location **low** is initially active.

E : is a finite multiset of edges called *transitions* where each transition, t, is a directed edge from a source location v_i to a target location v_j. The set E in Figure 2.9

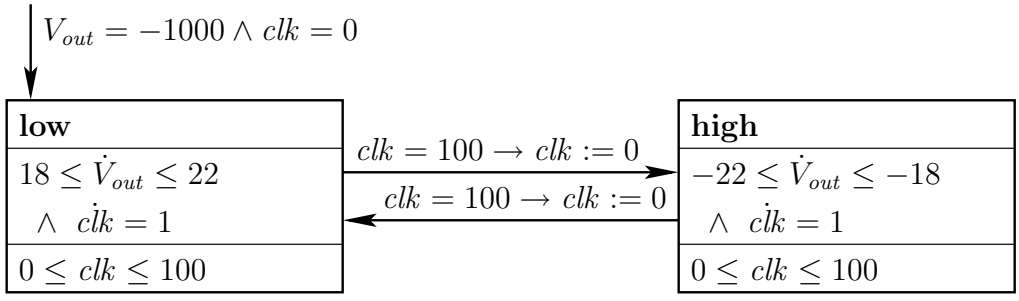


Figure 2.9. LHA model of the switched capacitor integrator circuit.

contains two members: $t_1 = (\mathbf{low}, \mathbf{high})$ and $t_2 = (\mathbf{high}, \mathbf{low})$. To simplify the presentation, it is assumed that there is at most a single edge between two locations (i.e., E is a set rather than a multiset). However, the methods described in this chapter can be readily extended to support multiple edges between two locations.

inv: is a labeling function that assigns an invariant condition to each control location. The invariant is an HSL formula over the data variables in the system. If the automaton is in location v, $inv(v)$ may force a transition to occur by preventing time from progressing beyond a point in which $inv(v)$ is true. An example of an invariant in Figure 2.9 is $0 \leq clk \leq 100$ in location **low**.

dif: is a labeling function that assigns a range of rates, $dif(v)$, to each variable in each control location v. For example, location **low** in Figure 2.9 has ranges of rates of $18 \leq \dot{V}_{out} \leq 22$ and $\dot{clk} = 1$.

act: is a labeling function that assigns an action, $act(t)$, to each transition $t \in E$. The action is a guarded command $act(t) = (guard(t) \rightarrow assign(t))$ where $guard(t)$ is an HSL formula without Booleans and $assign(t)$ is a set of data variable assignments of the form $x_k := [l_k, u_k]$ which assigns an inclusive range of rational values between l_k and u_k to x_k. When assigning a single value, the abbreviated form $x_k := a_k$ is used. In Figure 2.9, the action for the transition between **low** and **high** is $clk = 100 \rightarrow clk := 0$ where $clk = 100$ is the guard and $clk := 0$ is the assignment. Note that $clk = 100$ is represented in HSL as $clk \geq x_0 + 100 \wedge x_0 \geq clk - 100$.

syn: is a labeling function that assigns a set of *synchronization labels* to each transition $(v_i, v_j) \in E$. Synchronization labels are used for communication between automata in a parallel composition of automata. Figure 2.9 is a single automaton so synchronization labels are not used.

Formal semantics for LHA are given in [7]. Intuitively, transitions in LHA are controlled by a combination of guards and invariants. While in a location, the data variables change at their specified rate as long as the invariant is satisfied. If progress would violate the invariant, time progression is halted. In Figure 2.9, beginning in location **low** with V_{out} equal to $-1000\ mV$ and clk equal to 0, V_{out} increasing at a rate between 18 and 22 $mV/\mu s$ for 100 μs. When clk reaches 100, V_{out} is between 800 and 1200 mV. Once clk equals 100, the enabling condition on the transition between **low** and **high**

becomes enabled and the transition occurs before *clk* increases beyond 100 which would violate the invariant. While in location **high**, V_{out} decreases at a rate been 18 and 22 $mV/\mu s$ until *clk* equals 100. At this point V_{out} would be between -1600 and -600 mV, and a transition into the **low** state occurs and the process repeats.

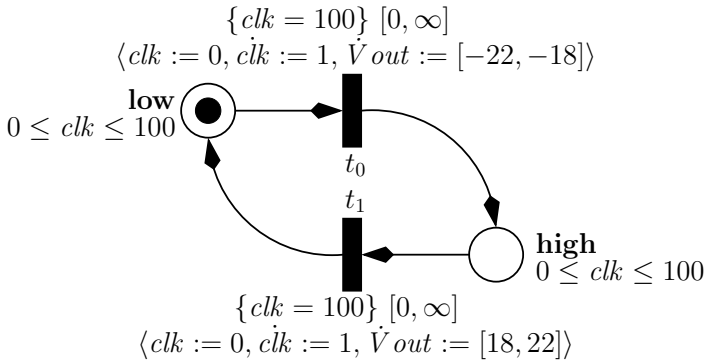
All languages that can be specified by LHAs can also be specified by LHPNs. This is shown by describing a straightforward translation from LHAs to LHPNs. The reverse is also believed to be true, but a proof is not shown here as it is not important for this research. Given this translation from LHA to LHPNs, it is useful to consider the properties that the resulting LHPNs have in comparison to the original LHA. Given the direct translation from LHA to LHPNs, it is asserted that the set of languages specified by an LHPN generated from an LHA is equivalent to the languages specified by the original LHA. In other words, consider an LHA H and the language that it specifies $\mathcal{L}(H)$, and an LHPN N that is converted from H and the language that it specifies $\mathcal{L}(N)$. The two languages are equivalent, i.e., $\mathcal{L}(H) \equiv \mathcal{L}(N)$. Therefore, the results of analyzing the LHPN N (i.e., whether or not it violates a given property) also apply to the original LHA H.

The algorithm for generating an LHPN from an LHA that does not contain synchronization labels is shown in Figure 2.10. The translation begins by creating places for each location in the LHA and assigning invariants to those places based on the invariants on the locations in the LHA. Continuous variables are created for each data variable in the LHA and since there are no Boolean signals in LHAs, the corresponding LHPN contains no Boolean signals. Next, for each edge in the LHA, a transition and arcs connecting the places to and from the transition are created. As the transitions are created, enabling conditions and variable assignments are extracted from the LHA's actions, delays are set to zero, Boolean assignments are set to empty sets, and rate assignments are extracted from the location that corresponds to the transition's outgoing place. The LHPN in Figure 2.11 shows the result of automatically translating the LHA in Figure 2.9 to an LHPN.

When an LHA contains synchronization labels, the translation becomes slightly more complicated. Two possible approaches exist for converting parallel compositions of automata ($\{H_0...H_n\}$) that use synchronization labels to LHPNs. The first approach is to merge the LHAs into a single automaton that does not contain synchronization labels and convert the resulting LHA into an LHPN. An algorithm for composing multiple automata

$\langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$ **LHAtoLHPN** $(\langle X, O, E, inv, dif, act, syn \rangle)$
assert $syn == \{\emptyset\}$
for each $v_i \in O$
$P = P \cup p_{v_i}$
$Inv(p_{v_i}) = inv(v_i)$
end for
$V = X$
$B = \{\emptyset\}$
for each $(v_i, v_j) \in E$
$T = T \cup t_{v_i,v_j}$
$F = F \cup (p_{v_i}, t_{v_i,v_j}) \cup (t_{v_i,v_j}, p_{v_j})$
$En(t_{v_i,v_j}) = guard(v_i, v_j)$
$D(t_{v_i,v_j}) = (0, \infty)$
$BA(t_{v_i,v_j}) = \{\emptyset\}$
$VA(t_{v_i,v_j}) = assign(v_i, v_j)$
$RA(t_{v_i,v_j}) = dif(v_j)$
end for
$M_0 = p_{v_0}$
$S_0 = \{\emptyset\}$
$Q_0 = X_0$
$R_0 = dif(v_0)$
return $\langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$
end

Figure 2.10. Algorithm for converting LHAs to LHPNs. This algorithm takes a tuple representing an LHA and returns a tuple representing an LHPN.



$$Q_0 = \{Vout = -1000\} \quad R_0 = \{\dot{clk} = 1, \dot{V}out = [18, 22]\} \quad S_0 = \{\}$$

Figure 2.11. LHPN of the integrator generated from the LHA integrator model.

into a single automaton is shown in [49].

An alternative approach, which works with the majority of LHA but perhaps not all, requires special LHPN segments to be inserted where synchronization labels exist. The LHPN segment shown in Figure 2.12 is inserted for each edge (v_i, v_j) in H_i that has a synchronization label l. If H_i contains no edges with synchronization label l, the Boolean variable l_i is set to true in the initial state. The LHPN segment operates by modeling a four-phase handshake. When the guard on each edge is satisfied, the Boolean signal l_i is set to true. If at any time the guard becomes unsatisfied, l_i is reset to false and the handshake is aborted. When all Boolean signals l_i become true, a transition occurs which results in the assignments on the edge being performed. Finally, the Boolean signals are reset to false for use in future transitions.

2.8 Generating LHPNs from VHDL-AMS

VHDL-AMS is a hardware description language that includes extensions specifically for describing analog and mixed-signal circuits. By providing a translation mechanism from VHDL-AMS to LHPNs, many of the hurdles associated with verification can potentially be avoided because designers who are already familiar with VHDL-AMS are not required to learn abstract modeling methods. VHDL-AMS was designed to allow a textual description of AMS circuits which can be simulated. Since an LHPN simulation cycle behaves in the same way that the VHDL-AMS simulation cycle behaves, the simulations that would result from an LHPN that has been converted from VHDL-AMS are the same as the simulations that would result from the original VHDL-AMS.

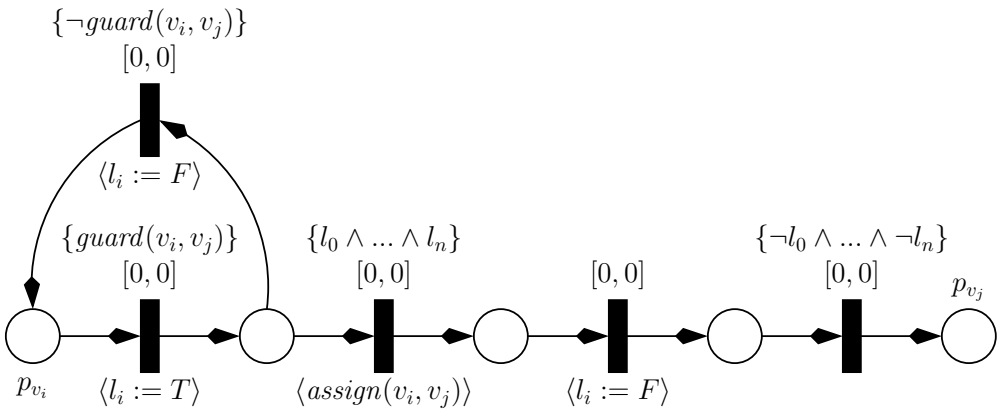


Figure 2.12. Converting sync labels into an LHPN representation.

The VHDL-AMS to LHPN compiler is built using methods described in [71] and currently works with a subset of the VHDL-AMS language. Methods for generating LHPNs from many VHDL statements for representing digital systems are described in [71]. Specifically, variables of types **std_logic** for representing Boolean signals are allowed and sequential behavior can be specified using **process** statements without sensitivity lists. Within a **process**, supported statements are **wait**, signal assignment, **if-use**, **case**, and **while-loop**.

Simulators that support the VHDL-AMS extensions seem to vary in the semantics that are implemented. Therefore, a subset of the AMS extensions has been selected such that the semantics seem to be fairly consistent across simulators. The supported subset of VHDL-AMS allows the creation of a continuous value using a **quantity** of type **real**, the initialization of continuous variables using **break** statements, and the assignments of rates to real quantities using the **'dot** notation within simultaneous **if-use** and **case-use** statements. Additionally, the use of **'above** to test the value of real quantities, and the specification of properties using **assert** statements is allowed. For convenience, VHDL-AMS descriptions also use procedures defined in the `handshake` and `nondeterminism` packages [57]. The **assign** procedure performs an assignment to a signal at some random time within a bounded range specified by its parameters and waits until the assignment has been performed before returning. The **span** procedure takes two real values and returns a random value within that range. The **span** procedure is used to assign a range of rates to a continuous variable.

An LHPN representation of the simultaneous **if-use** statement in Figure 2.13 is shown in Figure 2.14. The condition in each **if** statement is mapped to a guard on a transition and the inverses of the other conditions are asserted to ensure that only a single transition is enabled to fire at any one time. A rate assignment to a continuous variable is performed when a condition is satisfied by associating the rate assignment with the particular transition. A very similar translation is used for the simultaneous **case-use** statement. Additionally, Boolean variables corresponding to the possible rates on the continuous variable are updated to reflect the new rate. These Boolean variables are used to prevent repeated firings of the same transition.

A VHDL-AMS **assert** statement can be used to state basic safety properties about the system. Figure 2.15 shows the LHPN for representing the statement **assert**(f) where f is a Boolean statement over the variables in the system. The net in Figure 2.15 operates

```
if b1 use
   v'dot == span(l1, u1);
elsif b2 use
   v'dot == span(l2, u2);
else
   v'dot == span(l3, u3);
end use;
```

Figure 2.13. VHDL-AMS **if-use** statement.

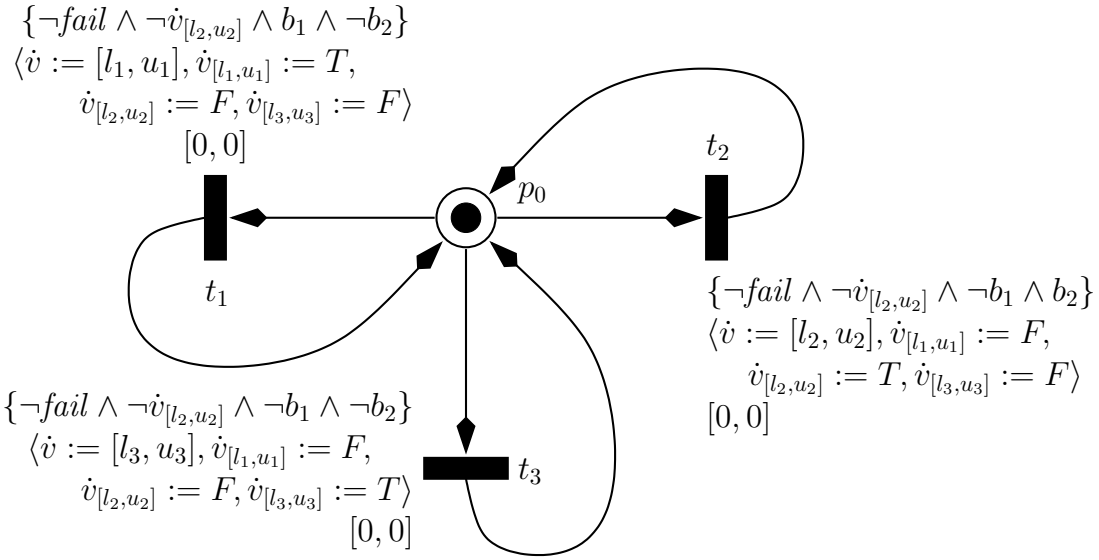


Figure 2.14. Representing VHDL-AMS **if-use** statements as LHPNs.

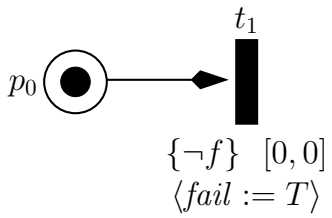


Figure 2.15. Representing VHDL-AMS **assert** statements as LHPNs.

by firing a transition when f is no longer satisfied. Upon firing the transition, the signal *fail* is set to **true**. In order to support continuous quantities in **simultaneous use** and **assert** statements, the **'above** over real quantities is converted to an inequality within the guards on the transitions.

Figure 2.16 shows a VHDL-AMS description for the circuit in Figure 2.1. This description tracks the real quantity *Vout* that represents the output voltage. The LHPN shown in Figure 2.17 is automatically generated from the VHDL-AMS model in Figure 2.16. While similar to the VHDL-AMS description in [55], the VHDL-AMS shown in Figure 2.16 is more concise because the analysis allows rates to be specified as ranges. This model tracks the real quantity *Vout* that represents the output voltage. The **break** statement sets the initial value for *Vout*. The Boolean variable Vin determines the rate of *Vout* using the **if-use** statements. When Vin is 0, *Vout* increases at a rate between 18 and 22 mV/μs. When Vin is 1, *Vout* decreases at a rate between -22 and -18 mV/μs. The **if-use** statement is compiled into the LHPN in Figure 2.17a. The **process** statement is compiled into the LHPN in Figure 2.17b. Initially *Vout* is -1000 mV and increasing between 18 and 22 mV/μs. After 100 μs, Vin is assigned to one by the **assign** function which causes *Vout* to begin decreasing at a rate of -22 to -18 mV/μs. The **assert** statement is used to check if *Vout* falls below -2000 mV or goes above 2000 mV and is compiled into the LHPN shown in Figure 2.17c which fires a transition to set the Boolean signal *fail* to true when the assertion is violated.

2.9 Approximating Differential Equations

The capability to convert differential equations to approximate LHPN representations is also beneficial since analog circuits are commonly modeled using systems of ordinary differential equations. The approximation approach described in this section relies on the capability to calculate the rates specified by the differential equations for a large number of points within a region of interest. Any mathematical tool can be used to perform this sampling. In this case, MATLAB was used. Since only a sampling of the continuous region is being performed, this approach is approximate in nature rather than conservative. In other words, it may not encompass all the behavior of the original differential equations. However, increasing the number of data points in the sampling improves the approximation. This modeling technique allows a designer to model very complex differential equations; however, since it is approximate in nature, the results

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.handshake.all;
use work.nondeterminism.all;
entity integrator is
end integrator;
architecture switchCap of integrator is
   quantity Vout:real;
   signal Vin:std_logic := '0';
begin
   break Vout => -1000.0; --Initial value
   if Vin='0' use
      Vout'dot == span(18.0, 22.0);
   elsif Vin = '1' use
      Vout'dot == span(-22.0, -18.0);
   end use;
   process begin
      assign(Vin,'1',100,100);
      assign(Vin,'0',100,100);
   end process;
   assert (Vout'above(-2000.0) and
       not Vout'above(2000.0))
      report ''error''
      severity failure;
end switchCap;
```

Figure 2.16. VHDL-AMS for a switched capacitor integrator.

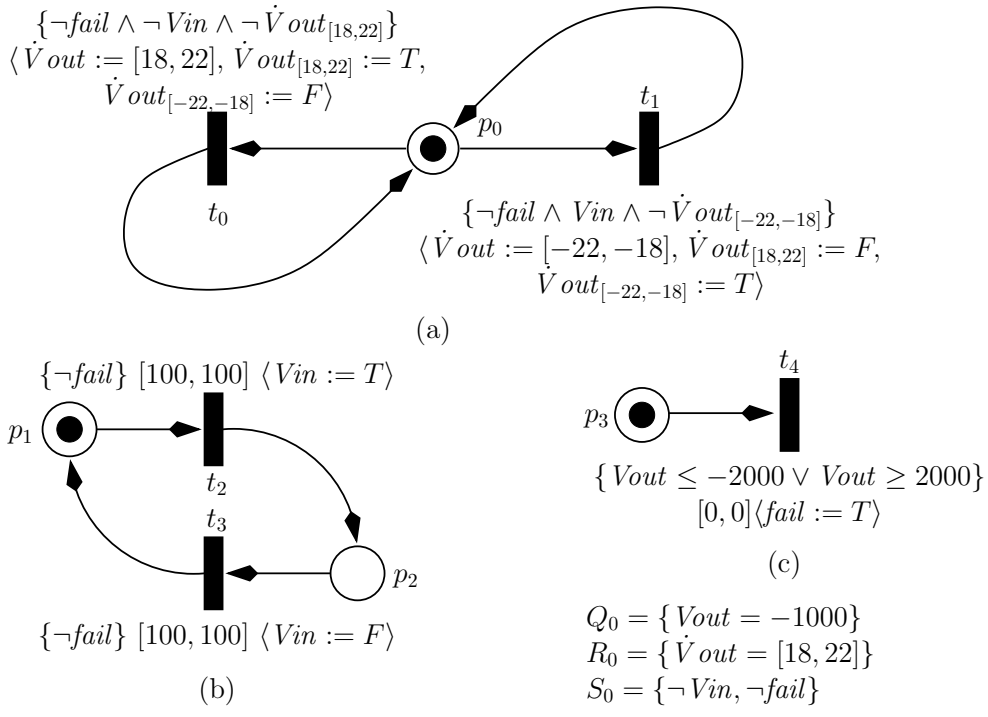


Figure 2.17. LHPN of the switched capacitor integrator generated from VHDL-AMS.

cannot be fully trusted.

After the sampling of the differential equations is performed, the data points are decomposed into subregions in which the rate of change is within a range of constant rates as determined by the minimum and maximum rate of the data points within the subregion—a form that LHPNs can efficiently represent. Any discretization method with the goal of minimizing the resulting number of subregions may be utilized with the additional restrictions that subregions be rectangular in shape and have only a single neighboring subregion on each side in each dimension. The number subregions that are created directly influences the accuracy of the results. Fewer subregions results in greater approximation while more subregions reduces the amount of approximation but increases the analysis time since the resulting LHPN is larger.

After decomposing the data points into discrete subregions, an LHPN can be generated. Figure 2.18 shows an example of a two-dimensional surface divided into subregions where each subregion is approximated by a range of rates. The surface could be any surface expressed by a set of differential equations. The region where $0 \leq x \leq x_0$ and $0 \leq y \leq y_0$ is shown with greater detail. The continuous variables x and y increase or decrease at a range of rates within each region. The discrete transitions t_1 and t_2 allow transitions from p_0 when x and y increase beyond $x_0 - \delta$ and $y_0 - \delta$, respectively, and different rates are necessary. The discrete transitions t_3 and t_4 allow transitions in the reverse direction into p_0 when x and y fall below $x_0 + \delta$ and $y_0 + \delta$, respectively. Note that the delta amount δ is used to prevent rapid transitions between neighboring regions when near the boundary. Upon transitioning, new rate ranges are assigned to each continuous variable that correlate with the ingoing region. Each subregion has a corresponding Petri net representation, resulting in a LHPN that approximates the surface.

One possible method for decomposing the continuous state space into a discrete number of states where rates are assumed to be constant within a range is described here. This approach is similar to that proposed in [45, 46, 47, 48] where the continuous state space is divided into regions and each region is represented in a Boolean manner. From this decomposition, a transition relation is created by selecting test points in each region to determine reachable next states. This Boolean abstraction allows them to perform model checking using standard Boolean based approaches. While a very promising approach, this technique necessarily loses significant accuracy in the abstraction to a Boolean model. The method also is not guaranteed to be conservative since the test points chosen do

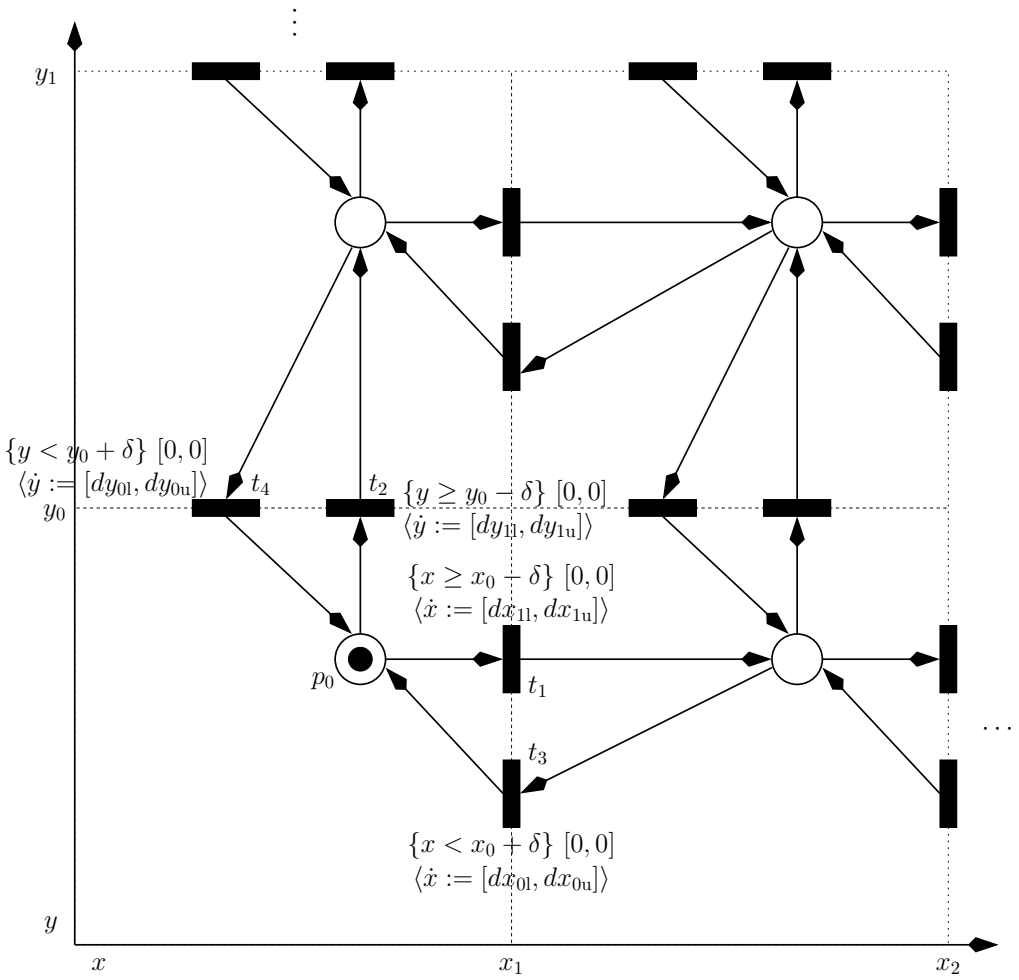


Figure 2.18. Representing differential equations using LHPNs.

not necessarily find all possible reachable states in the original continuous model. As discretization of the continuous space continues, a primary goal is to minimize the number of discrete subregions generated. The size of the final LHPN grows as the number of discrete subregions increases, so it is important to intelligently select where divisions in the continuous space are made.

The continuous space is initially restricted to a finite region by the user, and the space is sampled at many points throughout that region. It is important to oversample the system to ensure that important points are not ignored. In our implementation, a dense matrix of data points for each continuous variable over the user specified range is calculated. A matrix L is then calculated that contains the combined vectors of all continuous variables in the system.

Divisions are iteratively added to L until the user specified number of subregions has been created. A division is defined as a straight cut through the region. A division location is determined by attempting all possible divisions and then selecting the division which has the least cost. Cost is defined as follows:

$$cost(R) = \frac{\sum_{r \in R} l_m(r) N_r}{N}$$

where R is the set of subregions, N_r is the number of data points in subregion r, and N is the number of data points in the entire region. $l_m(r)$ is a measure of the variation of the data points in subregion r:

$$l_m(r) = 1 - \frac{\min_{y \in r} y}{\max_{y \in r} y}$$

Essentially, cost is a weighted average of the variation among the data points in all subregions. Once the regions have been selected, for each continuous variable in the system, the floor and ceiling of all the data points in each subregion is calculated and used as the lower bound rate and upper bound rate, respectively, for that subregion.

This modeling method is illustrated using the tunnel diode oscillator example from [46]. This example is also used by Gupta et al. in [43] to illustrate the analog verification technique they describe. The same numerical parameters as Gupta et al. are used. The tunnel diode oscillator example is shown in Figure 2.19. This circuit is supposed to oscillate, and the goal of verification would be to determine for what parameters and initial conditions it oscillates. This circuit can be described with two differential equations:

$$\frac{dVc}{dt} = \frac{1}{C}(-h(Vc) + Il)$$

$$\frac{dIl}{dt} = \frac{1}{L}(-Vc - R \cdot Il + Vin)$$

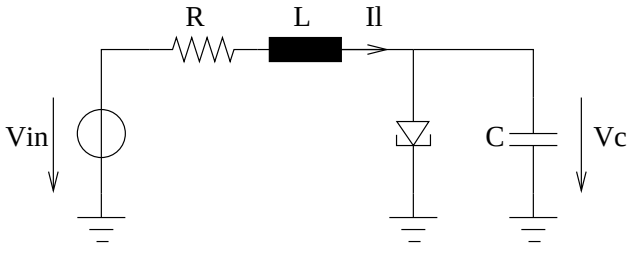


Figure 2.19. Tunnel diode oscillator circuit.

where h is a piecewise model of the tunnel diode behavior:

$$h(V_{\mathrm{d}}) = \begin{cases} 6.0105V_{\mathrm{d}}^3 - 0.9917V_{\mathrm{d}}^2 + 0.0545V_{\mathrm{d}} & 0 \leq V_{\mathrm{d}} \leq 0.055 \\ 0.0692V_{\mathrm{d}}^3 - 0.0421V_{\mathrm{d}}^2 + 0.004V_{\mathrm{d}} + 8.95794 \cdot 10^{-4} & 0.055 \leq V_{\mathrm{d}} \leq 0.35 \\ 0.2634V_{\mathrm{d}}^3 - 0.2765V_{\mathrm{d}}^2 + 0.0968V_{\mathrm{d}} - 0.0112 & 0.35 \leq V_{\mathrm{d}} \leq 0.50 \end{cases}$$

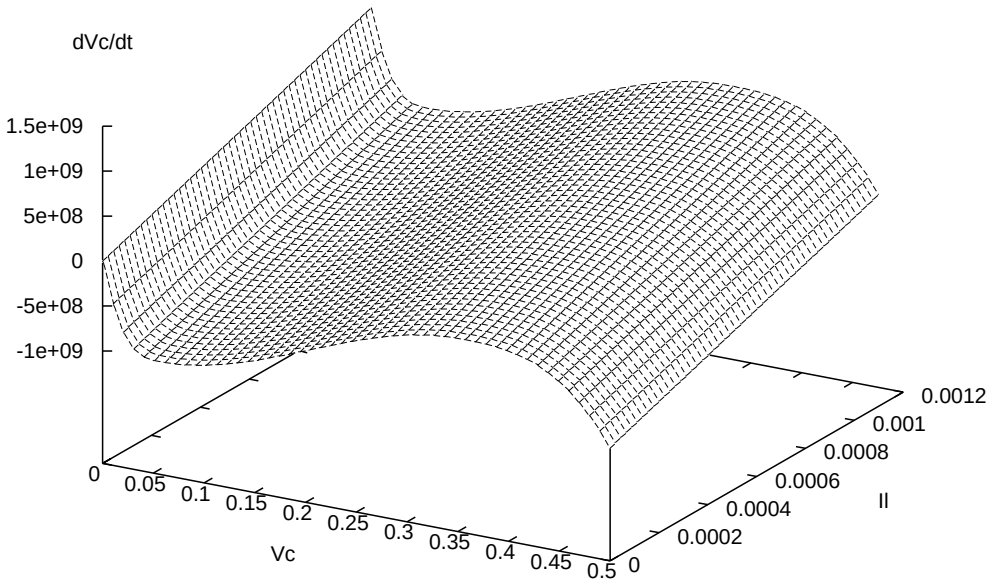
Plots of these two functions are shown in Figure 2.20.

Using this discretization method, 16 discrete subregions are required to model the oscillatory/nonoscillatory behavior of the circuit. Using a naive discretization method where divisions are selected so that all subregions are of equal size requires 36 discrete subregions to reproduce the oscillatory/nonoscillatory behavior. Plots showing how the continuous space is divided using this approach and the naive approach are shown in Figure 2.21 and Figure 2.22, respectively. Note, that in these examples, the rates of Vc and Il within each region are specified as single values rather than ranges. Figure 2.23 shows how the continuous space would have been divided if 16 regions is used. In this figure, the approximation contains no indication that at low voltage there is a spike in the rate of Vc. This over approximation causes incorrect behavior in simulation.

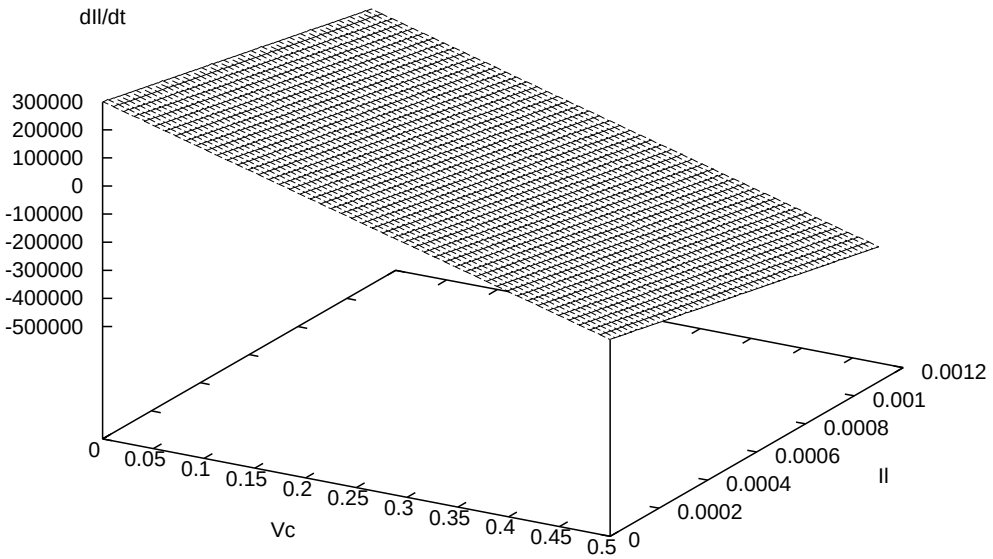
2.10 Summary

The LHPN model, a key contribution of this dissertation, provides a mechanism for modeling AMS circuits. The use of timing bounds on transitions and labelings containing continuous variables make LHPNs well-suited for this purpose. The languages that are accepted by LHPNs are specified to be the set of allowed executions as specified by the formal semantics of this model. LHPNs can be simulated using an approach that relies on selection of a delta time step; however, due to this reliance the executions that result from the simulator do not correspond directly to the formal semantics of the LHPN, i.e., it may allow executions that are not possible and disallow executions that are possible.

A method for converting LHA, an alternate hybrid system modeling formalism, into LHPNs is presented. The languages allowed by LHAs can be represented using LHPNs as shown by the straightforward conversion process. Therefore, any assertions based on the LHPN analysis are also assertions on the original LHA. VHDL-AMS provides a convenient input format for LHPNs. An LHPN that is generated from VHDL-AMS corresponds directly to the semantics of the VHDL-AMS that is believed to be implemented by a variety of VHDL-AMS simulators. Therefore, statements derived from the analysis of the resulting LHPN apply directly to the original VHDL-AMS.

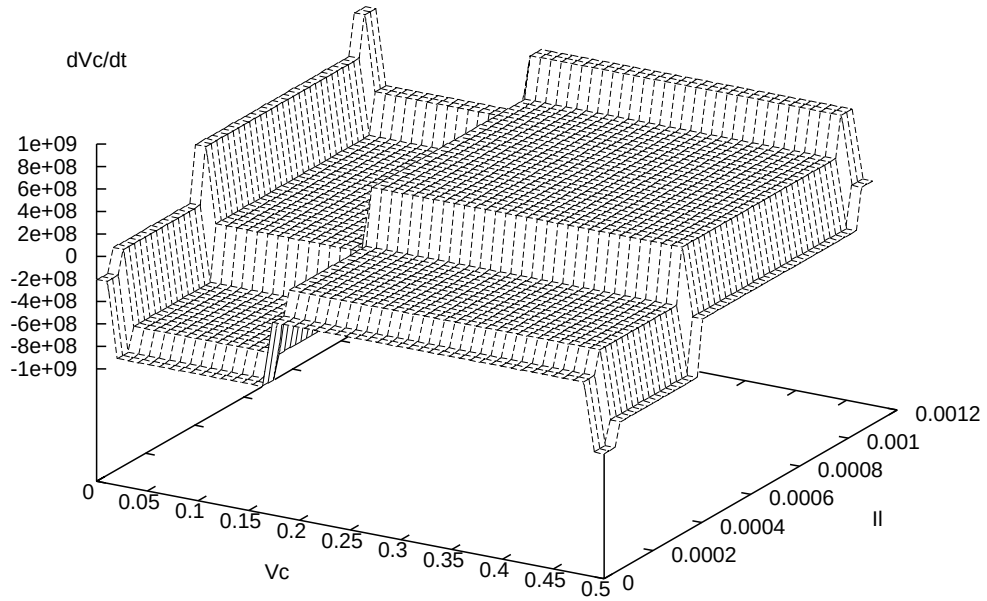


(a)

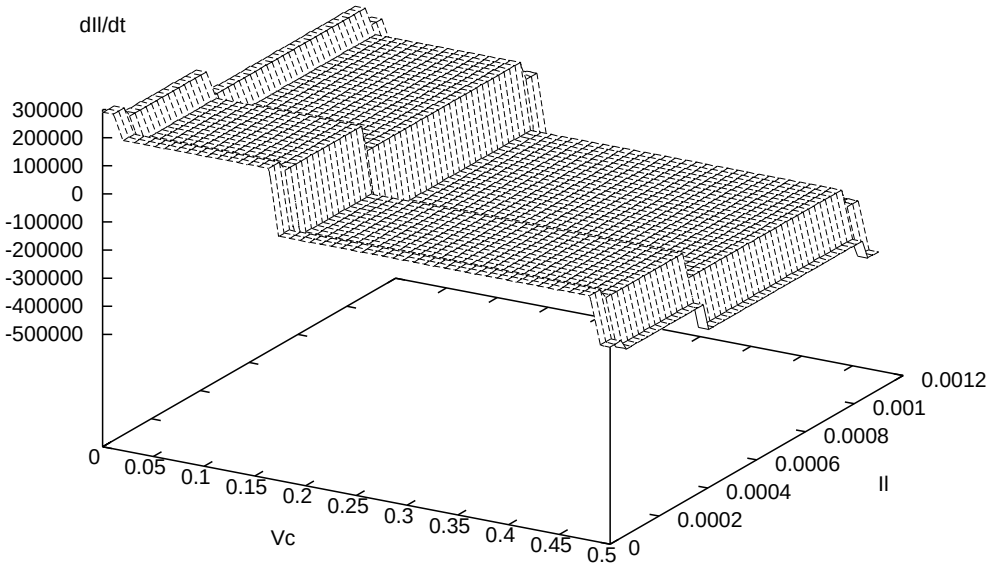


(b)

Figure 2.20. Differential equation plots for tunnel diode oscillator. (a) Rate of change for Vc for various operating points. (b) Rate of change for Il for various operating points.

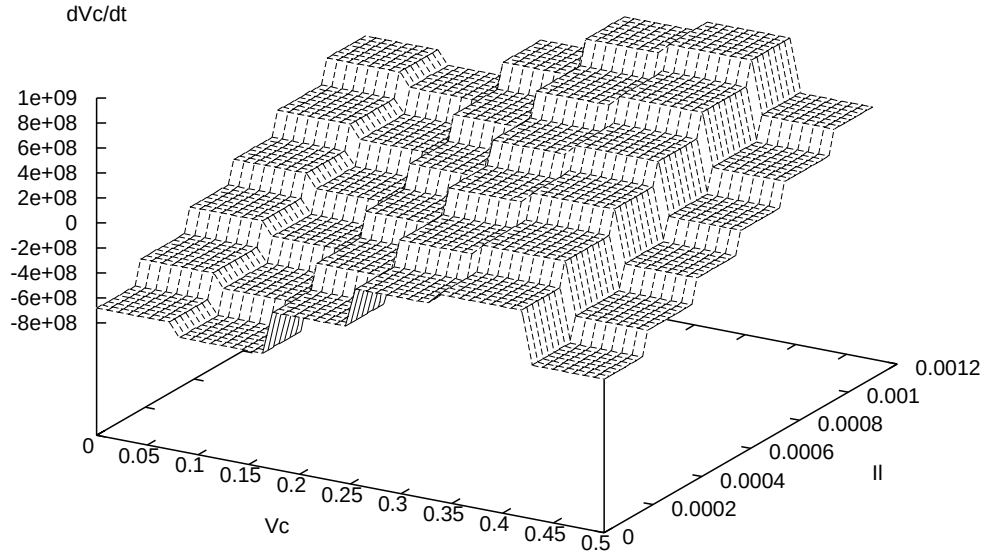


(a)

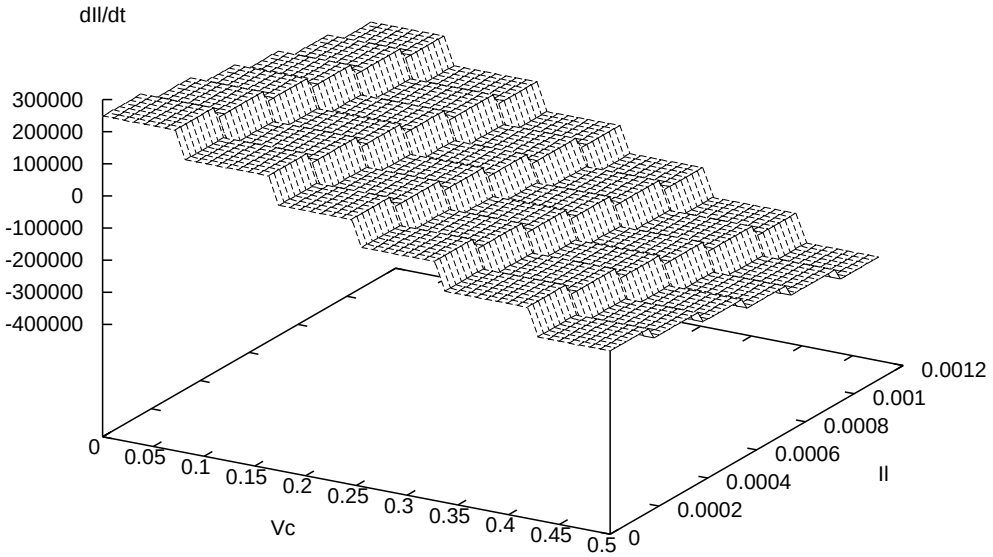


(b)

Figure 2.21. Approximation of continuous rates using described method. (a) Rate of change for Il for various operating points. (b) Rate of change for Vc for various operating points.

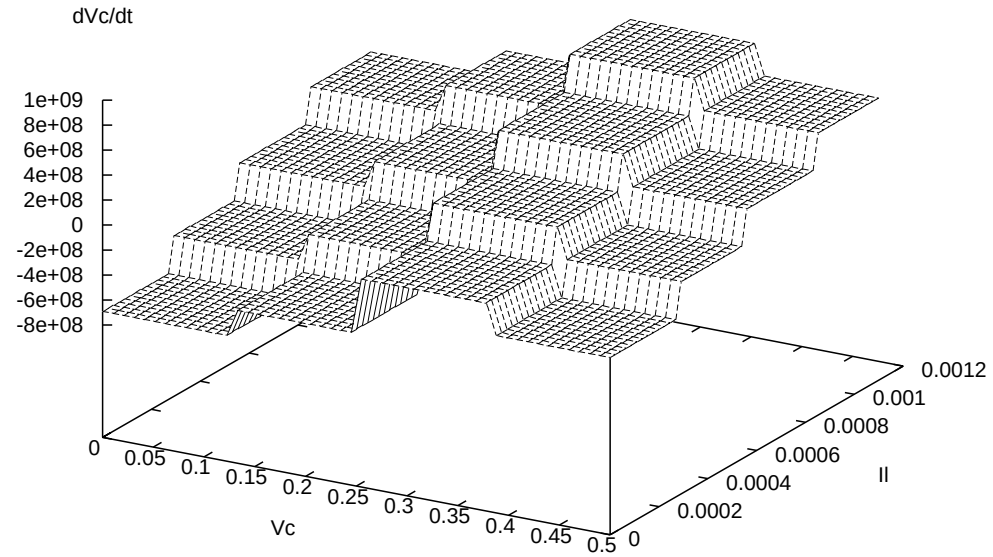


(a)

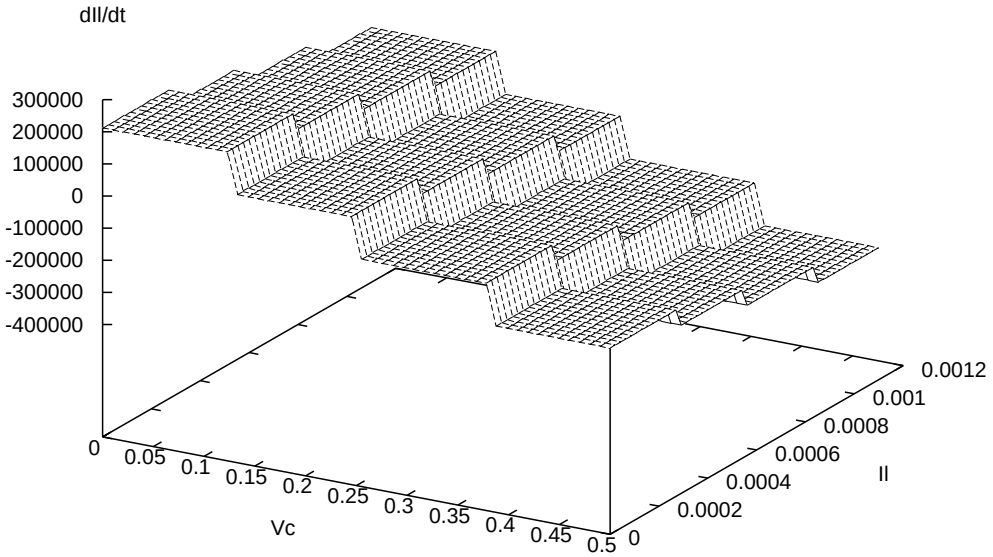


(b)

Figure 2.22. Approximation of continuous rates with 36 subregions using naive method. (a) Rate of change for Il for various operating points. (b) Rate of change for Vc for various operating points.



(a)



(b)

Figure 2.23. Approximation of continuous rates with 16 subregions using naive method. (a) Rate of change for Il for various operating points. (b) Rate of change for Vc for various operating points.

The method for approximating differential equations using LHPNs allows for systems of differential equations to be modeled and analyzed using LHPNs. Since the conversion is approximate in nature, assertions about the original differential equation system based on LHPN analyses are not conclusive. The capability to model and analyze differential equation models using LHPNs, however, may be useful to designers by providing one possible approach to performing state space explorations rather than just series of simulations.

CHAPTER 3

SYMBOLIC MODEL OF LHPNS

In order for analysis to proceed, a symbolic model is generated from the LHPN that contains the essential information for analysis. The symbolic model consists of three components: an *invariant*, a set of possible rates, and a set of *guarded commands*. These three components of the symbolic model are used by the symbolic model checkers to perform the state space exploration of the original LHPN. Therefore, it is important that the symbolic model accurately reflects the semantics of the original LHPN. Specifically, if the model checker is performing an exact state space exploration given a symbolic model, the resulting state space should be equivalent to the language specified by the original LHPN. Similarly, if the model checker is performing an abstract state space exploration given the symbolic model, the resulting state space should fully encompass the language specified by the original LHPN, plus some additional state space, potentially. The remainder of this chapter provides an algorithmic construction of the symbolic model components.

3.1 Implicit Model Variables

Before constructing the symbolic model, a set of real variables and two additional sets of Boolean variables are created in addition to the sets defining an LHPN. The set of real variables is used to track the values of the clocks on each transition. These real variables are referred to as *transition clocks* and denoted as c_t for the clock on transition t. The first set of Boolean variable is known as *clock active variables*, and is used to keep track of whether or not the clocks on transitions are active. Clock active variables are denoted by a_t for the clock active variable on transition t. The second set of Boolean variables is known as *Boolean rate variables* used for determining at which rate each continuous variable is currently changing. Boolean rate variables are denoted by $\dot{v}_{[r_l,r_u]}$ for the Boolean variable corresponding to the continuous variable v currently advancing at a range of rates $[r_l, r_u]$ These sets are formally defined as follows:

- $C : \{c_t \mid t \in T\}$ where c_t is the real value of the clock on transition t;
- $A : \{a_t \mid t \in T\}$ where a_t is a Boolean value that indicates if the clock on a transition (c_t) is active; and
- $BR : \{\dot{v}_{[r_l,r_u]} \mid v \in V \wedge ((v, r_l, r_u) \in R_0 \vee \exists t.(v, r_l, r_u) \in RA(t))\}$.

The notation $BR(v)$ is used to represent the subset of variables in BR that affects the rate of the continuous variable v, i.e., those of the form $\dot{v}_{[r_l,r_u]}$.

3.2 Invariant

The invariant ($\phi_{\mathcal{I}}$) is an HSL statement that must be satisfied in every state of the system and is calculated as shown in Equation 3.1.

$$\phi_{\mathcal{I}} = \Phi \wedge \bigwedge_{t \in T} (a_t \Rightarrow \bullet t \wedge En(t) \wedge 0 \leq c_t \leq u(t)) \wedge (\overline{a_t} \Rightarrow \overline{\bullet t} \vee \widetilde{En(t)}) \tag{3.1}$$

The invariant first states that only the discrete states (represented by Φ) can be reached. The formula Φ is found by performing a state space exploration of the LHPN while neglecting the continuous variables. The discrete state space exploration is based on the Petri net algorithm described in [61] with extensions to include values of Boolean signals and Boolean rate variables in the state space. In other words, Φ is a formula over the Boolean variables for the Petri net marking, Boolean signals, and Boolean rate variables. Calculation of Φ is performed by first finding the initial Boolean state, iteratively applying transition firings beginning from the initial state until all states are found, then inserting clock active Boolean variables based on the presets of transitions and the Boolean portion of the enabling conditions.

The initial Boolean state is constructed in a straightforward manner based on M_0, S_0, and R_0 as shown in the algorithm in Figure 3.1. For the integrator example in Figure 2.5, the initial Boolean state is calculated to be:

$$initBoolState = p_0 \overline{p_1}\, p_2\, \overline{p_3}\, p_4\, \overline{Vin}\, \overline{fail}\, \dot{V}\, out_{[18,22]}\, \overline{\dot{V}\, out_{[-22,-18]}}$$

After finding the initial Boolean state, the four sets of Boolean expressions representing *all predecessors marked* (APM), *no predecessors marked* (NPM), *all successors marked* (ASM), and *no successors marked* (NSM) are calculated. These Boolean expressions are constructed from the LHPN using the algorithm shown in Figure 3.2, and are used to simulate transition firings when calculating the Boolean state space. This is demonstrated

```
buildInitBoolState(⟨P, T, B, V, F, L, M0, S0, Q0, R0⟩)
    initBoolState = true
    // Initial marking
    for each (p ∈ P)
        if (p ∈ M0)
            initBoolState = initBoolState ∧ p
        else
            initBoolState = initBoolState ∧ p̄
        end if
    end for
    // Initial Boolean signal values
    for each ((b, val) ∈ S0)
        if (val == true)
            initBoolState = initBoolState ∧ s
        else if (val == false)
            initBoolState = initBoolState ∧ s̄
        end if
    end for
    // Initial Boolean rate variable values
    for each ((v, rl, ru) ∈ R0)
        initBoolState = initBoolState ∧ v̇[rl,ru]
        for each (x ∈ BR(v) − v̇[rl,ru])
            initBoolState = initBoolState ∧ x̄
        end for
    end for
    return initBoolState
end
```

Figure 3.1. Algorithm for constructing the initial state of the LHPN.

by firing transition t_1 in the the integrator example from Figure 2.5. The following Boolean expressions are first constructed:

$$\begin{aligned} APM_{t_1} &= p_2 \\ NPM_{t_1} &= \overline{p_2} \\ ASM_{t_1} &= p_3\, Vin \\ NSM_{t_1} &= p_3\, Vin \end{aligned}$$

By cofactoring with respect to an APM_{t_1}, the set of states which enabled that transition are calculated. Additionally, the preset places are removed from the expression. In the example, cofactoring the initial state by APM_{t_1} results in:

$$initBoolState_{APM_{t_1}} = p_0 \overline{p_1}\, \overline{p_3}\, p_4\, \overline{Vin}\, \overline{fail}\, \dot{V}\, out_{[18,22]}\, \overline{\dot{V}\, out_{[-22,-18]}}$$

Next, taking the product of the result and NPM_{t_1} simulates the removal of the tokens

buildTransStatements($\langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$)
  $transStatements = \emptyset$
  for each $(t \in T)$
    `// Marking`
    $APM_t = \bigwedge_{p_i \in {}^\bullet t} p_i$
    $NPM_t = \bigwedge_{p_i \in {}^\bullet t} \overline{p_i}$
    $ASM_t = \bigwedge_{p_i \in t^\bullet} p_i$
    $NSM_t = \bigwedge_{p_i \in t^\bullet} p_i$
    `// Boolean Assignments`
    for each $((b, \textbf{true}) \in BA(t))$
      $ASM_t = ASM_t \wedge b$
      $NSM_t = NSM_t \wedge b$
    end for
    for each $((b, \textbf{false}) \in BA(t))$
      $ASM_t = ASM_t \wedge \overline{b}$
      $NSM_t = NSM_t \wedge b$
    end for
    `// Rate Assignments`
    for each $((v, r_l, r_u) \in RA(t))$
      $ASM_t = ASM_t \wedge \dot{v}_{[r_l, r_u]}$
      $NSM_t = NSM_t \wedge \dot{v}_{[r_l, r_u]}$
      for each $(x \in BR(v) - \dot{v}_{[r_l, r_u]})$
        $ASM_t = ASM_t \wedge \overline{x}$
        $NSM_t = NSM_t \wedge x$
      end for
    end for
    `// Boolean Portion of Enablings`
    $originalAPM = APM_t$
    $originalNPM = NPM_t$
    if (*boolPortion*$(En(t))$
      $transStatements = transStatements \cup \langle APM_t, NPM_t, ASM_t, NSM_t \rangle$
    else
      for each $(product \in boolPortion(En(t)))$
        $APM_t = originalAPM$
        $NPM_t = originalNPM$
        for each $(b \in B)$
          if $(\overline{b} \in product)$
            $APM_t = APM_t \wedge \overline{b}$
            $NPM_t = NPM_t \wedge \overline{b}$
          else if $(b \in product)$
            $APM_t = APM_t \wedge b$
            $NPM_t = NPM_t \wedge b$
          end if
        end for
        $transStatements = transStatements \cup \langle APM_t, NPM_t, ASM_t, NSM_t \rangle$
      end for
    end if
  end for
  return *transStatements*
end

Figure 3.2. Algorithm for constructing characteristic functions used by **findDisStates**.

from the preset places. In the examples, this results in:

$$initBoolState_{APM_{t_1}} \wedge NPM_{t_1} = p_0 \overline{p_1}\, \overline{p_2}\, \overline{p_3}\, p_4\, \overline{Vin}\, \overline{fail}\, \dot{V}out_{[18,22]}\, \overline{\dot{V}out_{[-22,-18]}}$$

The next step is to existentially abstract the corresponding NSM from the result. This has the effect of removing the successor places and variables to which Boolean assignments are being performed. Existentially abstracting NSM_{t_1} from the example, results in:

$$\exists_{NSM_{t_1}} (initBoolState_{APM_{t_1}} \wedge NPM_{t_1}) = p_0 \overline{p_1}\, \overline{p_2}\, p_4\, \overline{fail}\, \dot{V}out_{[18,22]}\, \overline{\dot{V}out_{[-22,-18]}}$$

The final step is to apply the corresponding ASM. This has the effect of placing tokens in the postset places and applying the Boolean assignments. The final result is the next state:

$$(\exists_{NSM_{t_1}} (initBoolState_{APM_{t_1}} \wedge NPM_{t_1})) \wedge ASM_{t_1} =$$
$$p_0 \overline{p_1}\, \overline{p_2}\, p_3\, p_4\, Vin\, \overline{fail}\, \dot{V}out_{[18,22]}\, \overline{\dot{V}out_{[-22,-18]}}$$

Note that in this brief example, no Boolean rate variable updates or Boolean enabling conditions are present; however, they are updated and enforced as transition firings are applied in a similar manner. The complete algorithm for finding the discrete states is shown in Figure 3.3. The discrete state space for the integrator example in Figure 2.5 is:

$$\begin{aligned}
\Phi \;=\; & (p_0\, \overline{p_1}\, p_2\, \overline{p_3}\, p_4\, \overline{fail}\, \overline{Vin}\, \overline{\dot{V}out_{[-22,-18]}}\, \dot{V}out_{[18,22]}) \vee \\
& (p_0\, \overline{p_1}\, \overline{p_2}\, p_3\, p_4\, \overline{fail}\, Vin\, \overline{\dot{V}out_{[-22,-18]}}\, \dot{V}out_{[18,22]}) \vee \\
& (\overline{p_0}\, p_1\, \overline{p_2}\, p_3\, p_4\, \overline{fail}\, Vin\, \dot{V}out_{[-22,-18]}\, \overline{\dot{V}out_{[18,22]}}) \vee \\
& (\overline{p_0}\, p_1\, p_2\, \overline{p_3}\, p_4\, \overline{fail}\, \overline{Vin}\, \dot{V}out_{[-22,-18]}\, \overline{\dot{V}out_{[18,22]}}) \vee \\
& (p_0\, \overline{p_1}\, p_2\, \overline{p_3}\, \overline{p_4}\, fail\, \overline{Vin}\, \overline{\dot{V}out_{[-22,-18]}}\, \dot{V}out_{[18,22]}) \vee \\
& (p_0\, \overline{p_1}\, \overline{p_2}\, p_3\, \overline{p_4}\, fail\, Vin\, \overline{\dot{V}out_{[-22,-18]}}\, \dot{V}out_{[18,22]}) \vee \\
& (\overline{p_0}\, p_1\, \overline{p_2}\, p_3\, \overline{p_4}\, fail\, Vin\, \dot{V}out_{[-22,-18]}\, \overline{\dot{V}out_{[18,22]}}) \vee \\
& (\overline{p_0}\, p_1\, p_2\, \overline{p_3}\, \overline{p_4}\, fail\, \overline{Vin}\, \dot{V}out_{[-22,-18]}\, \overline{\dot{V}out_{[18,22]}})
\end{aligned}$$

After calculating the discrete state space, Φ, the next step in constructing the system invariant, $\phi_{\mathcal{I}}$, as shown in Equation 3.1 is to insert known information about the continuous state space. This is performed using the clock active variables. Specifically, for a transition's clock to be active, the preset must be marked, the enabling condition must be satisfied, and the clock must be greater than zero but not greater than its upper bound.

```
findDiscreteStates(initStates, transStatements)
    reached = initStates
    from = initStates
    do
        to = false
        for each (⟨APM_i, NPM_i, ASM_i, NSM_i⟩ ∈ transStatements)
            to = ∃_{NSM_i}(from_{APM_i} ∧ NPM_i) ∧ ASM_i
        end for
        from = to − reached
        reached = reached ∨ from
    while (from ≠ false)
    return reached
end
```

Figure 3.3. Algorithm for constructing Φ.

This portion of $\phi_{\mathcal{I}}$ prevents an active clock from exceeding its upper bound. The last part of $\phi_{\mathcal{I}}$ states that if a transition's clock is not active it must either have an unmarked place in its preset or the *nonstrict inverse* $(\widetilde{En(t)})$ of the enabling condition must be satisfied. In the nonstrict inverse, all $\geq$ separation predicates become $\leq$ separation predicates and vice-versa. For example, the nonstrict inverse of the HSL formula $a \wedge x \leq 2000$ is $\overline{a} \vee x \geq 2000$. The last two portions of $\phi_{\mathcal{I}}$ when taken together enforce the activation or deactivation of a clock if a changing continuous variable should cause an enabling condition to change evaluation. For the integrator example in Figure 2.5, the invariant is calculated to be:

$$
\begin{aligned}
\phi_{\mathcal{I}} \;=\; & \Phi \wedge (a_{t_0} \Rightarrow p_0 \wedge \mathit{Vin} \wedge c_{t_0} = 0) \wedge (\overline{a_{t_0}} \Rightarrow \overline{p_0} \vee \overline{\mathit{Vin}}) \wedge \\
& (a_{t_1} \Rightarrow p_1 \wedge \overline{\mathit{Vin}} \wedge c_{t_1} = 0) \wedge (\overline{a_{t_1}} \Rightarrow \overline{p_1} \vee \mathit{Vin}) \wedge \\
& (a_{t_2} \Rightarrow p_2 \wedge 0 \leq c_{t_2} \leq 100) \wedge (\overline{a_{t_2}} \Rightarrow \overline{p_2}) \wedge \\
& (a_{t_3} \Rightarrow p_3 \wedge 0 \leq c_{t_3} \leq 100) \wedge (\overline{a_{t_3}} \Rightarrow \overline{p_3}) \wedge \\
& (a_{t_4} \Rightarrow p_4 \wedge c_{t_4} = 0 \wedge \\
& (\mathit{Vout} \leq -2000 \vee \mathit{Vout} \geq 2000)) \wedge \\
& (\overline{a_{t_4}} \Rightarrow \overline{p_4} \vee (\mathit{Vout} \geq -2000 \wedge \mathit{Vout} \leq 2000))
\end{aligned}
$$

3.3 Possible Rate Sets

The set of possible rates ($\mathcal{R}$) consists of an HSL statement indicating a possible Boolean rate assignment and the set of rate assignments to continuous variables corresponding to the statement ($\langle \phi_R, R \rangle$). This set is constructed from Φ, the Boolean state set, by existentially abstracting all nonrate Boolean variables. Each product term in Φ

corresponds to a ϕ_R of a pair in $\mathcal{R}$. The Boolean rate assignment sets (R) are built from the product terms as shown in the algorithm in Figure 3.4. First a conjunction of the nonrate Boolean variables is constructed. Next, for each product term in the Boolean state, the Boolean variables are existentially abstracted, resulting in a conjunction of Boolean rate variables. For each Boolean rate variable that appears in the positive form, a mapping of the continuous variable to the rate range is created. For example, the possible rate set for the integrator LHPN in Figure 2.5 is:

$$\begin{aligned}\mathcal{R} \;=\; & \{\langle \dot{V}out_{[-22,-18]} \wedge \overline{\dot{V}out_{[18,22]}}, \{\dot{V}out := [-22,-18]\}\rangle, \\ & \langle \overline{\dot{V}out_{[-22,-18]}} \wedge \dot{V}out_{[18,22]}, \{\dot{V}out := [18,22]\}\rangle\}\end{aligned}$$

3.4 Guarded Commands

The set of guarded commands ($\mathcal{C}$) is used to determine in each state which transitions are enabled and the effect on the state due to the firing of a transition. It is constructed using a set of *primary guarded commands* ($\mathcal{C}_P$) and a set of *secondary guarded commands* ($\mathcal{C}_S$). Each guarded command consists of a *guard*, ϕ_G, represented using an HSL formula and a set of *commands*, $\mathcal{A}$, to be performed when the guard is satisfied.

A primary guarded command is created for each transition $t \in T$. The guard for transition t ensures that the preset for t is marked, the enabling condition on t is satisfied, and the clock associated with t is active and exceeds its lower bound. The commands for transition t cause the postset of t to become marked and the application of the assignments associated with t. Formally, the set of primary guarded commands is defined as follows:

$$\mathcal{C}_P = \{\langle \phi_{G_P}(t), \mathcal{A}_P(t)\rangle \mid t \in T\} \tag{3.2}$$

buildPossibleRateSet(Φ)
  $boolVars = \prod_{p\in P} p \wedge \prod_{b\in B} b \wedge \prod_{t\in T} a_t$
  $\mathcal{R}$ = ∅
  for each (product term in Φ)
    $\phi_R = \Phi$.ExistAbstract($boolVars$)
    $R = \{\dot{v} := [r_l, r_u] \quad | \quad \exists \dot{v}_{[r_l,r_u]} \in BR.(\phi_R \wedge \dot{v}_{[r_l,r_u]})\}$
    $\mathcal{R} = \mathcal{R} \cup \langle \phi_R, R\rangle$
  end for
  return $\mathcal{R}$
end

Figure 3.4. Algorithm for constructing the possible rate set, $\mathcal{R}$.

where $\phi_{G_P}(t) = (\bullet t \wedge \overline{t \bullet - \bullet t} \wedge En(t) \wedge a_t \wedge c_t \geq l(t))$ and $\mathcal{A}_P(t) = \{(\bullet t - t\bullet) := F, (t\bullet) := T, a_t := F, c_t := [-\infty, \infty], BA(t), VA(t), RA(t)\}$. The primary guarded command for transition t_2 in Figure 2.5 is:

$$\begin{aligned} \phi_{G_P}(t_2) &= p_2 \wedge \overline{p_3} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\ \mathcal{A}_P(t_2) &= \{p_2 := F, p_3 := T, Vin := T, \\ &\quad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\} \end{aligned}$$

Two secondary guarded commands are created for each transition $t \in T$, one to activate and one to deactivate the clock associated with t. The first one activates the clock for t and sets it to zero when its preset is marked and its enabling condition is true. The second one deactivates the clock when t is no longer enabled and sets its values to $[-\infty, \infty]$. This has the effect of removing the clock from the state space. The set of secondary guarded commands is defined as follows:

$$\mathcal{C}_S = \{\langle \phi_{G_{SA}}(t), \mathcal{A}_{SA}(t) \rangle, \langle \phi_{G_{SD}}(t), \mathcal{A}_{SD}(t) \rangle \mid t \in T\} \tag{3.3}$$

where $\phi_{G_{SA}}(t) = \bullet t \wedge En(t) \wedge \overline{a_t}$, $\mathcal{A}_{SA}(t) = \{a_t := T, c_t := [0, 0]\}$, $\phi_{G_{SD}}(t) = (\overline{\bullet t} \vee \widetilde{En(t)}) \wedge a_t$, and $\mathcal{A}_{SD}(t) = \{a_t := F, c_t := [-\infty, \infty]\}$. The activating and deactivating guarded commands for transition t_0 in Figure 2.5 are:

$$\begin{aligned} \phi_{G_{SA}}(t_0) &= p_0 \wedge Vin \wedge \overline{a_{t_0}} \\ \mathcal{A}_{SA}(t_0) &= \{a_{t_0} := T, c_{t_0} := [0, 0]\} \\ \phi_{G_{SD}}(t_0) &= (\overline{p_0} \vee \overline{Vin}) \wedge a_{t_0} \\ \mathcal{A}_{SD}(t_0) &= \{a_{t_0} := F, c_{t_0} := [-\infty, \infty]\} \end{aligned}$$

The sets $\mathcal{C}_P$ and $\mathcal{C}_S$ are merged to form the set $\mathcal{C}$. It is necessary to merge these commands because the firing of a transition may result in the activation or deactivation of clocks associated with other transitions by changing the marking or the values of the Boolean or continuous variables. The basic idea is that for each transition, t, the effect of its assignments associated with its primary guarded command $\mathcal{A}_P(t)$ must be checked against the guards $\phi_{G_{SA}}(t')$ and $\phi_{G_{SD}}(t')$ for each other transition t' to determine if the assignment may have enabled the guard. If the assignments have no effect on the guard or disable it, then the secondary for t' is not merged with the primary for t. If the assignment would make the guard true, then the commands associated with the secondary must be

combined with those for the primary. Finally, if the assignment may have changed the guard's evaluation, then two guarded commands must be constructed. One is for the case in which the guard for the secondary is true in which the commands are merged, and the other is for when the guard is false in which the secondary commands are not merged. A detailed algorithm for merging guarded commands is shown in Figure 3.5. Note that after performing the merge operation, secondary guarded commands whose guards contain inequalities are inserted into the final guarded command set. This is necessary because as time moves forward, the secondary guarded commands could become enabled and cause clocks to be activated or deactivated. However, before the secondary guarded commands are added, their guards must be modified to enforce the threshold on the continuous variables. The algorithm for modifying the guards to enforce thresholds is shown in Figure 3.6. For example, consider a situation where a transition has the enabling condition $x \geq 5$. The clock on this transition can be activated either when its preset becomes marked when x is already greater than or equal to five, or by x becoming equal to five while the preset is already marked. The first case is handled by the merged guarded command while the second case should be handled by a secondary guarded command that ensures that x is equal to five and continues to increase above five, i.e., when $x \geq 5 \wedge x \leq 5 \wedge \texttt{incr}(x)$ where $\texttt{incr}(x)$ returns the disjunction of the Boolean rate variables where the rates are increasing. Similarly, the algorithm in Figure 3.6 makes use of $\texttt{decr}(x)$ which returns a disjunction of the Boolean rate variables where x is decreasing.

In the integrator example (see Figure 2.5), since the primary guarded command for t_2 assigns *Vin* to true, a condition in the guard of the activating guarded command on t_0, they are merged into the guarded command shown below:

$$\begin{aligned}
\phi_G(t_2, t_0) &= p_0 \wedge p_2 \wedge \overline{p_3} \wedge \overline{a_{t_0}} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\
\mathcal{A}(t_2, t_0) &= \{p_2 := F, p_3 := T, \mathit{Vin} := T, \\
&\quad a_{t_0} := T, c_{t_0} := [0, 0], \\
&\quad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\}
\end{aligned}$$

3.5 Specifying Properties

Properties to be checked are specified using a dense real-time version of CTL, a branching time logic, known as *timed computation tree logic* (TCTL) that has been extended with clock variables. The formal syntax and semantics of TCTL is described in [50]. The syntax is reproduced here as follows:

```
mergeGuardedCommands(C_P, C_S)
   C = C_P
   for each (⟨φ_G, A⟩ ∈ C)
      for each (⟨φ_{G_S}, A_S⟩ ∈ C_S)
         if (φ_{G_S}[A] == φ_{G_S})
            // Assignments do not effect secondary guarded command so do nothing.
            continue;
         else if (φ_{G_S}[A] == false)
            // Assignments disable secondary guarded command so do nothing.
            continue;
         else if (φ_{G_S}[A] == true)
            // Assignments immediately enable secondary guarded command so
            // firing ⟨φ_G, A⟩ should result in immediate firing of ⟨φ_{G_S}, A_S⟩.
            A = A ∪ A_S
         else
            // Assignments did effect the secondary guard so it may be
            // necessary to modify the guarded command or create
            // a new guarded command.
            if (φ_G ∧ φ_{G_S}[A] == false)
               // The secondary guard with assignments applied still does not
               // enable guarded command so do nothing.
               continue;
            end if
            if (φ_G ∧ ¬φ_{G_S}[A]! = false)
               // Is it possible for the primary guarded command and the
               // inverse of the secondary guarded command to be satisfied
               // simultaneously?    If so, there are two cases.
               φ'_G = φ_G ∧ ¬φ_{G_S}[A]
               A' = A
               C = C ∪ ⟨φ'_G, A'⟩
            else
               φ_G = φ_G ∧ φ_{G_S}[A]
               A = A ∪ A_S
            end if
         end if
      end for
   end for
   for each (⟨φ_{G_S}, A_S⟩ ∈ C_S)
      if (φ_{G_S} contains separation predicates)
         // Time elapse could enabled these guarded commands so they must
         // be included.
         C = C ∪ ⟨enforceThreshold(φ_{G_S}), A_S⟩
      end if
   end for
   return C
end
```

Figure 3.5. Algorithm for merging the primary and secondary guarded commands.

```
enforceThreshold(φ'_GS)
    φ'_GS = true
    for each (product term p ∈ φ_GS)
        for each (inequality i ∈ p in the form v ≥ c)
            φ = p.Cofactor(i)
            φ = φ ∧ (v ≥ c) ∧ (v ≤ c) ∧ incr(v)
            φ'_GS = φ'_GS ∨ φ
        end for
    end for
    for each (product term p ∈ φ_GS)
        for each (inequality i ∈ p in the form v ≤ c)
            φ = p.Cofactor(i)
            φ = φ ∧ (v ≥ c) ∧ (v ≤ c) ∧ decr(v)
            φ'_GS = φ'_GS ∨ φ
        end for
    end for
    return φ'_GS
end
```

Figure 3.6. Modifies secondary guarded command's guard to enforce threshold.

$$\varphi \quad ::= \quad \phi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathrm{E}[\varphi_1 \ \mathrm{U} \ \varphi_2] \mid \mathrm{A}[\varphi_1 \ \mathrm{U} \ \varphi_2] \mid z.\varphi$$

In this formulation of TCTL, where ϕ is an HSL formula, a set of specification clocks is created where z is a member and "z." is the *reset quantifier*. Specification clocks do not control the behavior of the system and are used to express timing requirements of a specification. The reset quantifier causes z to be assigned to zero. The operations $\mathrm{E}[\varphi_1 \ \mathrm{U} \ \varphi_2]$ and $\mathrm{A}[\varphi_1 \ \mathrm{U} \ \varphi_2]$ represent the notions of *possibly* and *inevitably*, respectively. Intuitively, $\mathrm{E}[\varphi_1 \ \mathrm{U} \ \varphi_2]$ holds in a state if φ_2 becomes true on some path from that state and φ_1 is true until φ_2 becomes true. Similary, $\mathrm{A}[\varphi_1 \ \mathrm{U} \ \varphi_2]$ holds in a state if φ_2 becomes true on every path from that state and φ_1 is true until φ_2 becomes true. In addition to the operators included in the formal syntax, additional arithmetic, Boolean, and temporal operators can be defined in terms of the formal syntax. For example, the temporal operators EF φ and AG φ are equivalent to $\mathrm{E}[\mathbf{true} \ \mathrm{U} \ \varphi]$ and $\neg(\mathrm{EF} \ \neg\varphi)$, respectively, and the temporal operators AF φ and EG φ are equivalent to $\mathrm{A}[\mathbf{true} \ \mathrm{U} \ \varphi]$ and $\neg(\mathrm{AF} \ \neg\varphi)$, respectively.

Revisiting the integrator example, the property to specify that a state where *fail* is never reached can be specified in TCTL as $\phi_{init} \implies \mathrm{AG}(\neg \mathit{fail})$. This property is automatically generated from the **assert** statement in the VHDL-AMS code. More complex properties can be manually provided by the user, if desired.

The model checking algorithm proceeds over the structure of $\mathbf{T}\mu$ calculus formula, a language for expressing properties of systems using least and greatest fixpoint operators.

This property of the specification language is important because several symbolic model checking methods (including the BDD based approach described in Chapter 4) do not allow for manipulation of individual states as is required for temporal logic specifications. Therefore, the TCTL property is first translated into a $\mathbf{T}\mu$ formula. $\mathbf{T}\mu$ calculus has the following grammar, as defined in [50]:

$$\varphi ::= Y \mid \phi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \triangleright \varphi_2 \mid z.\varphi \mid \mu Y.\varphi \mid \nu Y.\varphi$$

where ϕ is an HSL formula, z is a *specification clock variable*, and Y is a *formula variable* used in fixpoint computation. The next operator "$\triangleright$" means that φ_1 is true as time elapses until a discrete transition is taken resulting in φ_2. When the specification clock variable z is assigned to zero in φ, $z.\varphi$ is true. The expressions $\mu Y.\varphi$ and $\nu Y.\varphi$ are the least fixpoint and greatest fixpoint, respectively, of φ with the formula variable Y bound inside φ.

The methods for specifying all TCTL requirements as fixpoints of $\mathbf{T}\mu$ are described in detail in [50] and are briefly included here for completeness. First, each subformula of the form A[φ_1 U φ_2] is replaced with the formula $\mu Y.(\varphi_2 \vee \neg z.(\mathrm{E}[(\neg Y)\ \mathrm{U}\ (\neg(\varphi_1 \vee Y) \vee z > c)]))$ where c is any integer constant greater than zero. Note that the value of c may impact the number of iterations performed by the fixpoint computation in the model checking algorithm. Next, each subformula of the form E[φ_1 U φ_2] is replaced with the formula $\mu Y.(\varphi_2 \vee (\varphi_1 \triangleright Y))$.

Given this translation process, the property for the integrator in Figure 2.5 is transformed into the following $\mathbf{T}\mu$ formula:

$$\phi_{init} \Longrightarrow \neg\mu Y.[fail \vee (\mathbf{true} \triangleright Y)]$$

where ϕ_{init} is the initial set of states:

$$\begin{aligned} \phi_{init} \quad = \quad & p_0\, \overline{p_1}\, p_2\, \overline{p_3}\, p_4\, \overline{Vin}\, \overline{Fail}\, \overline{\dot{V}\, out_{[-22,-18]}}\ \dot{V}\, out_{[18,22]} \\ & \overline{a_{t_0}}\, \overline{a_{t_1}}\, a_{t_2}\, \overline{a_{t_3}}\ \overline{a_{t_4}} \wedge c_{t_2} = 0 \wedge Vout = -1000 \end{aligned}$$

If a state in which *fail* is true cannot be reached from the initial state then the formula evaluates to true.

CHAPTER 4

BDD BASED MODEL CHECKER

Henzinger et al. developed a model checking algorithm for timed automata which proceeds recursively over the structure of $\mathbf{T}\mu$ calculus formulas in [50]. This algorithm is shown in Figure 4.1. Basic Boolean operations are performed by recursively applying the algorithm to each portion of the $\mathbf{T}\mu$ calculus formula and then performing the necessary Boolean operation on the results. The base case is when an HSL formula is reached. In this case, the model's invariant is applied to remove obviously unreachable behavior. The next portion of the algorithm is $\triangleright$, the next-state calculation. There are two central components used when calculating the next-state: the *weakest precondition* calculation (*pre*) which is responsible for determining which states could have been reached by firing discrete transitions, and the *time elapse* calculation ($\leadsto$) which evolves continuous variables. The clock specification portion of the algorithm ($z.\varphi$) is performed by evaluating φ and then setting the clock variable z to zero. The final portion of

$$
\begin{array}{rcl}
|\phi| & := & \phi_{\mathcal{I}} \wedge \phi \\
|\neg\varphi| & := & \phi_{\mathcal{I}} \wedge \neg|\varphi| \\
|\varphi_1 \vee \varphi_2| & := & |\varphi_1| \vee |\varphi_2| \\
|\varphi_1 \triangleright \varphi_2| & := & |(|\varphi_1| \vee |\varphi_2|) \leadsto pre(|\varphi_2|)| \\
|z.\varphi| & := & |\varphi|[z := 0] \\
|\mu Y.\varphi| & := & \text{the result of the following iteration:} \\
 & & \phi_{new} := \mathbf{false} \\
 & & \mathbf{repeat} \\
 & & \quad \phi_{old} := \phi_{new} \\
 & & \quad \phi_{new} := |\varphi[Y := \phi_{old}]| \\
 & & \mathbf{until}\ (\phi_{new} \Longrightarrow \phi_{old}) \\
 & & \mathbf{return}\ \phi_{old}
\end{array}
$$

Figure 4.1. Symbolic analysis algorithm for $\mathbf{T}\mu$ calculus (courtesy of [50]).

the algorithm is the fixpoint computation which iteratively subsitutes the result of the previous iteration for the fixpoint variable Y until the result no longer changes.

Upon termination of the algorithm, the resulting HSL formula is equivalent to the invariant $\phi_{\mathcal{I}}$ if the system does not violate the property and the model is non-Zeno [1] (i.e., that the model never reaches a state where time can progress no further). The models discussed in this dissertation can be verified to be non-Zeno for a specified amount of time by testing if a transition that is forced to fire after a specified amount of time fires. Additionally, it must be noted to avoid confusion that this algorithm performs the state exploration in a backwards fashion beginning from a set of error states as shown in Figure 4.2. Beginning from the error states, a state space exploration is performed in a backwards fashion. If it is found that the initial state intersects the found states, an error state can be reached from the initial state meaning that the property has been violated.

The core algorithm in Figure 4.1 was later applied to the verification of embedded system by Alur et al. in [8]. However, in order to support the linear hybrid automata model, a more general model than the timed automata used by Henzinger et al., the symbolic model is adapted to support continuous variables that can change at rates other than one, and the time elapse calculation is enhanced to support the ability for continuous variables to change at rates other than one.

One interesting recent development is the use of this algorithm to model check timed automata using *Binary Decision Diagrams* (BDDs) [3] as the state space representation by Seshia et al. [65]. A BDD is a directed graph representation of a Boolean function. The

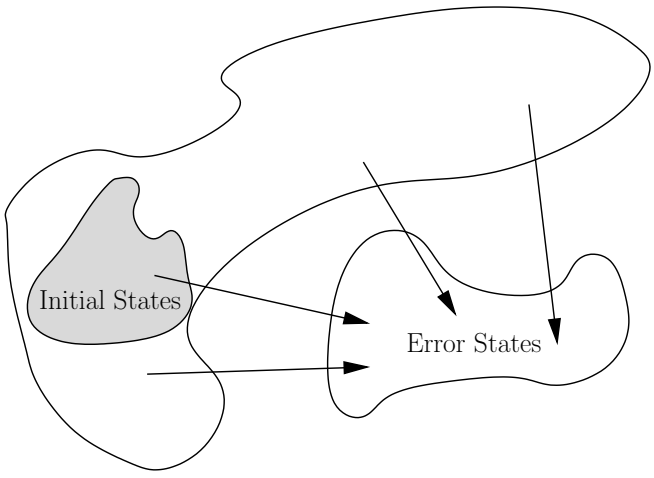


Figure 4.2. Backwards model checking.

graph consists of decision nodes with two outgoing edges and two terminals representing **true** and **false**. Each decision node is labeled with a particular Boolean variable and the edges represent assignments of **true** or **false** to that variable. Most current BDD implementations place an ordering on the variables and apply basic reduction rules to the graph. Therefore, the term BDD normally refers to *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [22]. These two additions resulted in an efficient data structure and algorithms to perform Boolean operations on this data structure that are frequently used in digital logic synthesis and verification tools, among others. Significant research has also been performed into how the variable ordering impacts the size of a Boolean function's BDD representation. While this research could potentially be utilized to improve performance of the methods described in this chapter, significant effort has not been devoted to investigating that potential.

In [65], Seshia and Bryant describe a Boolean symbolic model checking procedure for real-time systems where separation predicates are mapped to Boolean variables and real variables can only change at a rate of one. The analysis proceeds using Boolean operations with the regular addition of Boolean variables representing transitivity relations. This dissertation extends these works by using the ideas presented in [65] to use Boolean methods for the verification of more general hybrid systems where real variables can change at any rate within a range allowing for modeling and verification of analog and mixed-signal circuits. In order to achieve this goal, restrictions are first placed upon separation predicates present in HSL formulas to ensure a canonical form. These canonical separation predicates are then mapped to BDD variables, resulting in a purely Boolean HSL representation. Several BDD packages including BuDDy [28] and CUDD [68] are available for the implementation of this algorithm. BDD packages typically offer similar functionality; therefore the main differentiating factors are performance, implementation language, platform availability, and stability. Due to its wide spread use, competitive performance, and maturity, the CUDD library is used to implement this chapter's algorithms. Due to the Boolean representation of HSL formulas, the algorithm of Figure 4.1 requires several modifications. For example, it is necessary to insert constraints relating Boolean variables that map to separation predicates. This is necessary to reduce redundancy and eliminate contradictions among separation predicates. Constraint construction is a very expensive operation; therefore constraints are not formed as frequently as is necessary to ensure exactness of the algorithm. This results in a model checking algorithm that while

conservative in nature has dramatically improved performance. The remainder of this chapter describes each component of the BDD based model checking algorithm in detail.

The overall algorithm for performing BDD based model checking of LHPNs, which is very similar to the original model checking algorithm presented in Figure 4.1, is shown in Figure 4.3. This algorithm proceeds recursively over the structure of φ, a $\mathbf{T}\mu$ property, given the symbolic model for the system to be verified. The symbolic model consists of the initial state (ϕ_{init}), the invariant ($\phi_{\mathcal{I}}$), the guarded command set ($\mathcal{C}$), and the possible rate sets ($\mathcal{R}$). When φ is an HSL formula, the system invariant is applied to it to remove portions of the state space that are certain to be unreachable. The cases for handling negation and disjunction in the $\mathbf{T}\mu$ formula are performed by recursively applying `bddCheck` to the subformulas and then performing the necessary action on the results. In the least fixpoint calculation, Y is a formula variable bound inside the portion of the $\mathbf{T}\mu$ formula φ. The fixpoint proceeds by substituting Y for ϕ_{old} which starts out as **false**. The `bddCheck` algorithm is applied to the substituted form of φ.

```
// φ_init - Global variable representing initial state of LHPN (Boolean HSL formula)
// φ_I - Global variable representing LHPN invariant (Boolean HSL formula)
// C - Global variable representing LHPN guarded command set
// R - Global variable representing LHPN possible rate set
bddCheck(φ)
   // φ - Tμ property under verification
   switch type of φ's expression
   case φ
      return φ_I ∧ φ
   case ¬φ
      return φ_I ∧ ¬bddCheck(φ)
   case φ1 ∨ φ2
      return bddCheck(φ1) ∨ bddCheck(φ2)
   case φ1 ▷ φ2
      φ1 = bddCheck(φ1) ∨ bddCheck(φ2)
      φ2 = pre(bddCheck(φ2), φ_I, C)
      return bddCheck(timeElapse(φ1, φ2, φ_I, R))
   case z.φ
      φ1 = bddCheck(φ)
      return specifyClock(z, φ1)
   case μY.φ
      φ_new = false
      do
         φ_old = φ_new
         φ_new = simplifyRestrict(bddCheck(substitute(φ, Y, φ_old)))
      while (¬simplifyRestrict(checkImplication(φ_new, φ_old)))
      return φ_old
   end switch
```

Figure 4.3. BDD based model checking algorithm (adapted from [65]).

This chapter is devoted to first describing the method by which HSL is mapped to a Boolean representation and the issues associated with this representation. Specifically, specialized algorithms are used to attempt to reduce the current size of the state representation and to determine if one HSL formula implies another. These algorithms are referred to as `simplifyRestrict` and `checkImplication`, respectively. Given the Boolean representation, the core components of the model checking algorithm: clock specification (`specifyClock`), transition precondition (`pre`), and progression of time (`timeElapse`), are described. Additionally, approaches for performing an optimized time elapse and for generating error traces are presented.

4.1 Representing HSL Formulas Using BDDs

The verification algorithm relies on performing Boolean operations using BDDs. Thus, it is necessary to efficiently represent HSL formulas as Boolean formulas. The details of this Boolean representation are presented in this section. First, a method for generating canonical separation predicates, which are then mapped to BDD variables, is described. Given this Boolean representation, it is necessary to periodically construct Boolean constraints among the separation predicates. Additionally, the `simplifyRestrict` and `checkImplication` algorithms are detailed.

4.1.1 Canonical Separation Predicates

An approach for generating canonical separation predicates which is similar to that suggested for octagonal polyhedra in [56] is described in this section. A canonical representation is necessary so that $2x_1 \geq x_2 + 1$ maps to the same BDD variable as the equivalent separation predicate with different coefficients $4x_1 \geq 2x_2 + 2$. The canonical representation is of the form $c_1x_1 \geq c_2x_2 + c_3$ with the following restrictions where x_0 is a special variable representing zero:

- The continuous variables x_1 and x_2 are distinct.
- If $x_1 = x_0$ or $x_2 = x_0$ then the corresponding constant c_1 or c_2 is one.
- The constants c_1 and c_2 are not both negative.
- If c_1 or c_2 is negative (but not both), then in the ordered set of real variables, x_1 comes before x_2.

- The constant c_1 has an absolute value of one, and the constants c_2 and c_3 are rational numbers using arbitrarily large integers as the numerator and the denominator.

Using the special variable x_0 and the fact that separation predicates of the form $c_1x_1 > c_2x_2 + c_3$ are equivalent to $\overline{c_2x_2 \geq c_1x_1 + -c_3}$, the above form with the restrictions can represent any linear separation predicate between two real variables in a unique way. Table 4.1 shows the calculations for constructing the canonical form of a separation predicate. It is assumed that the separation predicate is provided in the form $c_1x_1 \geq c_2x_2 + c_3$ and that it is not trivially true or false. For example $x \geq x + 2$ is trivially false. Note that if both real variables x_1 and x_2 are x_0, the separation predicate is always trivially true or false so Table 4.1 does not account for this case. The table also assumes that separation predicates where x_1 equals x_2 are first rewritten as separation predicates in terms of x_0.

For each canonical separation predicate that is generated, a corresponding BDD variable is created. Creation of this mapping of BDD variables to separation predicates allows for an entirely Boolean representation of the state space. For clarity, throughout the remainder of this chapter, separation predicates are shown; however those separation predicates are actually being mapped to BDD variables representing the canonical form of that separation predicate.

Table 4.1. Constructing canonical separation predicates. Given a separation predicate of the form $c_1x_1 \geq c_2x_2 + c_3$, this table shows the calculation, which results in the equivalent canonical separation predicate.

	$x_1 = x_0$ $x_2 \neq x_0$	$x_1 \neq x_0$ $x_2 = x_0$	$x_1 \neq x_0 \quad x_2 \neq x_0$	
			$x_1 < x_2$	$x_1 > x_2$
$c_1 > 0$ $c_2 < 0$	$x_2 \geq x_0 + \frac{c_3}{\lvert c_2 \rvert}$	$x_1 \geq x_0 + \frac{c_3}{c_1}$	$x_1 \geq \frac{c_2}{c_1}x_2 + \frac{c_3}{c_1}$	$x_2 \geq \frac{-c_1}{\lvert c_2 \rvert}x_1 + \frac{c_3}{\lvert c_2 \rvert}$
$c_1 < 0$ $c_2 > 0$	$x_0 \geq x_2 + \frac{c_3}{c_2}$	$x_0 \geq x_1 + \frac{c_3}{\lvert c_1 \rvert}$	$-x_1 \geq \frac{c_2}{\lvert c_1 \rvert}x_2 + \frac{c_3}{\lvert c_1 \rvert}$	$-x_2 \geq \frac{-c_1}{c_2}x_1 + \frac{c_3}{c_1}$
$c_1 > 0$ $c_2 > 0$	$x_0 \geq x_2 + \frac{c_3}{c_2}$	$x_1 \geq x_0 + \frac{c_3}{c_1}$	$x_1 \geq \frac{c_2}{c_1}x_2 + \frac{c_3}{c_1}$	
$c_1 < 0$ $c_2 < 0$	$x_2 \geq x_0 + \frac{c_3}{\lvert c_2 \rvert}$	$x_0 \geq x_1 + \frac{c_3}{\lvert c_1 \rvert}$	$x_2 \geq \frac{-c_1}{\lvert c_2 \rvert}x_1 + \frac{c_3}{\lvert c_2 \rvert}$	

4.1.2 Constraint Generation

As operations are performed on the state space and new separation predicates are inserted into the representation, it is necessary to form additional Boolean relationships among the new separation predicates and those that already exist. These new relationships are referred to as *constraints*. This process is necessary for several reasons. First, sets of separation predicates that are infeasible can be eliminated. For example, a portion of the state space representation asserting $x \geq 2$ and $x \leq 0$ can be eliminated. Second, when variables are removed from the state representation, it is necessary to ensure that information is not lost. For example, consider a situation where the separation predicates $x \geq y$ and $y \geq z$ are present in the state, but the continuous variable y is going to be abstracted away. Abstracting y would result in the removal of these two separation predicates and the loss of the implicit knowledge that $x \geq z$. Third, application of constraints may result in tighter relationships among variables. For example, if the inequalities $x \geq y+2$, $y \geq z+3$, and $x \geq z+6$ are present in the state representation, the first two inequalities can be used to generate a tighter constraint stating that $x \geq z+5$. This is again particularly important when a continuous variable is removed from the state.

Two general types of constraints are constructed: transitivity constraints and implications constraints. As new separation predicates are generated and mapped to Boolean variables, constraints are added to create relationships with existing variables. The first type of constraint creates a transitivity relation between two separation predicates that share a real variable and a third, newly created separation predicate. For example,

$$\begin{aligned} 2x \geq 3y + 5 \wedge 4y \geq 5z + 5 \quad &\Rightarrow \\ \frac{2}{3}x \geq y + \frac{5}{3} \wedge y \geq \frac{5}{4}z + \frac{5}{4} \quad &\Rightarrow \quad \frac{2}{3}x \geq \frac{5}{4}z + \frac{5}{3} + \frac{5}{4} \\ &\Rightarrow \quad 8x \geq 15z + 35 \end{aligned}$$

The second type of constraints are created between pairs of separation predicates where one separation predicate implies the other. For example, $2x \geq 3y + 5 \Rightarrow 2x \geq 3y + 4$ and $2x > 3y + 5 \Rightarrow 2x \geq 3y + 5$. Note that during constraint generation, the normal and inverted forms of the separation predicates must be considered.

The algorithm for generating all constraints for a specified variable x and HSL expression ϕ represented as a BDD is shown in Figure 4.4. A similar algorithm that generates all the constraints and applies them to the initial BDD is shown in Fig-

ure 4.5. These algorithms operate by iterating through each pair of BDD variables in the support of ϕ that map two separation predicates containing x, the real variable over which constraints are being generated. It then determines if implication constraints and transitivity constraints exist between those two separation predicates. The methods for finding these constraints are discussed later in this section. The algorithm in Figure 4.4 returns a conjunction of the newly formed constraints. The algorithm in Figure 4.5 returns a conjunction of ϕ, the BDD that is passed into the function, and the new constraints. These algorithms require a real variable to be specified as the variable over which constraints should be generated because constraints are generally applied before removing potentially important separation predicates over specific variables. By only adding constraints over real variables that are being removed, the goal is to reduce the number of new separation predicates and thus the number of BDD variables that get

```
getConstraints(φ, x)
   // Iterate through all pairs of separation predicates containing x and form
   // conjunction of all constraints.
   for each BDD variable φ_i ∈ φ mapping to separation predicate containing x
      for each BDD variable φ_j ∈ φ mapping to separation predicate containing x
         if (φ_i == φ_j) continue
         φ_cons = φ_cons ∧ implies(φ_i, φ_j)
         φ_cons = φ_cons ∧ trans(φ_i, φ_j, x)
      end for
   end for
   return φ_cons
```

Figure 4.4. Finds constraints for a given real variable x over ϕ.

```
addConstraints(φ, x)
   // Iterate through each discrete marking and generate transitivity
   // constraints for that portion of φ
   for each product term φ_i ∈ Φ
      φ_part = φ ∧ φ_i
      for each BDD variable φ_j ∈ φ_part mapping to separation predicate containing x
         for each BDD variable φ_k ∈ φ_part mapping to separation predicate containing x
            if (φ_j == φ_k) continue
            φ_part = φ_part ∧ implies(φ_j, φ_k)
            φ_part = φ_part ∧ trans(φ_j, φ_k, x)
         end for
      end for
      φ_result = φ_result ∨ φ_part
   end for
   return φ_result
```

Figure 4.5. Adds constraints for a given real variable x to ϕ.

built.

The algorithm for finding implication constraints between two separation predicates, ϕ_1 and ϕ_2, is shown in Figure 4.6. This algorithm makes use of four additional functions shown in Figure 4.7. When forming implication constraints, the real variables and slopes of each separation predicate must match. The function `timpliest` considers the two separation predicates in their true form where both separation predicates are "$\geq$" inequalities and `fimpliesf` considers the two separation predicates in their false form where both separation predicates are "$<$" inequalities. The final two algorithms, `timpliesf` and `fimpliest`, handle the situations when one or the other separation predicate is inverted and is thus a "$<$" inequality, and the other is in the noninverted, "$\geq$" inequality. In this case, it is slightly more complicated to determine if an implication relationship exists due to the canonical form of the separation predicates. Specifically, the possibility of a negative constant must be considered. For example, an implication constraint can be formed between the two inequalities $-2x_1 \geq 3x_2 + 4$ and $2x_1 \geq -3x_2 + -3$ by inverting the second constraint and multiplying it through by -1. This results in the implication constraint $-2x_1 \geq 3x_2 + 4 \Rightarrow -2x_1 > 3x_2 + 3$.

The algorithm for finding transitivity constraints between two separation predicates, ϕ_1 and ϕ_2, with the real variable x as the variable over which transitivity is formed, referred to as the *pivot* variable, is shown in Figure 4.8. However, transitivity constraints can also be generated given two arbitrary separation predicates without specifying a pivot value as shown in Tables 4.2 and 4.3. A tabular form of the conditions under which a transitivity constraint can be constructed between two true separation predicates

```
implies(φ1 , φ2)
    if (timpliest(φ1 , φ2))
        φresult = φresult ∧ (¬φ1 ∨ φ2)
    end if
    if (fimpliesf(φ1 , φ2))
        φresult = φresult ∧ (φ1 ∨ ¬φ2)
    end if
    if (timpliesf(φ1 , φ2))
        φresult = φresult ∧ (¬φ1 ∨ ¬φ2)
    end if
    if (fimpliest(φ1 , φ2))
        φresult = φresult ∧ (φ1 ∨ φ2)
    end if
    return φresult
```

Figure 4.6. Finds implication constraints among ϕ_1 and ϕ_2.

timpliest(ϕ_1, ϕ_2)
  ϕ_1 maps to the separation predicate $c_1x_1 \geq c_2x_2 + c_3$
  ϕ_2 maps to the separation predicate $c_4x_4 \geq c_5x_5 + c_6$
  if $(x_1 == x_4 \wedge x_2 == x_5 \wedge c_1 == c_4 \wedge c_2 == c_5 \wedge c_3 > c_6)$
    return true
  end if
  return false

fimpliesf(ϕ_1, ϕ_2)
  return timplifest(ϕ_2, ϕ_1);

timpliesf(ϕ_1, ϕ_2)
  ϕ_1 maps to the separation predicate $c_1x_1 \geq c_2x_2 + c_3$
  ϕ_2 maps to the separation predicate $c_4x_4 \geq c_5x_5 + c_6$
  if $(c_1 > 0 \wedge c_2 > 0 \wedge c_4 > 0 \wedge c_5 > 0)$
    if $(x_1 == x_5 \wedge x_2 == x_4 \wedge \frac{c_2}{c_1} == \frac{c_4}{c_5} \wedge \frac{c_3}{c_1} > -\frac{c_6}{c_5})$
      return true
    end if
  else if $((c_1 < 0 \wedge c_2 > 0 \wedge c_4 > 0 \wedge c_5 < 0) \vee (c_1 > 0 \wedge c_2 < 0 \wedge c_4 < 0 \wedge c_5 > 0))$
    if $(x_1 == x_4 \wedge x_2 == x_5 \wedge \frac{c_1}{c_2} == \frac{c_4}{c_5} \wedge c_3 > -c_6)$
      return true
    end if
  end if
  return false

fimpliest(ϕ_1, ϕ_2)
  ϕ_1 maps to the separation predicate $c_1x_1 \geq c_2x_2 + c_3$
  ϕ_2 maps to the separation predicate $c_4x_4 \geq c_5x_5 + c_6$
  if $(c_1 > 0 \wedge c_2 > 0 \wedge c_4 > 0 \wedge c_5 > 0)$
    if $(x_1 == x_5 \wedge x_2 == x_4 \wedge \frac{c_1}{c_2} == \frac{c_5}{c_4} \wedge -\frac{c_3}{c_2} > \frac{c_6}{c_4})$
      return true
    end if
  else if $((c_1 < 0 \wedge c_2 > 0 \wedge c_4 > 0 \wedge c_5 < 0) \vee (c_1 > 0 \wedge c_2 < 0 \wedge c_4 < 0 \wedge c_5 > 0))$
    if $(x_1 == x_4 \wedge x_2 == x_5 \wedge \frac{c_1}{c_2} == \frac{c_4}{c_5} \wedge -c_3 > c_6)$
      return true
    end if
  end if
  return false

Figure 4.7. Algorithms to determine if ϕ_1 implies ϕ_2 in their regular and negated forms.

```
trans(φ1 , φ2 , x)
   φ1 maps to the separation predicate c1x1 ≥ c2x2 + c3
   φ2 maps to the separation predicate c4x4 ≥ c5x5 + c6
   // Is first separation predicate in true or false form for transitivity relation?
   if (x1 == x ∧ c1 > 0)
      xi = x2; ci = c2/c1; c = −c3/c1
      p1inverted = true
   else if (x1 == x ∧ c1 < 0)
      xi = x2; ci = c2/c1; c = −c3/c1
      p1inverted = false
   else if (x2 == x ∧ c2 > 0)
      xi = x1; ci = c1/c2; c = c3/c2
      p1inverted = false
   else if (x2 == x ∧ c2 < 0)
      xi = x1; ci = c1/c2; c = c3/c2
      p1inverted = true
   else
      return true
   end if
   // Is second separation predicate in true or false form for transitivity relation?
   if (x4 == x ∧ c4 > 0)
      xj = x5; cj = c5/c4; c = c + c6/c4
      p2inverted = false
   else if (x4 == x ∧ c4 < 0)
      xj = x5; cj = c5/c4; c = c + c6/c4
      p2inverted = true
   else if (x4 == x ∧ c4 > 0)
      xj = x4; cj = c4/c5; c = c + −c6/c5
      p2inverted = true
   else if (x5 == x ∧ c5 < 0)
      xj = x4; cj = c4/c5; c = c + −c6/c5
      p2inverted = false
   else
      return true
   end if
```

Figure 4.8. Finds transitivity constraints for ϕ_1 and ϕ_2 over a real variable.

```
// Now form transitivity relation(s)
φ_new1 maps to the separation predicate c_i x_i ≥ c_j x_j + c
φ_new2 maps to the separation predicate c_j x_j ≥ c_i x_i + −c
if (¬p1inverted ∧ ¬p2inverted ∧ φ_new1 ∈ φ)
    φ_cons = φ_cons ∧ (¬(φ_1 ∧ φ_2) ∨ φ_new1)
else if (p1inverted ∧ p2inverted ∧ φ_new2 ∈ φ)
    φ_cons = φ_cons ∧ (¬(¬φ_1 ∧ ¬φ_2) ∨ ¬φ_new2)
else if (p1inverted ∧ ¬p2inverted)
    if (φ_new1 ∈ φ)
        φ_cons = φ_cons ∧ (¬(¬φ_1 ∧ φ_2) ∨ ¬φ_new1)
    end if
    if (φ_new2 ∈ φ)
        φ_cons = φ_cons ∧ (¬(¬φ_1 ∧ φ_2) ∨ ¬φ_new2)
    end if
else if (¬p1inverted ∧ p2inverted)
    if (φ_new1 ∈ φ)
        φ_cons = φ_cons ∧ (¬(φ_1 ∧ ¬φ_2) ∨ ¬φ_new1)
    end if
    if (φ_new2 ∈ φ)
        φ_cons = φ_cons ∧ (¬(φ_1 ∧ ¬φ_2) ∨ ¬φ_new2)
    end if
end if
return φ_cons
```

Figure 4.8 continued.

Table 4.2. Constructing transitivity constraints between pairs of separation predicates. The variables e_1 and e_2 correspond to $c_1x_1 \geq c_2x_2 + c_3$ and $c_4x_4 \geq c_5x_5 + c_6$, respectively.

		$e_1 \wedge e_2$	$\neg e_1 \wedge \neg e_2$
$x_2 = x_4$	$c_2 > 0, c_4 > 0$	$\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})$	$\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})}$
	$c_2 < 0, c_4 < 0$	$\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})$	$\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})}$
	$c_2 > 0, c_4 < 0$	No Relation	No Relation
	$c_2 < 0, c_4 > 0$	No Relation	No Relation
$x_1 = x_5$	$c_1 > 0, c_5 > 0$	$\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})$	$\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})}$
	$c_1 < 0, c_5 < 0$	$\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})$	$\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})}$
	$c_1 > 0, c_5 < 0$	No Relation	No Relation
	$c_1 < 0, c_5 > 0$	No Relation	No Relation
$x_1 = x_4$	$c_1 > 0, c_4 > 0$	No Relation	No Relation
	$c_1 < 0, c_4 < 0$	No Relation	No Relation
	$c_1 > 0, c_4 < 0$	$\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})$	$\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})}$
	$c_1 < 0, c_4 > 0$	$\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})$	$\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})}$
$x_2 = x_5$	$c_2 > 0, c_5 > 0$	No Relation	No Relation
	$c_2 < 0, c_5 < 0$	No Relation	No Relation
	$c_2 > 0, c_5 < 0$	$\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})$	$\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})}$
	$c_2 < 0, c_5 > 0$	$\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})$	$\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})}$

Table 4.3. Constructing transitivity constraints between pairs of separation predicates. The variables e_1 and e_2 correspond to $c_1x_1 \geq c_2x_2 + c_3$ and $c_4x_4 \geq c_5x_5 + c_6$, respectively.

		$\neg e_1 \wedge e_2$	$e_1 \wedge \neg e_2$
$x_2 = x_4$	$c_2 > 0, c_4 > 0$	No Relation	No Relation
	$c_2 < 0, c_4 < 0$	No Relation	No Relation
	$c_2 > 0, c_4 < 0$	$\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})$ $\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})}$	$\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})$ $\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})}$
	$c_2 < 0, c_4 > 0$	$\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})$ $\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})}$	$\frac{c_5}{c_4}x_5 \geq \frac{c_1}{c_2}x_1 + (\frac{-c_6}{c_4} + \frac{-c_3}{c_2})$ $\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_5}{c_4}x_5 + (\frac{c_3}{c_2} + \frac{c_6}{c_4})}$
$x_1 = x_5$	$c_1 > 0, c_5 > 0$	No Relation	No Relation
	$c_1 < 0, c_5 < 0$	No Relation	No Relation
	$c_1 > 0, c_5 < 0$	$\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})$ $\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})}$	$\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})$ $\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})}$
	$c_1 < 0, c_5 > 0$	$\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})$ $\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})}$	$\frac{c_2}{c_1}x_2 \geq \frac{c_4}{c_5}x_4 + (\frac{-c_3}{c_1} + \frac{-c_6}{c_5})$ $\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_2}{c_1}x_2 + (\frac{c_6}{c_5} + \frac{c_3}{c_1})}$
$x_1 = x_4$	$c_1 > 0, c_4 > 0$	$\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})$ $\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})}$	$\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})$ $\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})}$
	$c_1 < 0, c_4 < 0$	$\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})$ $\overline{\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})}$	$\frac{c_2}{c_1}x_2 \geq \frac{c_5}{c_4}x_5 + (\frac{-c_3}{c_1} + \frac{c_6}{c_4})$ $\overline{\frac{c_5}{c_4}x_5 \geq \frac{c_2}{c_1}x_2 + (\frac{-c_6}{c_4} + \frac{c_3}{c_1})}$
	$c_1 > 0, c_4 < 0$	No Relation	No Relation
	$c_1 < 0, c_4 > 0$	No Relation	No Relation
$x_2 = x_5$	$c_2 > 0, c_5 > 0$	$\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})$ $\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})}$	$\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})$ $\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})}$
	$c_2 < 0, c_5 < 0$	$\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})$ $\overline{\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})}$	$\frac{c_4}{c_5}x_4 \geq \frac{c_1}{c_2}x_1 + (\frac{c_6}{c_5} + \frac{-c_3}{c_2})$ $\overline{\frac{c_1}{c_2}x_1 \geq \frac{c_4}{c_5}x_4 + (\frac{c_3}{c_2} + \frac{-c_6}{c_5})}$
	$c_2 > 0, c_5 < 0$	No Relation	No Relation
	$c_2 < 0, c_5 > 0$	No Relation	No Relation

and two false separation predicates is shown in Table 4.2. Similarly, a tabular form of the conditions under which a transitivity constraint can be constructed between two separation predicates where one or the other is false is shown in Table 4.3. The table's cells show the new separation predicate that is constructed in the transitivity constraint. As an example, consider the two separation predicates $2x \geq 3y+5$ and $2x \geq 4z+8$. Given the variable x over which to form transitivity constraints, it must first be determined whether each separation predicate should be in the inverted or noninverted form to create the transitivity relationship. In this case, since $2x \geq 3y+5$ has x on the left-hand side of the inequality, the separation predicate must be inverted to get x onto the right-hand size of the inequality. This results in $3y > 2x-5$. Similarly, the second separation predicate is tested to check if it is the correct form to form transitivity constraints. Since x is already on the left-hand side in this case, no modification is necessary. Once it is determined if the separation predicates are in the proper form, the portions of the new separation predicate can be calculated. Note that two separation predicates are created since one inequality contains a "$>$" and the other contains a "$\geq$". After creating the separation predicates, the original BDD is tested to see if they are already used within the BDD. If they are, the transitivity constraints are inserted. Otherwise, they are ignored. In this case, the transitivity constraints that are inserted are $3y > 2x - 5 \wedge 3x \geq 4z + 8 \Rightarrow 9y \geq 12z + 1$ and $3y > 2x - 5 \wedge 3x \geq 4z + 8 \Rightarrow 9y > 12z + 1$.

4.1.3 Reducing BDD Size Using `simplifyRestrict`

The potential size of the state representation grows exponentially with each new BDD variable that is created. Therefore, steps must be taken to try to reduce the size of the BDD state representation as analysis proceeds. One of these steps is the use of the `simplifyRestrict` operation. The algorithm for performing this action is shown in Figure 4.9. This algorithm finds the constraints for each real variable in ϕ and uses the BDD `Restrict` operation to remove paths in ϕ that violate those constraints. Details about the BDD `Restrict` operation are shown in [29]. Briefly, as it traverses the ϕ, it eliminates paths on which ϕ_{cons} is false as long as the resulting BDD is not enlarged. The `simplifyRestrict` operation can be very time consuming, so it is applied only once for each fixpoint iteration in the `bddCheck` algorithm.

```
simplifyRestrict(φ)
   for each (x ∈ {x0 ∪ V ∪ C})
      φcons = getConstraints(φ, x);
      φ = φ.Restrict(φcons)
   end for
   return φ
```

Figure 4.9. Algorithm for performing simplify restrict operation.

4.1.4 Checking Implication

In order to determine if one HSL formula implies a second HSL formula, it is necessary for them to have equivalent support sets. Therefore, transitivity constraints are inserted in terms of both ϕ_1 and ϕ_2 before the implication is determined. The algorithm for performing this operation is shown in Figure 4.10.

4.2 Weakest Precondition

The weakest precondition operation, $pre(\phi)$, calculates all the possible states that could have resulted in ϕ by firing discrete transitions. In particular, for each guarded command, $\langle \phi_G, \mathcal{A} \rangle \in \mathcal{C}$, it first performs the assignments ($\mathcal{A}$) to the current set of states, and then applies the guard (ϕ_G). By taking the disjunction of the result for each guarded command, all possible previous states are determined. Finally, ϕ is disjunctively combined with the result, and $\phi_{\mathcal{I}}$ is conjunctively combined to ensure that impossible states are not introduced into the calculation. This is defined formally below:

```
checkImplication(φ1, φ2)
   // First form transitivity constraints around x0.
   for each product term φi ∈ Φ
      φpart = (φ1 ∨ φ2) ∧ φi
      φcons = getConstraints(φpart, x0)
      φ1 = φ1 ∧ φcons
      φ2 = φ2 ∧ φcons
   end for
   // Now form transitivity constraints around all other real variables.
   for each (x ∈ {V ∪ C})
      for each product term φi ∈ Φ
         φpart = (φ1 ∨ φ2) ∧ φi
         φcons = getConstraints(φpart, x)
         φ1 = φ1 ∧ φcons
         φ2 = φ2 ∧ φcons
      end for
   end for
   return φ1 ∧ ¬φ2
```

Figure 4.10. Algorithm to check an implication relationship.

$$pre(\phi) \doteq \phi_{\mathcal{I}} \wedge (\phi \vee \bigvee_{\langle \phi_G, \mathcal{A} \rangle \in \mathcal{C}} \phi_G \wedge (\phi_{\mathcal{I}} \wedge \phi)[\mathcal{A}])) \tag{4.4}$$

An example of applying a transition precondition step to the integrator example is shown in Figure 4.11. Beginning with the state shown in Figure 4.11b, applying the transition precondition operation in a backwards fashion results in the state shown in Figure 4.11a. In this example, *Vout* is between -200 and 2000 mV and the clock c_{t_2} has a value between 0 and 100. The only transition that could have fired in order to reach this state is transition t_3. Therefore, the value of the clock associated with t_3 must have been at 100. Additionally, upon firing t_3, the value of the clock on t_2 is assigned to zero.

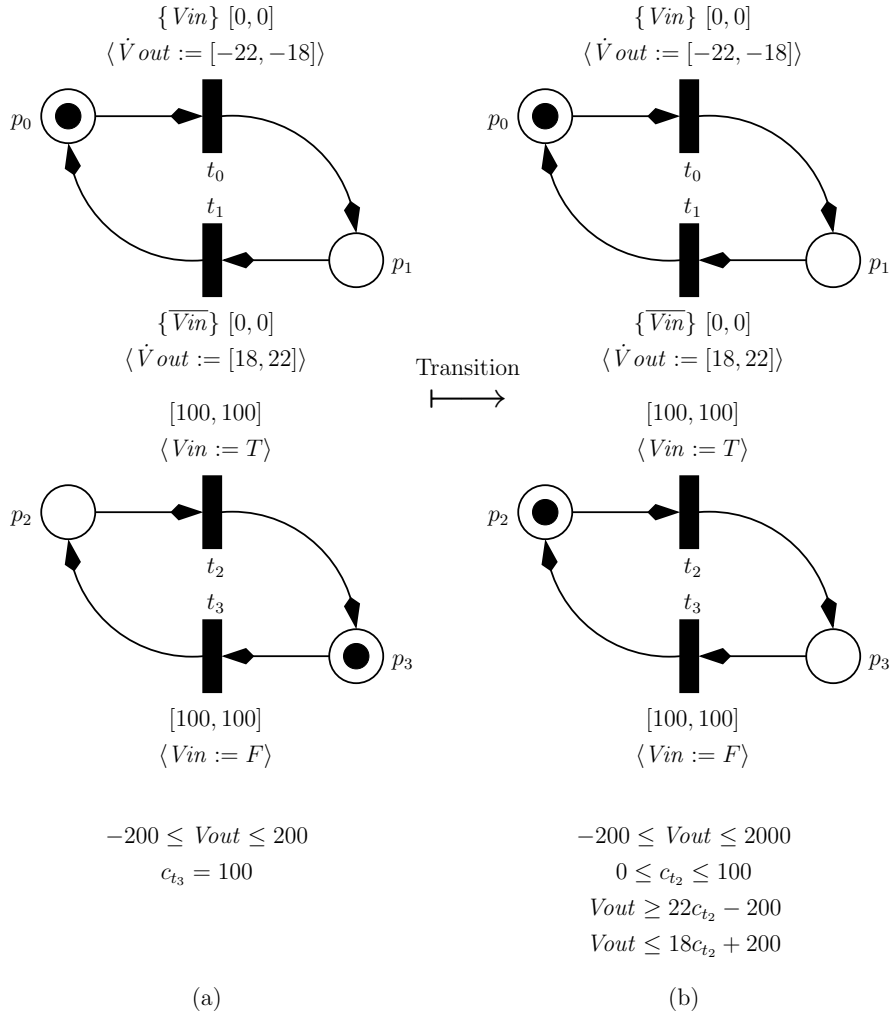


Figure 4.11. Application of weakest precondition operator to the integrator example. (a) Result of applying weakest precondition. (b) Initial state.

Therefore, the relationships between *Vout* and c_{t_2} determine that *Vout* was previously between -200 and 200 as shown in Figure 4.11a.

The algorithm for applying the weakest precondition is shown in Figure 4.12. This algorithm calls `pret` for each guarded command applying the invariant as it goes, and returns the disjunctive combination of applying each guarded command to ϕ. The `pret` algorithm is shown in Figure 4.13. The first step in `pret` is to perform the

```
pre(φ, φ_I, C)
    φ_result = φ
    for each (⟨φ_G, A⟩ ∈ C)
        φ_result = φ_result ∨ pret(φ_I ∧ φ, φ_G, A)
    end for
    return φ_I ∧ φ_result
```

Figure 4.12. Algorithm to perform transition precondition operation.

```
pret(φ, φ_G, A)
    // Perform Boolean assignments by cofactoring the variables.
    for each ((b := true) ∈ A)
        φ = φ.Cofactor(v)
    end for
    for each ((b := false) ∈ A)
        φ = φ.Cofactor(¬v)
    end for
    // Perform [−∞, ∞] assignments by abstracting all
    // separation predicates that contain the variable.
    for each ((v := [−∞, ∞]) ∈ A)
        for each BDD variable φ_v mapping to a separation predicate containing v
        φ = φ.ExistAbstract(φ_v)
    end for
    // Perform remainder of assignments by substituting for new separation
    // predicates where necessary.
    for each BDD variable, φ_i ∈ φ mapping to c1x1 ≥ c2x2 + c3
        if ((x1 := [a1, a1]) ∈ A ∧ (x2 := [a2, a2]) ∈ A)
            φ_sub = x0 ≥ x0 + c3 − a1c1 + a2c2
            φ = φ.Compose(φ_sub, φ_i)
        else if ((x1 := [a1, a1]) ∈ A)
            φ_sub = x0 ≥ c2xc + c3 − a1c1
            φ = φ.Compose(φ_sub, φ_i)
        else if ((x2 := [a2, a2]) ∈ A)
            φ_sub = c1x1 ≥ x0 + c3 − a2c2
            φ = φ.Compose(φ_sub, φ_i)
        end if
    end for
    // Return conjunction guard condition and φ[A].
    return φ ∧ φ_G
```

Figure 4.13. Transition precondition algorithm for a given guarded command.

Boolean variable assignments for the guarded command using the BDD cofactor operation. Next, the $[-\infty, \infty]$ assignments are performed by existentially abstracting all inequalities containing the variable to which the assignment is being made. Finally, the remaining real variable assignments are performed by finding the inequalities containing the necessary real variable and replacing them with a newly created inequality in which the assignment has been made using the BDD compose operation. Note that, this method of performing real variable assignment assumes that a range of values is not being assigned to the variable. This method is used to improve performance; however if ranges of assignments are required, another approach whereby an inequality representing the range of assignments is applied and transitivity constraints are generated could be used. The final step is to apply the guarded command's guard by taking the conjunction of the guard and the result of performing the assignments.

4.3 Time Elapse

The time elapse operation ($\rightsquigarrow$) calculates all the states that can reach ϕ_2 by allowing time to elapse while remaining in ϕ_1 in between. The general idea of time elapse is that the state region ϕ_2 is expanded to include all states that can reach ϕ_2 by moving time backward. The result is then intersected with all the states that can result in ϕ_1 by moving time backward up to the point where $\phi_1 \wedge \phi_2$ is no longer satisfied. Figure 4.14 presents a visual representation of the time elapse operation. Given an initial state region,

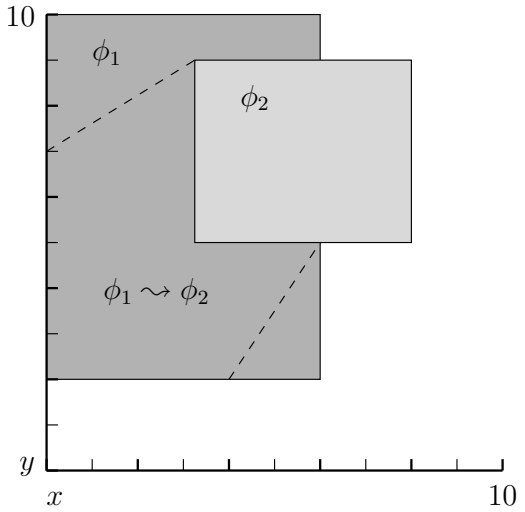


Figure 4.14. Visual representation of $\phi_1 \rightsquigarrow \phi_2$ where $1 \leq \dot{x} \leq 2$ and $1 \leq \dot{y} \leq 2$.

ϕ_2, the result of time elapse encompasses ϕ_2 plus the region within the dotted lines where ϕ_1 is satisfied.

An example of applying a time elapse operation to the integrator example is shown in Figure 4.15. Beginning with the state shown in Figure 4.15b applying the time elapse operation in a backwards fashion results in the state shown in Figure 4.15a. In this example, $Vout$ has a value of 2000 and the clock on transition t_2 has a value of 100. In

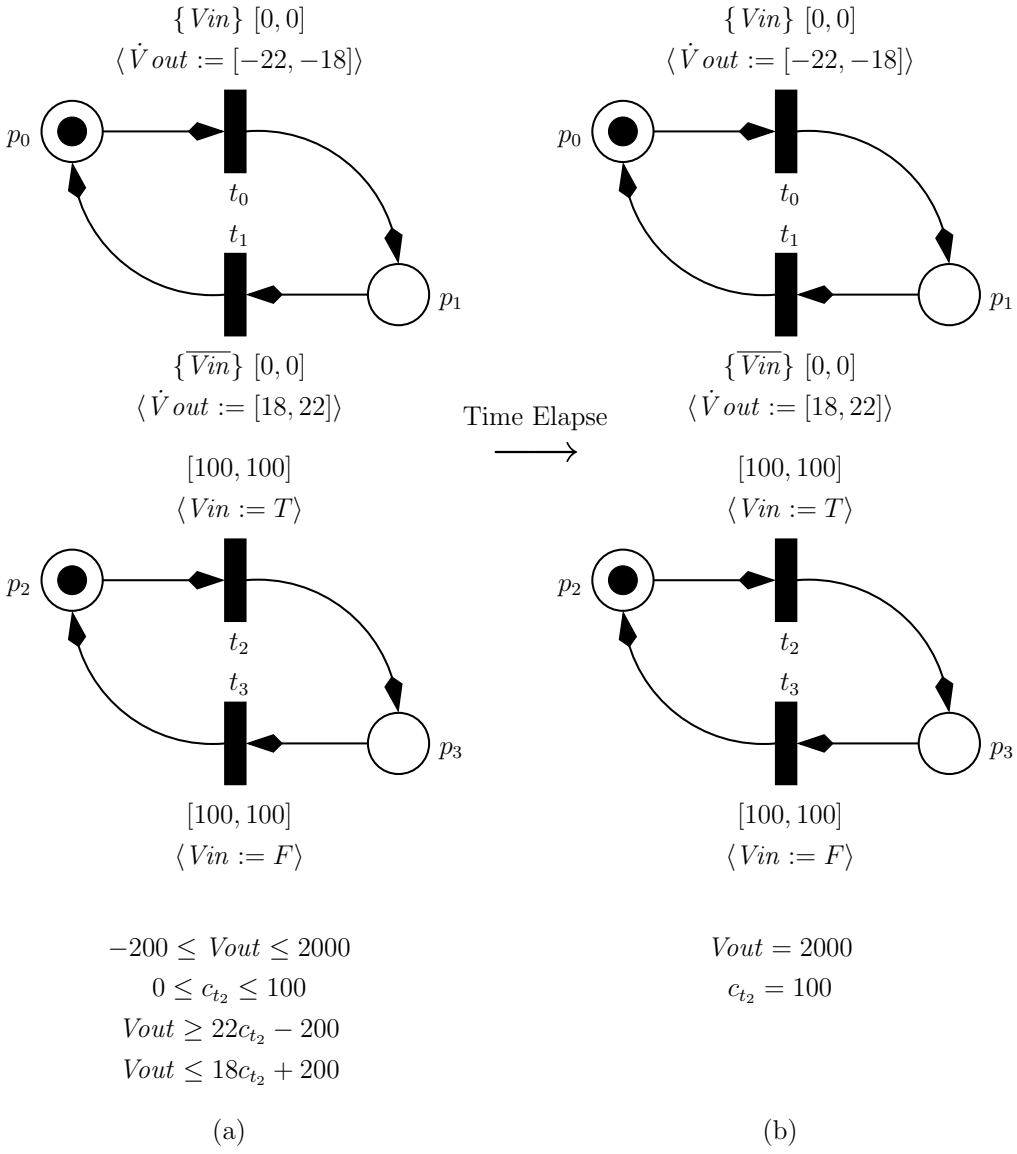


Figure 4.15. Application of time elapse operator to the integrator example. (a) Result of apply time elapse. (b) Initial state.

this state, the only action that can occur is time moving backwards. As time moves backwards, the value of *Vout* can go as low as -200 and the value of the clock on transition t_2 can get down to zero. Since the clock value cannot go below zero, time is restricted from moving backwards any further. During the time elapse calculation, additional relationships between *Vout* and c_{t_2} are formed since c_{t_2} bounds the range of values that *Vout* can have.

Figure 4.16 shows the top level algorithm for the time elapse calculation. Based on the value of ϕ_1, it calls two different versions of the time elapse calculation. The first version consists of the two functions `timeElapsePhi1` and `timeElapsePhi2`. This version is the basic time elapse calculation which works in all cases. However, in situations where ϕ_1 is equivalent to false, an optimized version of time elapse can be used which has greater performance.

4.3.1 Basic Time Elapse

The first step in performing the basic time elapse calculation is to construct a special rate BDD, ϕ_r, which is the disjunction of each rate BDD in the possible rate set, $\mathcal{R}$.

$$\phi_r = \bigvee_{\langle \phi_R, R \rangle \in \mathcal{R}} \phi_R \tag{4.5}$$

Given the definition of ϕ_r, time elapse can be calculated as follows:

$$\begin{aligned} \phi_1 \rightsquigarrow \phi_2 \quad \doteq \quad & \exists \delta \{ \delta \geq x_0 \wedge \exists \mathbf{c}_\delta \{ \phi_r[\dot{\mathbf{x}} := \mathbf{c}_\delta / \delta] \wedge \phi_2[\mathbf{x} := \mathbf{x} + \mathbf{c}_\delta] \wedge \\ & \forall \epsilon \{ x_0 \leq \epsilon \leq \delta \Rightarrow \exists \mathbf{c}_\epsilon \{ \phi_r[\dot{\mathbf{x}} := c_\epsilon / \epsilon] \wedge \phi_1[\mathbf{x} := \mathbf{x} + \mathbf{c}_\epsilon] \} \} \} \end{aligned} \tag{4.6}$$

This formulation of time elapse expands on the work of [50] to allow for nonunity rates based on concepts in [8]. During time elapse, the real variables ϵ and δ are introduced to evolve time, and $\mathbf{c}_\delta$ and $\mathbf{c}_\epsilon$ variables are introduced for each real variable in combination with ϕ_r to calculate the new range of values for each real variable as they evolve at their

```
timeElapse(φ1, φ2, φI, R)
   if (checkImplication(φ1, φI) == false)
      return optimizedTimeElapse(φ2, R)
   else
      φresult = ¬timeElapsePhi1(φ1, R, φI)
      return timeElapsePhi2(φ2, φresult, R)
   end if
```

Figure 4.16. Algorithm for time elapse operation.

varying rates. Additionally, a substitution is performed on ϕ_r using the corresponding δ and c_δ variables. For example, if ϕ_r contains a separation predicate of the form $\dot{x} \geq 2$, it is replaced with the new separation predicate $c_{\delta x} \geq 2\delta$. Similarly, a separation predicate $x \geq x_0 + 2$ in ϕ_1 or ϕ_2 is replaced with a new separation predicate $x \geq -c_{\delta x} + 2$. Once these substitutions have been performed, transitivity constraints are inserted with respect to the temporary variables resulting in new separation predicates where the values have been evolved based on δ.

The time elapse formula includes sums of two real variables. This operation cannot be performed using separation logic. Therefore, the δ, ϵ, $\mathbf{c}_\delta$, and $\mathbf{c}_\epsilon$ variables are replaced with negations of themselves:

$$\begin{aligned}\phi_1 \rightsquigarrow \phi_2 \quad \doteq \quad & \exists\bar{\delta}\{\bar{\delta} \leq x_0 \wedge \exists\bar{\mathbf{c}}_\delta\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\delta/\bar{\delta}] \wedge \phi_2[\mathbf{x} := \mathbf{x} - \bar{\mathbf{c}}_\delta] \wedge \\ & \forall\bar{\epsilon}\{\bar{\delta} \leq \bar{\epsilon} \leq x_0 \Rightarrow \exists\bar{\mathbf{c}}_\epsilon\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\epsilon/\bar{\epsilon}] \wedge \phi_1[\mathbf{x} := \mathbf{x} - \bar{\mathbf{c}}_\epsilon]\}\}\}\} \qquad (4.7)\end{aligned}$$

By performing this substitution, rather than substituting for the actual variables, the zero reference point x_0 can be shifted backwards by substituting it for the $\mathbf{c}_\delta$ and $\vec{c_\epsilon}$ variables:

$$\begin{aligned}\phi_1 \rightsquigarrow \phi_2 \quad \doteq \quad & \exists\bar{\delta}\{\bar{\delta} \leq x_0 \wedge \exists\bar{\mathbf{c}}_\delta\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\delta/\bar{\delta}] \wedge \phi_2[x_0 := \bar{\mathbf{c}}_\delta] \wedge \\ & \forall\bar{\epsilon}\{\bar{\delta} \leq \bar{\epsilon} \leq x_0 \Rightarrow \exists\bar{\mathbf{c}}_\epsilon\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\epsilon/\bar{\epsilon}] \wedge \phi_1[x_0 := \bar{\mathbf{c}}_\epsilon]\}\}\}\} \qquad (4.8)\end{aligned}$$

Finally, the universal quantification can be converted to an existential quantification:

$$\begin{aligned}\phi_1 \rightsquigarrow \phi_2 \quad \doteq \quad & \exists\bar{\delta}\{\bar{\delta} \leq x_0 \wedge \exists\bar{\mathbf{c}}_\delta\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\delta/\bar{\delta}] \wedge \phi_2[x_0 := \bar{\mathbf{c}}_\delta] \wedge \\ & \neg\exists\bar{\epsilon}\{\bar{\delta} \leq \bar{\epsilon} \leq x_0 \wedge \neg\exists\bar{\mathbf{c}}_\epsilon\{\phi_r[\dot{\mathbf{x}} := \bar{\mathbf{c}}_\epsilon/\bar{\epsilon}] \wedge \phi_1[x_0 := \bar{\mathbf{c}}_\epsilon]\}\}\}\} \qquad (4.9)\end{aligned}$$

The algorithms that implement Equation 4.9 are shown in Figure 4.17 and Figure 4.18. In these algorithms, d variables are used in place of $\mathbf{c}_\delta$ and $\mathbf{c}_\epsilon$ variables since only one set of these variables exist at any one time. The algorithm shown in Figure 4.17 is responsible for performing the ϵ portion of Equation 4.9 where the substitutions are performed for each separation predicate, the rate BDD is applied, transitivity constraints are applied for each of the temporary d variables and the temporary d variables are existentially abstracted.

The algorithm shown in Figure 4.18 is responsible for performing the δ portion of Equation 4.9. In this portion of the algorithm the substitutions are performed for each separation predicate, the rate BDD in terms of δ is applied, transitivity constraints are applied for each of the temporary d variables, and the temporary d variables are

```
timeElapsePhi1(φ1, R, φI)
   // All separation predicates containing x0 will be replaced with a new
   // separation predicate with a d variable corresponding to the non x0
   // variable in the separation predicate.
   for each BDD variable φi ∈ φ1 mapping to c1x1 ≥ c2x2 + c3
      if (x1 == x0)
         φsub = d2 ≥ c2x2 + c3
         dvars = dvars ∪ {d2}
      else if (x2 == x0)
         φsub = c1x1 ≥ d1 + c3
         dvars = dvars ∪ {d1}
      end if
      φ1.Compose(φsub, φi)
   end for
   φresult = φ1 ∧ ratesInTermsOf(R, ε)
   // Add transitivity constraints for all d variables
   for each (d ∈ dvars)
      φresult = addConstraints(φresult, d)
   end for
   // Abstract all separation predicates containing d variables.
   φabstract = conjunction of all separation predicates in φresult containing a d variable
   φresult = φresult.ExistAbstract(φabstract)
   φresult = ¬φresult ∧ (x0 ≥ ε) ∧ (ε ≥ δ)
   φresult = addConstraints(φresult, ε)
   // Abstract all separation predicates containing ε
   φabstract = conjunction of all separation predicates in φresult containing ε
   φresult = φresult.ExistAbstract(φabstract)
   φresult = ¬φresult ∧ (x0 ≥ δ) ∧ φI
   return φresult
```

Figure 4.17. First portion of algorithm for performing nonoptimized time elapse.

existentially abstracted. In the final steps, an additional constraint asserting that $\delta \leq x_0$ is inserted (keeping in mind that time is moving backwards), constraints over δ are added, and separation predicates containing δ are existentially abstracted.

4.3.2 Optimized Time Elapse

The previously described approach to performing the time elapse calculation has several disadvantages. In particular, due to the separation predicate substitution that occurs, a large number of separation predicates are created and mapped to BDD variables that are used for only a short period. Additionally, the constraint generation process is a time consuming operation, especially in this case where `addConstraints` is called once for each real variable. An optimized version of time elapse has been developed which can be applied in situations where ϕ_1 is true, i.e., when a safety property is being checked. Applying the knowledge that ϕ_1 is true to Equation 4.6, results in the following:

```
timeElapsePhi2(φ2, φresult, R)
   // All separatin predicates containing x0 will be replaced with a new
   // separation predicate with a d variable corresponding to the non x0
   // variable in the separation predicate.
   for each BDD variable φi ∈ φ2 mapping to an c1x1 ≥ c2x2 + c3
      if (x1 == x0)
         φsub = d2 ≥ c2x2 + c3
         dvars = dvars ∪ {d2}
      else if (x2 == x0)
         φsub = c1x1 ≥ d1 + c3
         dvars = dvars ∪ {d1}
      end if
      φ2.Compose(φsub, φi)
   end for
   φresult = φ2 ∧ φresult ∧ ratesInTermsOf(R, δ) ∧ (x0 ≥ δ)
   // Add transitivity constraints for all d variables
   for each (d ∈ dvars)
      φresult = addConstraints(φresult, d)
   end for
   // Abstract all separation predicates containing d variables.
   φabstract = conjunction of all separation predicates in φresult containing d variables.
   φresult = φresult.ExistAbstract(φabstract)
   φresult = φresult ∧ (x0 ≥ δ)
   φresult = addConstraints(φresult, δ)
   // Abstract all separation predicates containing δ
   φabstract = conjunction of all separation predicates in φresult containing δ
   φresult = φresult.ExistAbstract(φabstract)
   return φresult
```

Figure 4.18. Second portion of algorithm for performing nonoptimized time elapse.

$$\phi_1 \rightsquigarrow \phi_2 \quad \doteq \quad \exists\delta\{\delta \geq x_0 \wedge \exists\mathbf{c}_\delta\{\phi_r[\dot{\mathbf{x}} := \mathbf{c}_\delta/\delta] \wedge \phi_2[\mathbf{x} := \mathbf{x} + \mathbf{c}_\delta]\}\} \tag{4.10}$$

Given this simplified calculation of time elapse, Theorem 3.2 from [50] is adapted for use with separation predicates that allow for nonunity rates to avoid the introduction of $\mathbf{c}_\delta$ variables which would be immediately existentially abstracted from the representation. Using this method, new inequalities are directly constructed based on the rates of the continuous variables. This is performed by iterating over the possible rate sets, $\mathcal{R}$, and operating on each portion of the state space where ϕ_R is true as shown in Figure 4.19. However, in order for time to elapse, none of the clocks must have reached a value of zero. This is ensured using the `canTimeElapse` algorithm shown in Figure 4.20. A clock c is determined to have a value of zero if the separation predicates $x_0 \geq c + 0$ and $c \geq x_0 + 0$ exist simultaneously in the given product term, ϕ.

After ensuring that time can elapse, the separation predicates in that portion of the state space are evolved backwards based on the rates for each continuous variable in R. This is performed using the algorithm `addNewIneqs` shown in Figures 4.21. The algorithm

```
optimizedTimeElapse(φ, R)
    φ_result = false
    // Iterate over the possible rate sets.
    for each (⟨φ_R, R⟩ ∈ R)
        φ_part = φ ∧ φ_R
        for each product term φ_i ∈ φ_part
            if (canTimeElapse(φ_i))
                φ_i = addNewIneqs(φ_i, R)
                φ_i = abstractInconsistentIneqs(φ_i, R)
            end if
            φ_result = φ_result ∨ φ_i
        end for
    end for
    return φ_result
```

Figure 4.19. Algorithm for performing the optimized time elapse operation.

```
canTimeElapse(φ)
    // If a clock has a zero value, time cannot move any further
    // (Recall that exploration is performed in a backwards fashion.)
    // Note that φ must be a product term.
    for each term φ_1 ∈ φ
        if φ_1 maps to (x_0 ≥ c + 0) where c ∈ C
            for each term φ_2 ∈ φ
                if φ_2 maps to (c ≥ x_0 + 0) where c ∈ C
                    return false
                end if
            end for
        end if
    end for
    return true
```

Figure 4.20. Algorithm for determining if time can elapse in a given BDD.

operates on a particular product term ϕ given a rate set R that is associated with that product term. This algorithm does not create additional temporary real variables and all the separation predicates containing those variables that the previously discussed time elapse algorithm creates. Additionally, it attempts to insert only new separation predicates that tighten the state space, remove separation predicates that are no longer the tightest bound, and to remove separation predicates that are no longer consistent. It does this by iterating over all pairs of separation predicates in ϕ that contain x_0 and creating new temporary separation predicates that account for the other real variable in that separation predicate evolving at the specified rate. Note that a new BDD variable is not actually mapped to these temporary separation predicates until it is determined that they are necessary.

```
addNewIneqs(φ, R)
   // Find all pairs of separation predicates containing x_0 and
   // find tightening constraints for each combination of the inverted
   // and noninverted forms of those separation predicates.
   for each BDD variable φ_i ∈ φ mapping to c_1x_1 ≥ c_2x_2 + c_3 where (x_1 == x_0 ∨ x_2 == x_0)
      for each BDD variable φ_j ∈ φ mapping to c_4x_4 ≥ c_5x_5 + c_6 where (x_4 == x_0 ∨ x_5 == x_0)
         ⟨c'_1, x'_1, c'_2, x'_2, c'_3⟩ = trueForm(R, c_1, x_1, c_2, x_2, c_3);
         ⟨c'_4, x'_4, c'_5, x'_5, c'_6⟩ = trueForm(R, c_4, x_4, c_5, x_5, c_6);
         φ = φ ∧ transTT(c'_1, x'_1, c'_2, x'_2, c'_3, c'_4, x'_4, c'_5, x'_5, c'_6, φ_i, φ_j, φ)
         ⟨c'_1, x'_1, c'_2, x'_2, c'_3⟩ = trueForm(R, c_1, x_1, c_2, x_2, c_3);
         ⟨c'_4, x'_4, c'_5, x'_5, c'_6⟩ = falseForm(R, c_4, x_4, c_5, x_5, c_6);
         φ = φ ∧ transFT(c'_4, x'_4, c'_5, x'_5, c'_6, c'_1, x'_1, c'_2, x'_2, c'_3, φ_j, φ_i, φ)
         ⟨c'_1, x'_1, c'_2, x'_2, c'_3⟩ = falseForm(R, c_1, x_1, c_2, x_2, c_3);
         ⟨c'_4, x'_4, c'_5, x'_5, c'_6⟩ = trueForm(R, c_4, x_4, c_5, x_5, c_6);
         φ = φ ∧ transFT(c'_1, x'_1, c'_2, x'_2, c'_3, c'_4, x'_4, c'_5, x'_5, c'_6, φ_i, φ_j, φ)
         ⟨c'_1, x'_1, c'_2, x'_2, c'_3⟩ = falseForm(R, c_1, x_1, c_2, x_2, c_3);
         ⟨c'_4, x'_4, c'_5, x'_5, c'_6⟩ = falseForm(R, c_4, x_4, c_5, x_5, c_6);
         φ = φ ∧ transFF(c'_1, x'_1, c'_2, x'_2, c'_3, c'_4, x'_4, c'_5, x'_5, c'_6, φ_i, φ_j, φ)
      end for
   end for
   return φ
```

Figure 4.21. Algorithm for creating new separation predicates based on a given rate set.

The algorithms in Figures 4.22 and 4.23 create temporary separation predicates based on whether or not the original separation predicate is interpreted in its noninverted or inverted form, respectively. Consider the general sequence of events in the original time elapse calculation. First a separation predicate is replaced with a new separation predicate accounting for an amount by which it changes. For example, the separation predicate $x_1 \geq x_0 + c_3$ is replaced by $x_1 + d \geq x_0 + c_3$. In the next step, a separation predicate relating the d variable to the amount of time that has elapsed, δ, is introduced. For example, if x_1 has a rate range of $[r_l, r_u]$, the separation predicates $r_u\delta \geq d$ and $d \geq r_l\delta$ are introduced. Next, transitivity constraints around the d variables are added resulting in new separation predicates directly relating δ and x_1. These intermediate steps are skipped using the algorithms in Figures 4.22 and 4.23. Given the separation predicate $x_1 \geq x_0 + c_3$ and the fact that it is to be interpreted in its noninverted form, the temporary separation predicates $x_1 \geq r_u\delta + c_3$ is directly created. It is necessary to know if the separation predicate is being evaluated in its inverted or noninverted form because this impacts whether the lower or upper bound of the rate range is used in forming the new separation predicate.

The next step is to form transitivity relationships among the temporarily created separation predicates. Three main functions are used for this purpose: `transTT`, `transFF`,

```
⟨c1, x1, c2, x2, c3⟩ trueForm(R, c1, x1, c2, x2, c3)
    if (x2 == x0)
        (ẋ1 := [rl, ru]) ∈ R
        return ⟨c1, x1, ru, δ, c3⟩
    else if (x1 == x0)
        (ẋ2 := [rl, ru]) ∈ R
        return ⟨rl, δ, c2, x2, c3⟩
    end if
```

Figure 4.22. Evolve the true form of a separation predicate based on a given rate set.

```
⟨c1, x1, c2, x2, c3⟩ falseForm(R, c1, x1, c2, x2, c3)
    if (x2 == x0)
        (ẋ1 := [rl, ru]) ∈ R
        return ⟨c1, x1, rl, δ, c3⟩
    else if (x1 == x0)
        (ẋ2 := [rl, ru]) ∈ R
        return ⟨ru, δ, c2, x2, c3⟩
    end if
```

Figure 4.23. Evolve the false form of a separation predicate based on a given rate set.

and `transFT`, as shown in Figures 4.24, 4.25, and 4.26. Respectively, these algorithms take two noninverted separation predicate, two inverted separation predicates, and an inverted separation predicate and a noninverted separation predicate and attempt to form transitivity constraints between these separation predicates. The transitivity relationships are formed based on Tables 4.2 and 4.3, which are discussed previously. As new transitivity constraints are formed, they are checked to see if they provide additional information in the current portion of the state space using the `applyTransCons` method.

The final step of the `addNewIneqs` algorithm is to determine if a new separation predicate introduced by a transitivity constraint provides any additional information to the state space. Additionally, removal of separation predicates that are no longer useful would improve the performance of the analysis since the resulting BDD would be reduced in size. This is where the `applyTransCons` algorithm as shown in Figure 4.27 comes into play. This algorithm begins by first determining if the new separation predicate is trivially true or false and handling those cases appropriately. In the next step of the algorithm, the portion of ϕ where both separation predicates that form the transitivity constraint (ϕ_1 and ϕ_2) are true is calculated. This portion of ϕ is referred to as ϕ_{12part}. Next, all the separation predicates in ϕ_{12part} that either imply or are implied by the new separation predicate ϕ_{new} in either its inverted or noninverted form are found. These

```
transTT(c1, x1, c2, x2, c3, c4, x4, c5, x5, c6, φ1, φ2, φ)
    if (x2 == x4)
        if (c2 > 0 ∧ c4 > 0)
            φ = applyTransCons(c1/c2, x1, c5/c4, x5, c3/c2 + c6/c4, true, φ, φ1, φ2)
        else if (c2 < 0 ∧ c4 < 0)
            φ = applyTransCons(c5/c4, x5, c1/c2, x1, −c3/c2 − c6/c4, true, φ, φ1, φ2)
        end if
    else if (x1 == x4)
        if (c1 > 0 ∧ c4 < 0)
            φ = applyTransCons(c5/c4, x5, c2/c1, x2, c3/c1 + −c6/c4, true, φ, φ1, φ2)
        end if
    else if (x2 == x5)
        if (c2 > 0 ∧ c5 < 0)
            φ = applyTransCons(c1/c2, x1, c4/c5, x4, c3/c2 + −c6/c5, true, φ, φ1, φ2)
        end if
    end if
    return φ
```

Figure 4.24. Calls `applyTransCons` on two noninverted separation predicates.

```
transFF(c1, x1, c2, x2, c3, c4, x4, c5, x5, c6, φ1, φ2, φ)
    if (xj == xk)
        if (cj > 0 ∧ ck > 0)
            φ = applyTransCons(ci/cj, xi, cm/ck, xm, c1/cj + c2/ck, false, φ, ¬φ1, ¬φ2)
        else if (cj < 0 ∧ ck < 0)
            φ = applyTransCons(cm/ck, xm, ci/cj, xi, −c1/cj − c2/ck, false, φ, ¬φ1, ¬φ2)
        end if
    else if (xi == xk)
        if (ci > 0 ∧ ck < 0)
            φ = applyTransCons(cm/ck, xm, cj/ci, xj, −c1/ci − c2/ck, false, φ, ¬φ1, ¬φ2)
        end if
    else if (xj == xm)
        if (cj > 0 ∧ cm < 0)
            φ = applyTransCons(ci/cj, xi, ck/cm, xk, −c1/cj − c2/cm, false, φ, ¬φ1, ¬φ2)
        end if
    end if
    return φ
```

Figure 4.25. Calls `applyTransCons` on two inverted separation predicates.

separation predicates are used to determine if ϕ_{new} is the most tightly bound separation predicate or not. Specifically, the inverses of all separation predicates that imply ϕ_{new} are cofactored from ϕ_{12part}, resulting in the portion of ϕ_{12part} where the new separation predicate is the most tightly bounding. Next ϕ_{12part} is modified to remove the separation predicates that ϕ_{new} implies since those separation predicates are certainly not tightly bounding. Finally, the temporary separation predicate is mapped to a BDD variable and conjunctively combined with the remaining portion of the BDD.

transFT$(c_1, x_1, c_2, x_2, c_3, c_4, x_4, c_5, x_5, c_6, \phi_1, \phi_2, \phi)$
  if $(x_1 == x_4)$
    if $(c_1 > 0 \wedge c_4 > 0)$
      ϕ = `applyTransCons`$(\frac{c_2}{c_1}, x_2, \frac{c_5}{c_4}, x_5, -\frac{c_3}{c_1} + \frac{c_6}{c_4}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_5}{c_4}, x_5, \frac{c_2}{c_1}, x_2, \frac{c_3}{c_1} - \frac{c_6}{c_4}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
    else if $(c_1 < 0 \wedge c_4 < 0)$
      ϕ = `applyTransCons`$(\frac{c_2}{c_1}, x_2, \frac{c_5}{c_4}, x_5, -\frac{c_3}{c_1} + \frac{c_6}{c_4}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_5}{c_4}, x_5, \frac{c_2}{c_1}, x_2, \frac{c_3}{c_1} - \frac{c_6}{c_4}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
    end if
  else if $(x_2 == x_5)$
    `if` $(c_2 > 0 \wedge c_5 > 0)$
      ϕ = `applyTransCons`$(\frac{c_4}{c_5}, x_4, \frac{c_1}{c_2}, x_1, -\frac{c_3}{c_2} + \frac{c_6}{c_5}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_1}{c_2}, x_1, \frac{c_4}{c_5}, x_4, \frac{c_3}{c_2} - \frac{c_6}{c_5}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
    else if $(c_2 < 0 \wedge c_5 < 0)$
      ϕ = `applyTransCons`$(\frac{c_4}{c_5}, x_4, \frac{c_1}{c_2}, x_1, -\frac{c_3}{c_2} + \frac{c_6}{c_5}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_1}{c_2}, x_1, \frac{c_4}{c_5}, x_4, \frac{c_3}{c_2} - \frac{c_6}{c_5}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
    end if
  else if $(x_2 == x_4)$
    `if` $(c_2 > 0 \wedge c_4 < 0)$
      ϕ = `applyTransCons`$(\frac{c_5}{c_4}, x_5, \frac{c_1}{c_2}, x_1, -\frac{c_3}{c_2} - \frac{c_6}{c_4}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_1}{c_2}, x_1, \frac{c_5}{c_4}, x_5, \frac{c_3}{c_2} + \frac{c_6}{c_4}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
    else if $(c_2 < 0 \wedge c_4 > 0)$
      ϕ = `applyTransCons`$(\frac{c_5}{c_4}, x_5, \frac{c_1}{c_2}, x_1, -\frac{c_3}{c_2} - \frac{c_6}{c_4}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_1}{c_2}, x_1, \frac{c_5}{c_4}, x_5, \frac{c_3}{c_2} + \frac{c_6}{c_4}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
    end if
  else if $(x_1 == x_5)$
    if $(c_1 > 0 \wedge c_5 < 0)$
      ϕ = `applyTransCons`$(\frac{c_2}{c_1}, x_2, \frac{c_4}{c_5}, x_4, -\frac{c_3}{c_1} - \frac{c_6}{c_5}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_4}{c_5}, x_4, \frac{c_2}{c_1}, x_2, \frac{c_3}{c_1} + \frac{c_6}{c_5}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
    else if $(c_1 < 0 \wedge c_5 > 0)$
      ϕ = `applyTransCons`$(\frac{c_2}{c_1}, x_2, \frac{c_4}{c_5}, x_4, -\frac{c_3}{c_1} - \frac{c_6}{c_5}$, **false**, ϕ, $\neg\phi_1, \phi_2)$
      ϕ = `applyTransCons`$(\frac{c_4}{c_5}, x_4, \frac{c_2}{c_1}, x_2, \frac{c_3}{c_1} + \frac{c_6}{c_5}$, **true**, ϕ, $\neg\phi_1, \phi_2)$
    end if
  end if
  return ϕ

Figure 4.26. Calls `applyTransCons` on normal and inverted separation predicates.

```
applyTransCons(c1, x1, c2, x2, c3, tense, φ, φ1, φ2)
   if c1x1 ≥ c2x2 + c3 is vacuously true
      if (tense == true)
         return φ
      else
         return φ ∧ ¬(φ1 ∧ φ2)
      end if
   end if
   if c1x1 ≥ c2x2 + c3 is vacuously false
      if (tense == true)
         return φ ∧ ¬(φ1 ∧ φ2)
      else
         return φ
      end if
   end if
   // Portion of φ where both φ1 and φ2 are true
   φ12part = φ1 ∧ φ2 ∧ φ
   if (φ12part == false)
      return φ
   end if
   // Find all separation predicates in φ12part that either
   // imply or are implied by φnew.
   φnew maps to separatin predicate c1x1 ≥ c2x2 + c3
   for each BDD variable φi ∈ φ12part mapping to a separation predicate
      if (tense == true)
         if (timpliest(φnew, φi))
            new_implies.insert(φi)
         else if (timpliesf(φnew, φi))
            new_implies.insert(¬φi)
         end if
         if (timpliest(φi, φnew))
            implies_new.insert(φi)
         else if (timpliesf(φi, φnew))
            implies_new.insert(¬φi)
         end if
      else
         if (fimpliest(φnew, φi))
            new_implies.insert(φi)
         else if (fimpliesf(φnew, φi))
            new_implies.insert(¬φi)
         end if
         if (timpliesf(φi, φnew))
            implies_new.insert(φi)
         else if (fimpliesf(φi, φnew))
            implies_new.insert(¬φi)
         end if
      end if
   end for
```

Figure 4.27. Algorithm for applying a more tightly bounding transitivity constraint.

```
// Portion of φ_12part where φ_new will be the tightest bound.
φ_tight = φ_12part
for each φ_i ∈ implies_new
    φ_tight = φ_tight.Cofactor(¬φ_i)
end for
// If φ_new will never be the tightest bound, do nothing.
// Otherwise, remove all separation predicates from φ_12part that are
// less tight than φ_new.    This also remove inconsistent constraints.
if (φ_tight != false)
    φ = φ - φ_12part
    for each φ_i ∈ new_implies
        φ_12part = φ_12part.Cofactor(φ_i)
    end for
    if (tense == true)
        φ_12part = φ_12part ∧ φ_new
    else
        φ_12part = φ_12part ∧ ¬φ_new
    end if
    φ = φ ∨ φ_12part
end if
return φ
```

Figure 4.27 continued.

current portion of the state representation.

The situation becomes slightly more complicated when the separation predicate does not contain x_0. A separation predicate of the form $c_1x_1 \geq c_2x_2 + c_3$, can be eliminated if when considering the ranges of rates, the left side of the inequality decreases more rapidly than the right side of the inequality in the worse case scenario. To demonstrate this, let's examine the separation predicate as time moves forward. If x_1 is changing with the range of rates $[r_{l1}, r_{u1}]$ and x_2 is changing with the range of rates $[r_{l2}, r_{u2}]$, and time is incremented by δ time steps, the evolved separation predicate would be of the form $c_1(x_1+[r_{l1}, r_{u1}]\delta) \geq c_2(x_2+[r_{l2}, r_{u2}]\delta)+c_3$. Manipulating this separation predicate results in $c_1x_1 \geq c_2x_2 + c_3 + (-c_1[r_{l1}, r_{u1}]\delta + c_2[r_{l2}, r_{u2}]\delta)$. It can be seen that in order for the separation predicate to remain satisfied, the portion in parentheses must remain less than or equal to zero, i.e., $0 \geq -c_1[r_{l1}, r_{u1}]\delta + c_2[r_{l2}, r_{u2}]\delta$ or $c_1[r_{l1}, r_{u1}] \geq c_2[r_{l2}, r_{u2}]$. Since c_1 and c_2 can be either positive or negative, *min* and *max* determine which rate value to select for the worst case scenario. Therefore, in order for the true form of a separation predicate to remain consistent, it must hold that $min(c_1[r_{l1}, r_{u1}]) \geq max(c_2[r_{l2}, r_{u2}])$. Similarly, in order for the false form of a separation predicate to remain consistent, it must hold that $max(c_1[r_{l1}, r_{u1}]) \leq min(c_2[r_{l2}, r_{u2}])$. The algorithm in Figure 4.28 accounts for the fact that time is in fact moving in reverse during analysis.

The final step of the `abstractInconsistentIneqs` algorithm is to remove the non-inverted or inverted form of the separation predicate mapping to the BDD variable ϕ_i. Since it is only necessary to remove the noninverted or inverted form of the separation predicate, but not the entire BDD variable, a special calculation using the BDD cofactor operation is performed.

The final result of the optimized time elapse is the disjunction of each BDD corresponding to a given rate set where new inequalities have been generated and inconsistent inequalities have been removed. This results in an overall BDD that accounts for all the possible time evolutions that could have occurred.

4.4 Clock Specification

Clock specification is used to specify properties in terms of time. Therefore, it is necessary to assign zero to specially created clock values during the analysis using the `specifyClock` method shown in Figure 4.29. This method performs a simple assignment to a real variable by finding all separation predicates containing that real variable

```
abstractInconsistentIneqs(φ, R)
   for each BDD variable φ_i ∈ φ mapping to c1x1 ≥ c2x2 + c3
      abstractTrue = false
      abstractFalse = false
      // Cases where separation predicate contains x0
      if (x1 == x0 ∧ (ẋ2 := [rl, ru]) ∈ R ∧ rl > 0)
         abstractFalse = true
      else if (x2 == x0 ∧ (ẋ1 := [rl, ru]) ∈ R ∧ rl > 0)
         abstractTrue = true;
      end if
      if (x2 == x0 ∧ (ẋ1 := [rl, ru]) ∈ R ∧ ru < 0)
         abstractFalse = true
      else if (x1 == x0 ∧ (ẋ2 := [rl, ru]) ∈ R ∧ ru < 0)
         abstractTrue = true;
      end if
      // Cases where separation predicate does not contain x0
      (ẋ1 := [rl1, ru1]) ∈ R
      (ẋ2 := [rl2, ru2]) ∈ R
      if (c1 > 0 ∧ c2 > 0)
         if (c1rl1 > c2ru2)
            abstractTrue = true
         end if
         if (c2rl2 > c1ru1)
            abstractFalse = true
         end if
      else if (c1 < 0 ∧ c2 > 0)
         if (c1ru1 > c2ru2)
            abstractTrue = true
         end if
         if (c2rl2 > c1rl1)
            abstractFalse = true
         end if
      else if (c1 > 0 ∧ c2 < 0)
         if (c1rl1 > c2rl2)
            abstractTrue = true
         end if
         if (c2ru2 > c1ru1)
            abstractFalse = true
         end if
      // Both c1 and c2 cannot be negative by rules of canonicity.
      end if
      if (abstractTrue)
         φ = φ.Cofactor(φ_i) ∨ (¬φ_i ∧ ¬(φ.Cofactor(φ_i)) ∧ φ.Cofactor(¬φ_i))
      end if
      if (abstractFalse)
         φ = φ.Cofactor(¬φ_i) ∨ (φ_i ∧ ¬(φ.Cofactor(¬φ_i)) ∧ φ.Cofactor(φ_i))
      end if
      return φ
   end for
```

Figure 4.28. Algorithm for removing separation predicates that are inconsistent.

```
specifyClock(z, φ)
   for each BDD variable φ_i ∈ φ that maps to a separation predicate containing z
      φ_i maps to the separation predicate c1x1 ≥ c2x2 + c3
      if (x1 == z)
         φ_sub = x0 ≥ c2x2 + c3
      else if (x2 == z)
         φ_sub = c1x1 ≥ x0 + c3
      end if
      φ.Compose(φ_sub, φ_i)
   end for
   return φ
```

Figure 4.29. Algorithm for assigning the value zero to a clock.

and substituting them with new separation predicates where the real variable has been replaced with x_0.

4.5 Generating Error Traces

In situations where it has been determined that the system violates the property, it is very beneficial to provide an error trace showing how the error state is reached. However, error trace generation is complicated by the backwards and depth-first exploration properties of the model checker. An algorithm for generating traces is shown in Figure 4.30. This algorithm assumes that the $\mathbf{T}\mu$ property contained only a single fixpoint and that the result of each fixpoint iteration is stored in a stack represented by the variable X. Note that the result of each fixpoint iteration is an HSL formula. This algorithm operates by first finding the product term in the result of the fixpoint's final iteration that implied the initial state. The algorithm then works forwards from this portion of the initial state to determine how the violating portion of the state space is reached. At each step, the transition precondition and time elapse calculations are applied to each product term of the top element of the fixpoint stack X resulting in ϕ_{next}. If the current portion of the state implies ϕ_{next}, then ϕ_{next} is reachable from the current state. This algorithm incurs significant overhead due to the requirement of storing the result of each fixpoint iteration. Additionally, applying `pre` and `timeElapse` to individual product terms can be very expensive since large numbers of product terms can potentially exist.

4.6 Summary

This chapter introduced a BDD based model checking algorithm for LHPNs, a key contribution of this dissertation. This algorithm is Boolean in nature while supporting

```
printErrorTrace(X, φ_init, φ_I, C, R)
   // X is a stack containing the result of each fixpoint iteration.
   // First find a product term in the last iteration of the fixpoint
   // that is implied by the initial state.
   for each product term φ_pterm ∈ X.top()
      if (checkImplication(φ_init, ¬φ_pterm))
         φ_cur = φ_pterm
         print φ_cur
         break
      end if
   end for
   X.pop()
   // For each remaining element in the stack, apply pre
   // and timeElapse and determine if the result is implied
   /  by φ_cur
   while (¬X.empty())
      for each product term φ_pterm ∈ X.top()
         φ_next = timeElapse(φ_I, pret(φ_pterm, φ_I, C), φ_I, R)
         if (checkImplication(φ_cur, ¬φ_next))
            φ_cur = φ_pterm
            print φ_cur
            X.pop()
            break
         end if
      end for
   end while
```

Figure 4.30. Algorithm for generating error trace.

continuous variables that can change within a range of rates. For improved performance, an optimized time elapse calculation is described. This calculation allows for analysis of systems that would normally not be possible due to the large size of the BDDs that result. In addition to this optimization, care has been taken to apply transitivity constraints in a manner as to not impact performance too dramatically. However, this results in a conservative rather than exact analysis and thus the possibility of false negatives.

CHAPTER 5

SMT BASED BOUNDED MODEL CHECKER

The *Satisfiability Modulo Theories* (SMT) problem is a generalization of the *Boolean Satisfiability* (SAT) problem where Boolean variables are replaced by predicates from various background theories [60]. These theories may include linear real and integer arithmetic, uninterpreted functions, and the theories of various data structures such as lists, arrays, and bit vectors [11, 12, 18, 21, 37, 39, 42].

Initial SMT solver implementations functioned by translating SMT instances into Boolean SAT instances and passing those SAT instances to a Boolean SAT solver. For example, to support integer arithmetic, multiple Boolean variables are used as a bit representation for integers and the necessary integer theories are specified as Boolean operations on those individual bit variables. This can result in extremely large Boolean SAT instances; however, existing SAT solvers can be used directly without modification. Therefore, as Boolean SAT state of the art increases, the improvements immediately benefit the SMT solvers. This approach can also be severely restricting. The loss of higher level knowledge of the underlying theories requires the Boolean SAT solver to work much harder to discover simple concepts. This problem is manifested by the large Boolean SAT instances that result. Therefore, more recent SMT solvers [20, 42, 59] closely integrate theory-specific solvers with a DPLL (Davis-Putnam-Logemann-Loveland) approach to Boolean satisfiability [60]. These types of SMT solvers are often referred to as DPLL(T) [42]. In this type of architecture, the DPLL-based SAT solver passes conjunctions of predicates belonging to theory T to a specialized solver. The specialized solver is then responsible for deciding feasibility of those predicates. Additionally, the particular theory solver must be able to explain the reasons for infeasibility, if necessary.

In 2005 and 2006 the SMT-COMP competition [16, 17] was held which has helped to drive improvements in SMT solvers. In 2006, 12 SMT solvers competed in one or more of 11 divisions. The success of this competition can be largely attributed to SMT-

LIB [62, 63] which provides a standardized specification for SMT problems and a large library of benchmarks. Participants included Barcelogic [42, 59], MathSAT [20], and Yices [38]. In 2006, Barcelogic was second place in the divisions in which it competed. The Barcelogic solver supports difference logic over integers, equality with uninterpreted functions among others. The MathSAT solver currently supports theories of equality, uninterpreted functions, separation logic, and linear arithmetic over reals and integers. Yices, the stand-out winner of the 2006 competition, includes an incremental Simplex algorithm for the theory of linear arithmetic that is tightly integrated within the DPLL framework. Yices strong ability to work with the theory of linear arithmetic made it particularly well suited for hybrid system model checking. For these reasons, Yices was selected as the SMT solver that the LHPN model checker described in this section is based upon.

The basic algorithm for performing SMT based bounded model checking of LHPNs is shown in Figure 5.1. The algorithm proceeds by creating an SMT instance in which statements are asserted. The initial state is first asserted, followed by the invariant for each iteration. Each iteration's next states are calculated by firing transitions or elapsing time. This is performed by asserting a disjunction of the guarded commands and a time elapse formula. Finally, a failure condition is asserted in terms of state variables from each iteration. After asserting each of these components, the SMT satisfiability check is performed. Satisfiability indicates that the property is violated because there is an assignment indicating that the failure condition is reachable. Unsatisfiability indicates that the property could not be violated in that number of iterations. This does not necessarily indicate that the property cannot be violated; however, so this is a bounded model checker. The remainder of this chapter describes the SMT based bounded model checking algorithm in greater detail.

5.1 State Variables and Initial State

To proceed with SMT based bounded model checking, it is first necessary to create a set of state variables for each iteration of the exploration. The state variables for each iteration, i, are defined using the tuple $\langle M^i, S^i, Q^i, C^i, A^i, BR^i \rangle$. In other words, for each iteration, Boolean marking variables, Boolean signal variables, real variables, clock variables, Boolean clock active variables, and Boolean rate variables are created.

Throughout the SMT model checking procedure, the BDDs that are constructed

```
smtCheck(φ_init, φ_I, C, R, maxIterations)
   SMTInstance ins;
   iteration = 0
   // Assert initial state
   ins.assert(mkExprForBDD(φ_init), 0)
   while (iteration < maxIterations)
      // Assert invariant for current iteration
      ins.assert(mkExprForBDD(φ_I, iteration))
      trans = true
      for each ⟨φ_G, A⟩ ∈ C
         trans = trans ∨ mkExprForGC(φ_G, A, iteration)
      end for
      trans = trans ∨ mkExprForTimeElapse(R, iteration)
      ins.assert(trans)
      iteration++
   end while
   ins.assert(mkExprForFailProp(maxIterations))
   if (ins.check == true)
      return ''Property Violated''
   else
     return ''Property Not Violated''
   end if
```

Figure 5.1. Basic algorithm for performing SMT based model checking of LHPNs.

during symbolic model generation are used as the foundation from which the SMT based analysis is performed. This is necessary because basic Boolean operations are necessary in order to build the symbolic model. For example, during the invariant construction a Boolean based exploration of the LHPN is performed. Additionally, when the guarded commands are generated, Boolean operations must be used to merge primary and secondary guarded commands. BDDs provide a convenient and efficient mechanism for performing these operations. The pseudo-code for the algorithm that constructs SMT statements from arbitrary BDDs is shown in Figure 5.2. The algorithm first iterates over each product term in the BDD and then iterates over each term within the product term. Variables corresponding to Boolean variables in the symbolic model are directly mapped to the specified iteration's state variables. BDD variables that map to inequalities are converted to inequality expressions over the specified iteration's real variables. The exact procedure by which SMT statements are constructed depends on the particular SMT solver. Yices provides a convenient API for constructing SMT expressions using conjunction and disjunction operations.

After constructing each iteration's state variables, the first step in SMT model checking is to assert the initial state (ϕ_{init}) in terms of the initial iteration's variables (i.e., $i = 0$). The SMT assertation statement for the initial state is built directly from the BDD

```
mkExprForBDD(φ, iteration)
   i = iteration
   if (φ == true)
      return mkTrueExpr()
   else if φ == false
      return mkFalseExpr()
   else
      result = false
      for product term φ_i ∈ φ
         product = true
         for term t ∈ φ_i
            if t maps to (c1x1 ≥ c2x2 + c3) and is true literal
               result = result ∧ (c1x1 ≥ c2x2 + c3)
            else if t maps to (c1x1 ≥ c2x2 + c3) and is false literal
               result = result ∧ ¬(c1x1 ≥ c2x2 + c3)
            else if t maps to a Boolean and is true literal
               product = product ∧ t^i
            else if t maps to a Boolean and is false literal
               product = product ∧ ¬t^i
            end if
         end for
         result = result ∨ product
      end for
   end if
   return result
```

Figure 5.2. Algorithm for constructing SMT statements for BDDs.

representation of the initial state. For example, the following initial state is asserted in the SMT solver for the switched capacitor integrator in Figure 2.17:

$$p_0^0\,\overline{p_1^0}\,p_2^0\,\overline{p_3^0}\,p_4^0\;\overline{\overline{Vin^0}}\;\overline{\overline{fail^0}}\;\overline{\dot{V}\,out^0_{[-22,-18]}}\;\dot{V}\,out^0_{[18,22]}$$
$$\overline{a_{t_0}^0}\;\overline{a_{t_1}^0}\;a_{t_2}^0\;\overline{a_{t_3}^0}\;\overline{a_{t_4}^0} \wedge c_{t_2}^0 = 0 \wedge Vout^0 = -1000$$

5.2 Invariant

At each iteration of the analysis, it is necessary to assert the invariant in terms of that iteration's set of state variables. This is constructed from the BDD representation of the invariant using the method discussed previously. The invariant for the integrator in Figure 2.17 is asserted in the SMT solver as follows for each iteration i:

$$
\begin{array}{l}
p_0^i\,\overline{p_1^i}\,p_2^i\,\overline{p_3^i}\,p_4^i\;\overline{Vin^i}\;\overline{fail^i}\;\overline{\dot{V}out^i_{[-22,-18]}}\;\dot{V}out^i_{[18,22]} \vee \\
p_0^i\,\overline{p_1^i}\,\overline{p_2^i}\,p_3^i\,p_4^i\;Vin^i\;\overline{fail^i}\;\overline{\dot{V}out^i_{[-22,-18]}}\;\dot{V}out^i_{[18,22]} \vee \\
\overline{p_0^i}\,p_1^i\,\overline{p_2^i}\,p_3^i\,p_4^i\;Vin^i\;\overline{fail^i}\;\dot{V}out^i_{[-22,-18]}\;\overline{\dot{V}out^i_{[18,22]}} \vee \\
\overline{p_0^i}\,p_1^i\,p_2^i\,\overline{p_3^i}\,p_4^i\;\overline{Vin^i}\;\overline{fail^i}\;\dot{V}out^i_{[-22,-18]}\;\overline{\dot{V}out^i_{[18,22]}} \vee \\
p_0^i\,\overline{p_1^i}\,p_2^i\,\overline{p_3^i}\,\overline{p_4^i}\;\overline{Vin^i}\;fail^i\;\overline{\dot{V}out^i_{[-22,-18]}}\;\dot{V}out^i_{[18,22]} \vee \\
p_0^i\,\overline{p_1^i}\,\overline{p_2^i}\,p_3^i\,\overline{p_4^i}\;Vin^i\;fail^i\;\overline{\dot{V}out^i_{[-22,-18]}}\;\dot{V}out^i_{[18,22]} \vee \\
\overline{p_0^i}\,p_1^i\,\overline{p_2^i}\,p_3^i\,\overline{p_4^i}\;Vin^i\;fail^i\;\dot{V}out^i_{[-22,-18]}\;\overline{\dot{V}out^i_{[18,22]}} \vee \\
\overline{p_0^i}\,p_1^i\,p_2^i\,\overline{p_3^i}\,\overline{p_4^i}\;\overline{Vin^i}\;fail^i\;\dot{V}out^i_{[-22,-18]}\;\overline{\dot{V}out^i_{[18,22]}}
\end{array}
$$

5.3 Time Elapse

The time elapse portion of the next state assertion makes use of the possible rate set ($\mathcal{R}$) to calculate the values of real variables as a result of time moving forward. This algorithm is shown in Figure 5.3. In calculating the next state via time elapse, a new real variable is created representing the amount of time that has elapsed. This variable is referred to as $\delta^{i,j}$, and it represents the amount of time that has elapsed between iterations i and j. Since time is moving forward, $\delta^{i,j}$ is always greater than or equal to zero. Based on the current values of the Boolean rate variables, the real variables change by some multiple of $\delta^{i,j}$. Additionally, all clock variables increase by exactly $\delta^{i,j}$. Lastly, all Boolean variables in the next state have the same value as in the current state.

The complete time elapse assertion for the integrator in Figure 2.17, given the current state i and the next state j, is as follows:

$$
\begin{array}{l}
((\dot{V}out^i_{[18,22]} \wedge \overline{\dot{V}out^i_{[-22,-18]}} \wedge 18\delta^{i,j} + Vout^i \leq Vout^j \leq 22\delta^{i,j} + Vout^i) \vee \\
(\overline{\dot{V}out^i_{[18,22]}} \wedge \dot{V}out^i_{[-22,-18]} \wedge -22\delta^{i,j} + Vout^i \leq Vout^j \leq -18\delta^{i,j} + Vout^i)) \wedge \\
c^j_{t_0} = c^i_{t_0} + \delta^{i,j} \wedge c^j_{t_1} = c^i_{t_1} + \delta^{i,j} \wedge c^j_{t_2} = c^i_{t_2} + \delta^{i,j} \wedge c^j_{t_3} = c^i_{t_3} + \delta^{i,j} \wedge c^j_{t_4} = c^i_{t_4} + \delta^{i,j} \wedge \\
\delta^{i,j} \geq 0 \wedge a^j_{t_0} = a^i_{t_0} \wedge a^j_{t_1} = a^i_{t_1} \wedge a^j_{t_2} = a^i_{t_2} \wedge a^j_{t_3} = a^i_{t_3} \wedge a^j_{t_4} = a^i_{t_4} \wedge \\
\dot{V}out^j_{[18,22]} = \dot{V}out^i_{[18,22]} \wedge \dot{V}out^j_{[-22,-18]} = \dot{V}out^i_{[-22,-18]} \wedge \\
Vin^j = Vin^i \wedge fail^j = fail^i \wedge p^j_0 = p^i_0 \wedge p^j_1 = p^i_1 \wedge p^j_2 = p^i_2 \wedge p^j_3 = p^i_3 \wedge p^j_4 = p^i_4
\end{array}
$$

5.4 Transition Relations

The transition relation portion of the next state assertion makes use of the merged guarded command set ($\mathcal{C}$) to calculate the values of the next state variables based on

```
mkExprForTimeElapse(R, iteration)
   i = iteration;
   j = iteration + 1;
   result = δ^{i,j} ≥ 0
   // Increment real variables based on R
   rates = false
   for (⟨φ_R, R⟩ ∈ R)
      rate = mkExprForBDD(φ_R, i)
      for ((v̇ := [r_l, r_u]) ∈ R)
         rate = rate ∧ (v^j ≥ v^i + r_l δ^{i,j})
         rate = rate ∧ (v^j ≤ v^i + r_u δ^{i,j})
      end for
      rates = rates ∨ rate
   end for
   // All clocks increment by δ
   for (c ∈ C)
      result = result ∧ (c^j = c^i + δ^{i,j})
   end for
   // Next state Boolean variables get same value as current state
   for (b ∈ {M ∪ S ∪ A ∪ BR})
      result = result ∧ (b^j = b^i)
   end for
   result = result ∧ rates
   return result
```

Figure 5.3. Generating an SMT statement representing the time elapse calculation.

the values of the current state variables. The algorithm for constructing the assertion statement for a given guarded command is shown in Figure 5.4. Essentially, the guard portion (ϕ_G) of the guarded command is asserted in terms of the current state while the assignment portion of the guarded command makes use of both the current and the next state variables. Assignments that are specified in the assignment set ($\mathcal{A}$) are performed on the next state variables while variables that have no assignment performed on them are simply assigned the same value as in the current state. There is one exception, however. If a clock is assigned the range $[-\infty, \infty]$, no assignment is made to that clock variable. This has the effect of allowing the clock to remain undefined in the next state.

In the integrator example, the corresponding SMT assertion statement for the merged guarded command $\mathcal{C}(t_2, t_0)$, given the current state i and the next state j, is as follows:

```
mkExprForGC(φ_G , A , iteration)
   i = iteration;
   j = iteration + 1;
   // Clock does not change at all when guarded command is applied
   result = (δ^{i,j} = 0)
   // Guard in terms of current state
   result = result ∧ mkExprForBDD(φ_G)
   // Perform assignments on Boolean variables.    If no assignment is made,
   // next value is same as current value.
   for (b ∈ {M ∪ S ∪ A ∪ BR})
      if ((b := true) ∈ A)
         result = result ∧ (b^j = true)
      else if ((b := false) ∈ A)
         result = result ∧ (b^j = false)
      else
         result = result ∧ (b^j = b^i)
      end if
   end for
   // Perform assignments on real variables.    If no assignment is made,
   // next value is same as current value.    If assignment is to
   // [−∞, ∞], next value is left undefined.
   for (v ∈ {C ∪ Q})
      if ((v := [−∞, ∞]) ∈ A)
         // Do Nothing.
      else if ((v := [a_l, a_u]) ∈ A)
         result = result ∧ (v^j ≥ a_l)
         result = result ∧ (v^j ≤ a_u)
      else
         result = result ∧ (v^j = v^i)
      end if
   end for
   return result
```

Figure 5.4. Generating an SMT statement for a given guarded command.

$$
\begin{aligned}
&\delta^{i,j} = 0 \wedge p_0^i \wedge p_2^i \wedge \overline{p_3^i} \wedge \overline{a_{t_0}^i} \wedge a_{t_2}^i \wedge c_{t_2}^i \geq 100 \wedge \\
&p_0^j = p_0^i \wedge p_1^j = p_1^i \wedge p_2^j = \textbf{false} \wedge p_3^j = \textbf{true} \wedge p_4^j = p_4^i \wedge \\
&a_{t_0}^j = \textbf{true} \wedge c_{t_0}^j = 0 \wedge a_{t_1}^j = a_{t_1}^i \wedge c_{t_1}^j = c_{t_1}^i \wedge \\
&a_{t_2}^j = \textbf{false} \wedge a_{t_3}^j = a_{t_3}^i \wedge c_{t_3}^j = c_{t_3}^i \wedge a_{t_4}^j = a_{t_4}^i \wedge c_{t_4}^j = c_{t_4}^i \wedge \\
&Vin^j = Vin^i \wedge fail^j = fail^i
\end{aligned}
$$

Note that in this example, the variable $c_{t_2}^j$ is not assigned any value. This is because the assignment set dictates that it is assigned the value $[-\infty, \infty]$. By not performing any assignment on $c_{t_2}^j$, it can have any value.

An assertion for the next state calculation is made based on the disjunction of each guarded command and the time elapse assertion. This allows for a choice to be made about what happens to get to the next state to reach a failure condition, if possible.

5.5 Specifying Properties

The final step of the model checking procedure is to assert the property and apply the SMT checking procedure. If a satisfiable solution is found, this indicates that it is possible to reach the violating condition. Otherwise, the property is found not to be violated within that number of iterations. This, however, does not necessarily mean that the property can never be violated. The current SMT based model checker only supports the verification of safety properties, i.e., those properties that are specified in TCTL using only the AG operator. In the case of LHPNs that are generated from VHDL-AMS code, the property AG($\neg fail$) is equivalent to stating that the failure condition occurs if the signal *fail* is asserted. Therefore, in these cases, it is sufficient to construct a disjunction of the *fail* signal over all iterations as shown in Figure 5.5.

In the case of the integrator example in Figure 2.17 it is suitable to assert the disjunction of Boolean signal *fail* at each iteration as follows (for a maximum iteration count of five):

$$fail^0 \vee fail^1 \vee fail^2 \vee fail^3 \vee fail^4$$

In the event that the property is violated, the SMT solver generates a satisfying solution to the current context. This satisfying solution directly corresponds to a trace over all iteration's state variables beginning from an initial state that led to the error condition. In order to make the error trace more useful to designers, only the elements of the state that change at each step of the trace are displayed.

This model checker heavily relies on the SMT solver to find satisfying solutions to sets of inequalities over real variables. Therefore, the exactness of the model checking algorithm is directly dependent on the exactness of the SMT solver being used. From a theoretical standpoint, however, this model checking algorithm is exact and should not result in false negatives or false positives, aside from false positives that may arise as a result of not model checking over enough iterations.

```
mkExprForFailProp(φ, maxIterations)
   result = false
   for (i ∈ {0..maxIterations})
      result = result ∨ fail^i
   end for
   return result;
```

Figure 5.5. Generating an SMT statement representing the property under verification.

CHAPTER 6

RESULTS

The VHDL-AMS to LHPN compiler, the LHPN to symbolic model generator, the BDD based model checking algorithm (ATACS-BDD), and the SMT based bounded model checking algorithm (ATACS-SMT) described in this dissertation have been implemented and results are promising. This chapter describes examples that are used to test and analyze the performance of these verification methods. Performance results are also provided. The water level monitor example from the hybrid system domain models a basic system containing only a few discrete and continuous variables. Additionally, an example has been developed based on a basic analog circuit. Specifically, a switched capacitor integrator circuit has been modeled and analyzed. Each example can be made to satisfy or violate the specified property by modifying the model's parameters. All results in this chapter were collected on a 2 Ghz Intel Core Duo with 2 GB of main memory.

6.1 Water Level Monitor

The VHDL-AMS code shown in Figure 6.1 models a water level monitor for a tank. This is a standard hybrid system example from [4]. The water level monitor continuously senses the water level in a tank to ensure that it never empties and never goes above 13 inches. When the pump is not active, the water level decreases at 2 inches per second. When the pump is active, the water level increases at 1 inch per second. It takes 1 to 2 seconds for the pump to activate or deactivate when signaled. In this example, the real variable y, represents the current level of the water, and the variable *inc* indicates if the water is presently increasing or, in other words, if the pump is currently active. When the water level reaches 10 inches, the pump is deactivated 1 to 2 seconds later and the water level begins to decrease. One to 2 seconds after the water level reaches 5 inches, the pump is activated causing the water level to rise once again. The verification goal for this example is to ensure that the water level never becomes fully empty and never

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.handshake.all;
use work.nondeterminism.all;
entity water_level is
end water_level;
architecture monitor of water_level is
   quantity y:real;   -- water level
   signal inc:std_logic := '1'; -- is water level going up or down?
begin
   break y => 2.0; --Initial value
   if inc='1' use
      y'dot == 1.0;
   elsif inc = '0' use
      y'dot == -2.0;
   end use;
   process begin
      wait until y'above(10.0);
      assign(inc,'0',1,2);
      wait until not y'above(5.0);
      assign(inc,'1',1,2);
   end process;
   assert (y'above(0.0) and not y'above(13.0))
      report ``Overfill/underfill of the tank.''
      severity failure;
end monitor;
```

Figure 6.1. VHDL-AMS for a water level monitor.

rises above 13 inches. The model can be modified to cause failure by increasing the lower bound above zero inches and/or decreasing the upper bound below 13 inches.

The LHPN model of the water level monitor shown in Figure 6.2 is automatically generated from the VHDL-AMS in Figure 6.1. Initially, the water level is 2 inches and increasing at a rate of 1 inch per second. The LHPN in Figure 6.2a controls the rate of change of y based on the current value of *inc*. The LHPN in Figure 6.2b causes the value of *inc* to change based on the current water level. The final LHPN in Figure 6.2c assigns the variable *fail* to **true** if the property becomes violated.

Table 6.1 shows the result of applying ATACS-BDD and ATACS-SMT to the water example. For comparison, the results of applying ATACS-DBM are also provided. The upper half of the table shows the results of applying the three different model checkers to the water level monitor shown in Figure 6.2 with varying assertion ranges for the water level. Since ATACS-SMT can only provide a nonviolating result for a given bound on the number of iterations, timing results for upper bounds of 10, 20, 30, and 40 iterations are shown. For ATACS-BDD, the number of fixpoint iterations required to verify the

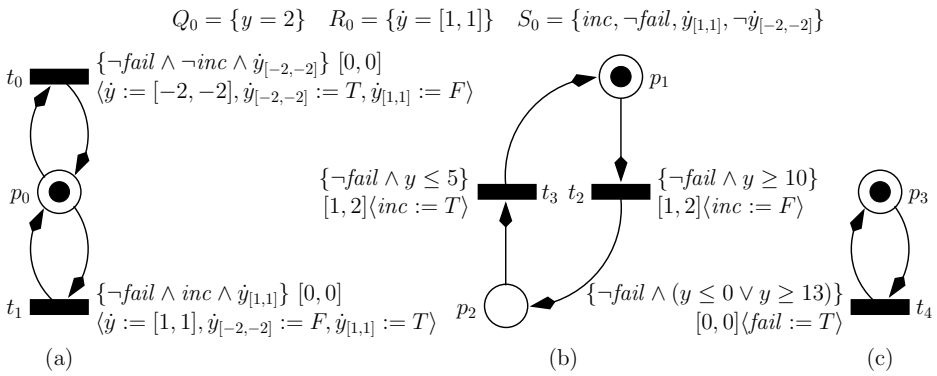


Figure 6.2. LHPN model of a water level monitor.

Table 6.1. Water level monitor verification results.

	Assertion Range	Exp. Result	ATACS-BDD		ATACS-SMT		ATACS-DBM	
			Time (s)	Iter.	Time (s)	Iter.	Time (s)	Zones
1x	$0 \leq y \leq 13$	Pass	< 1	5	< 1	10	< 2*	10*
	$0 \leq y \leq 13$	Pass	–	–	5	20	–	–
	$0 \leq y \leq 13$	Pass	–	–	29	30	–	–
	$0 \leq y \leq 13$	Pass	–	–	119	40	–	–
	$0 \leq y \leq 12$	Fail	< 1	4	< 1	10	< 1	6
	$1 \leq y \leq 13$	Fail	< 1	7	< 2	15	< 1	10
	$1 \leq y \leq 12$	Fail	< 1	4	< 1	10	< 1	6
2x	$1 \leq y \leq 25$	Pass	< 1	5	< 1	10	< 2	9
	$1 \leq y \leq 25$	Pass	–	–	3	20	–	–
	$1 \leq y \leq 25$	Pass	–	–	26	30	–	–
	$1 \leq y \leq 25$	Pass	–	–	104	40	–	–
	$1 \leq y \leq 24$	Fail	< 1	4	< 1	10	< 1	6
	$2 \leq y \leq 25$	Fail	< 1	7	< 2	15	< 1	10
	$2 \leq y \leq 24$	Fail	< 1	4	< 1	10	< 1	6

* Verification result does not match expected result.

property is shown. Note that in the case of ATACS-DBM, the property is incorrectly found to be violated for an assertion range of $0 \leq y \leq 13$. This is due to the DBM model checker's sensitivity to round off error since it uses integer approximation to represent the state space. In order to avoid this issue, the model can be multipled by a factor of two. This results in a model where y begins to increase two to four time units after y reaches ten, and y begins to decrease two to four times units after y reaches 20. Results of applying verification to the multiplied model are shown in the second half of Table 6.1. The assertions ranges also have to be multiplied by a factor of two. All model checkers, including ATACS-DBM, now get the proper result on all test cases.

6.2 Switched Capacitor Integrator

The next example for which results are provided is the switched capacitor integrator circuit which has been used as an example throughout this dissertation. The circuit, shown in Figure 2.1, takes as input a 5 kHz square wave that varies from -1000 mV to 1000 mV and generates a triangle wave as output representing the integral of the input voltage. The verification goal is to ensure that *Vout* never saturates (i.e., it is always between -2000 mV and 2000 mV). By varying the ranges of rate for *Vout*, the circuit can be made to violate or satisfy this property as shown in Table 6.2. In particular, when the lower and upper bound for these rates are equal, both the ATACS-BDD and ATACS-SMT model checkers determine in a few seconds of CPU time that the property is satisfied (i.e., the circuit does not saturate). When the lower and upper bounds are not equal, both ATACS-BDD and ATACS-SMT determine correctly in a few seconds that the circuit violates the property. This error occurs if the rising slew rate of *Vout* is consistently larger than the falling slew rate leading to a build up of charge eventually

Table 6.2. Switched capacitor integrator verification results.

Rate Ranges	Exp. Result	ATACS-BDD		ATACS-SMT		ATACS-DBM	
		Time (s)	Iter.	Time (s)	Iter.	Time (s)	Zones
$[20, 20]$	Pass	< 1	7	< 1	10	< 1	4
$[20, 20]$	Pass	–	–	7	20	–	–
$[20, 20]$	Pass	–	–	505	30	–	–
$[18, 22]$	Fail	< 2	11	< 2	15	N/A	N/A
$[18, 22]$ (Piecewise)	Fail	< 1	6	60	20	< 1	9

saturating the high supply rail. Once again, since ATACS-SMT can only demonstrate that the property is not violated for a specified number of iterations, results are provided where 10, 20, 30, and 40 iterations are allowed. Also, note that ATACS-DBM cannot directly support ranges of rates. Therefore, a piecewise approximate model must first be generated in which the rate of *Vout* initially increases at 18 mV/μs. After some random amount of time, the rate may switch to 22 mV/μs. Similar behavior is modeled when *Vout* is decreasing.

6.3 Corrected Switched Capacitor Integrator

Saturation of the switched capacitor integrator can be prevented using the circuit shown in Figure 6.3. The corresponding VHDL-AMS model of this circuit and LHPN model is shown in Figures 6.4 and 6.5, respectively. In this circuit, a resistor in the form of a switched capacitor is inserted in parallel with the feedback capacitor. This causes *Vout* to drift back to 0 V. In other words, if *Vout* is increasing, it increases faster when it is far below 0 V than when it is near or above 0 V. Therefore, the model for this circuit uses a *Vout* range of 28 to 37 mV/μs when it is below -1000 mV, a range of 18 to

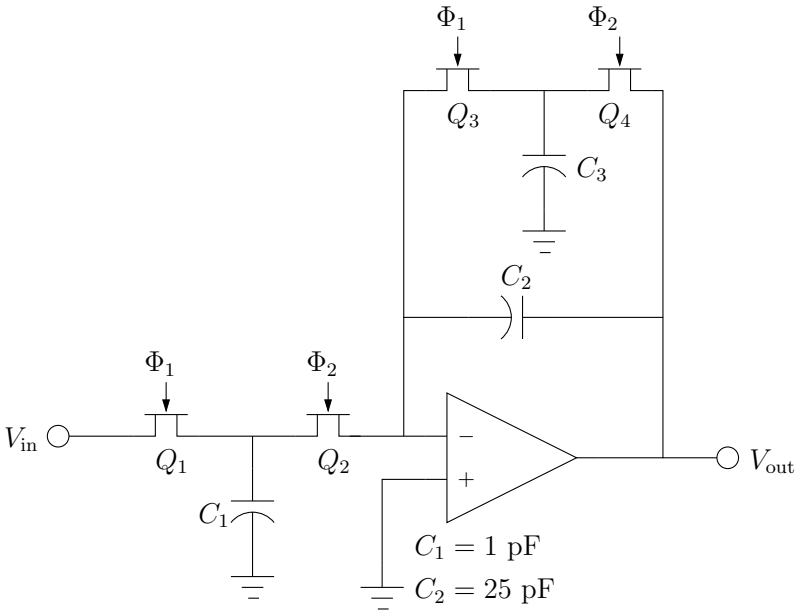


Figure 6.3. Circuit diagram of a corrected switched capacitor integrator.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.handshake.all;
use work.nondeterminism.all;
entity integrator is
end integrator;
architecture switchCap of integrator is
   quantity Vout:real;
   signal vin::std_logic := '0';
begin
   -- Initial Conditions
   break Vout => -1000.0
   if vin='0' use
      if (not vout'above(-1000.0))' use
         vout'dot == span(28.0, 37.0);
      elsif (not vout'above(0.0))' use
         vout'dot == span(18.0, 32.0);
      elsif (not vout'above(1000.0))' use
         vout'dot == span(8.0, 22.0);
      else
         vout'dot == span(3.0, 12.0);
      end use;
   elsif vin='1' use
      if (not vout'above(-1000.0))' use
         vout'dot == span(-12.0, -3.0);
      elsif (not vout'above(0.0))' use
         vout'dot == span(-22.0, -8.0);
      elsif (not vout'above(1000.0))' use
         vout'dot == span(-32.0, -18.0);
      else
         vout'dot == span(-37.0, -28.0);
      end use;
   end use;
   process begin
      assign(vin,'1',100,100);
      assign(vin,'0',100,100);
   assert (Vout'above(-2000.0) and
       not Vout'above(2000.0))
      report ``Error:  The output voltage railed.''
      severity failure;
end switchCap;
```

Figure 6.4. VHDL-AMS for a fixed switched capacitor integrator.

$Q_0 = \{\, Vout = -1000\}$ $R_0 = \{\, \dot{V}\, out = [18, 32]\}$ $S_0 = \{\neg\, Vin, \neg fail, \dot{V}\, out_{[18,32]}, \neg\, \dot{V}\, out_{[28,37]}, ...\}$

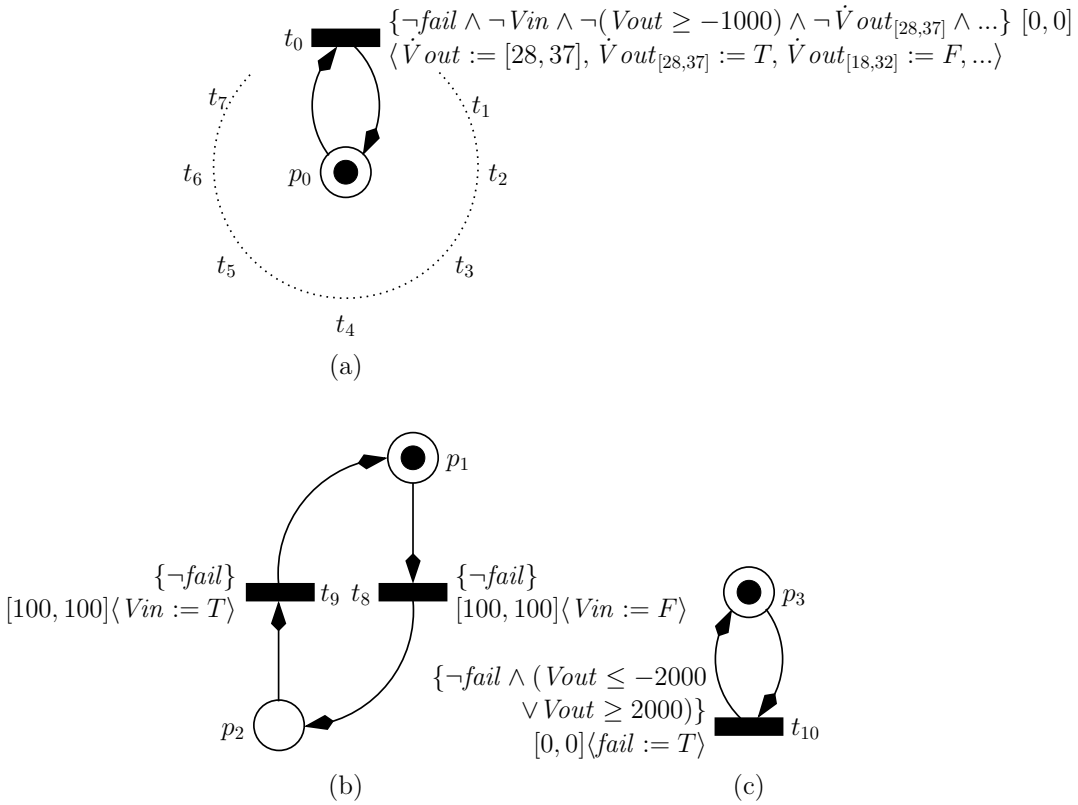


Figure 6.5. LHPN model of fixed switched capacitor integrator.

32 mV/μs when it is below 0 mV, a range of 8 to 22 mV/μs when it is below 1000 mV, and a range of 3 to 12 /mV/μs when it is above 1000 mV. A similar modification is made for the ranges of rates when *Vout* is decreasing.

Figure 6.5 shows the LHPN that is automatically generated from the VHDL-AMS in Figure 6.4. The rate is determined by the LHPN in Figure 6.5a. Note that seven transitions have been omitted to keep the figure readable. However, transitions t_1 through t_7 are of the same form as transition t_0 except with a different rate value and enabling condition on *Vout*. The LHPN in Figure 6.5b represents the input signal which oscillates between high and low every 100 μs, and the LHPN in Figure 6.5c is used to indicate if the failure condition has been reached.

Table 6.3 shows the result of applying ATACS-BDD, ATACS-SMT, and ATACS-DBM to the fixed integrator example. The table also shows results where the model has been modified to force failure. In this case, the circuit is forced to fail by modifying the rates

Table 6.3. Corrected switched capacitor integrator verification results.

Rate Ranges	Exp. Result	ATACS-BDD		ATACS-SMT		ATACS-DBM	
		Time (s)	Iter.	Time (s)	Iter.	Time (s)	Zones
$[-22,-8],[-12,-3]$	Pass	6*	6*	28	10	< 1	55
$[-22,-8],[-12,-3]$	Pass	–	–	388	20	–	–
$[-32,-8],[-32,-3]$	Fail	9	6	47	10	< 1	14

* Verification result does not match expected result.

such that when decreasing, the output voltage does not taper off as it reaches the rail value (i.e., when *Vout* is below zero). The results show that in the situation where the circuit should pass the verification, ATACS-BDD finds a failure. This false negative is attributed to the inexactness in ATACS-BDD that is present as a result of the not adding transitivity constraints at all necessary phases of the the analysis. Due to the high cost of performing this operation, attempts are made to avoid it. Therefore, potential exists for the analysis to generate false negatives as in this case. However, if transitivity constraints are inserted at all required steps of the analysis, system memory is quickly consumed and analysis can not complete. The ATACS-SMT model checker, however, can correctly determine that the circuit does not violate the property for up to 20 iterations since it is exact. ATACS-DBM correctly verifies the circuit in both situations.

6.4 Optimization Evaluation

Results comparing performance of ATACS-BDD with various optimizations enabled and disabled are shown in Table 6.4. Specifically, results where constraints are added on a limited basis and the optimized time elapse is used, are shown along side results where each of these capabilities has been disabled. The second column shows results where constraints are inserted at all necessary points but the optimized time elapse is still enabled. In this case, only the water level monitor examples are able to complete without running out of memory. The third column shows results where constraints are added on a limited basis but the original time elapse calculation is applied. In this case, the corrected switched capacitor integrator examples exceed memory after completing only a few iterations. These results clearly demonstrate the necessity of optimizations to achieve reasonable performance.

Table 6.4. Corrected switched capacitor integrator verification results.

Rate Ranges	Exp. Result	Opt. Cons. Opt. T.E.		Unopt. Cons. Opt. T.E.		Opt. Cons. Unopt. T.E.	
		Time (s)	Iter.	Time (s)	Iter.	Time (s)	Iter.
Water Level Monitor							
$0 \leq y \leq 13$	Pass	< 1	5	8	5	4	5
$0 \leq y \leq 12$	Fail	< 1	4	3	4	3	4
$1 \leq y \leq 13$	Fail	< 1	7	10	7	6	7
$1 \leq y \leq 12$	Fail	< 1	4	3	4	3	4
Switched Capacitor Integrator							
$[20, 20]$	Pass	< 1	7	OOM	5	4	7
$[18, 22]$	Fail	< 2	11	OOM	5	688	13
Corrected Switched Capacitor Integrator							
$[-22, -8], [-12, -3]$	Pass	6*	6*	OOM	2	OOM	3
$[-32, -8], [-32, -3]$	Fail	9	6	OOM	2	OOM	2

* Verification result does not match expected result.

OOM = Out of Memory.

CHAPTER 7

CONCLUSIONS

Formal verification of analog and mixed-signal circuits presents several challenges. The largest of these challenges is the representation of continuous variables. Therefore, efficient hybrid system verification methods provide potential approaches for verifying analog and mixed-signal circuits. The modeling and verification methods described in this dissertation demonstrate that hybrid system analysis approaches can be used with modifications. Additionally, to encourage the acceptance of formal verification in the analog and mixed-signal world, designers must be able to specify systems using familiar methods. Therefore, this dissertation describes methods for generating LHPN models from VHDL-AMS descriptions and differential equation models. These methods show success in the verification of AMS circuits.

7.1 Summary

This work begins by describing the approach for modeling AMS systems. Designs specified in a number of ways including VHDL-AMS are automatically compiled into an LHPN which includes Boolean signals to represent digital circuitry and continuous variables to model voltages and currents in the analog circuitry. The LHPN model provides a formalism for reasoning about the system being analyzed. The LHPN formalism is a primary contribution of this dissertation. This new modeling method is necessary because existing hybrid system models such has LHA rely on invariants to force transitions, a construct not naturally present in AMS circuits; and existing hybrid Petri net models relied on the notion of continuous quantities that flow between places which is superfluous to in this application. LHPN models can also be hand-generated by the user, if necessary.

After the LHPN model is created, a symbolic model is generated. The process of encapsulating the transition behavior of LHPNs in terms of an invariant and guarded command set is a further contribution of this dissertation. The symbolic model is well-suited for the BDD based symbolic model checking algorithm and SMT-based bounded

model checking algorithm. Additionally, system properties are specified as temporal logic formulas using TCTL. TCTL can be automatically generated from **assert** statements in VHDL-AMS or more complicated properties can be specified by the designer.

Once the symbolic model has been constructed, and the property has been specified, either a BDD based or SMT based model checking algorithm may be applied to check the property. In the BDD based model checking algorithm, separation predicates are mapped to Boolean variables so that analysis can be performed using BDD operations. The Boolean analysis relies on a canonical representation of a restricted form of inequalities. This algorithm necessitates the addition of constraints among separation predicates—an expensive operation to perform. Therefore, the algorithm attempts to avoid this operation resulting in an approximate model checker and the possibility of false negative results. Alternatively, in the SMT based model checking algorithm, separation predicates are directly handled by the SMT solver. The BDD based model checker is particularly well suited for analyzing abstracted models since for larger models, the number of BDD variables that get created can be very restrictive. The SMT based model checker can efficiently determine if the full model violates the property given a number of iterations, however it can never fully guarantee that the property is not violated. These methods are demonstrated on several examples from the hybrid system and AMS domains.

The contributions of this work result in new methods and tools necessary for the formal verification of AMS circuits. Specifically, a hybrid modeling method for AMS circuits known as LHPNs is developed and formally described, and a translation to a symbolic model suitable for model checking is described. Additionally, two main model checking algorithms, a BDD based algorithm and an SMT based algorithm, for performing state space explorations of LHPNs and verifying properties over the state space are developed. Finally, these methods are applied to several examples demonstrating the usefulness and necessity of these methods.

7.2 Challenges

There are several challenges associated with this research. Chief among these challenges is the adaptation of the time elapse method used by the BDD based model checker. Supporting variables that can change at ranges of rates is much more involved than in the timed case. The optimized version of time elapse that was developed is crucial to getting reasonable performance results.

Another challenge is that a perfect and exact adaptation of the BDD based model checking algorithm has difficulty completing on even simple examples. This is due to the fact that addition of transitivity constraints in theory needs to be performed frequently, but results in an extremely large BDD representation that includes loose constraints, which are difficult to later remove. Substantial time was spent experimenting with different approaches and placements of constraint generation. This results in a trade off between exactness and the ability to arrive at a verification result—a typical model checking issue.

A final issue is the backward and breadth first nature of the BDD based algorithm in combination with the use of BDDs. During the debugging process, trying to determine what state precedes a current state is counter-intuitive. Additionally, since this algorithm is breadth first, rather than working with individual states that are reachable via a single transition firing, sets of states are found that can result from firing all possible transitions. This is multiplied by the very large and complicated state representation due to the use of BDDs. This issue is partially alleviated by the use of the trace generation algorithm described in Chapter 4. However, it is not always possible to find a trace due to the trace algorithm's significant overhead and the algorithm does not indicate which transition resulted in the next state. Development of a BDD based forward reachability algorithm could contribute to reducing this issue as well. However, since the work described in this dissertation is based on previous work that performs backward reachability, a forward reachability algorithm has not yet been explored.

7.3 Future Work

There are many potential directions of further investigation in the area of formal verification of analog/mixed-signal circuits, and more generally in hybrid system modeling and verification. The development of verification methods for niche AMS applications, benchmark development, and abstraction approaches are three areas of future work that merit the most attention. Further descriptions of these particular areas of future work, and several others, are described here.

7.3.1 Generating Models from SPICE-decks

SPICE is generally considered to be the industrial standard for computer-aided analysis of microelectronic circuits. Providing a tool that generates models from SPICE-deck

input files or simulation data would help to encourage the adoption of formal verification techniques.

Along these lines, in [32], Dastidar and Chakrabarti describe a method for constructing a finite state machine model of an analog circuit by performing repeated SPICE simulations while varying parameter values. An LHPN model can be constructed in a similar manner from arbitrarily complex AMS circuits using SPICE simulations or systems of differential equations using differential equation simulations. This procedure could be performed by creating multiple simulations with varying input parameters resulting in sets of time series data. The simulation data would then be subdivided into regions based on several possible factors (e.g., equal numbers of data points in each region, evenly spaced regions, etc.). An LHPN or similar model could then be constructed from this subdivided data.

This approach to constructing models is neither exact nor over-approximate, but it allows for modeling of potentially very complex systems. Additionally, by adjusting the number of simulations and the ranges of parameter values, the precision of the model could be increased or decreased.

7.3.2 Generating VHDL-AMS from LHPNs

Another area of potential investigation is into the generation of VHDL-AMS descriptions from LHPN models. Consider a situation where a complicated circuit consisting of multiple components is being simulated. Often, a particular component of the circuit may be a bottleneck in the simulation of the overall circuit. By modeling that component as an LHPN, applying abstraction techniques, and then generating a VHDL-AMS description (or similar HDL description that can be simulated) of the abstracted model, there is potential to dramatically speed up the simulation while still being able to provide useful information to the designer.

7.3.3 Improved User Feedback

In the case of a property being violated, it is important to provide the designer with information about the cause of the failure rather than a simple report of failure. Initially, this information may consist of a trace that resulted in the failure. However, in a large circuit or in a situation where the failure does not occur for a very long time, a trace may also be of limited use to the designer. Much work remains to be done in providing designers with useful information to aid in pinpointing the exact cause of failure.

Another problem associated with model checking is determining if the property being verified is meaningful. A positive verification result may mean that the property being checked is valid or vacuously true. The portion of the design intended to be verified may not even be exercised by the property. Therefore, an approach to determining the coverage of the property would be very useful to designers in determining if the design is well tested.

7.3.4 Verification Reuse

Formal verification tool flows generally operate by taking an input circuit and property, and generating a verification result. Upon completion, all computations that are performed to determine the result are discarded. Therefore, after modifying the circuit and/or the property, the process begins from scratch. The goal of verification reuse is to use previous verification runs to more quickly perform future verification runs after either the model has been slightly changed or the property has been modified. Verification reuse can be particularly beneficial when the verification process is very time-consuming. *Incremental verification* provides one possible approach. In incremental verification, subsystems are identified and analysis is performed on each subsystem. If a modification is made to the system, only the impacted subsystems need to be reanalyzed. These methods can be investigated and applied to the AMS verification domain.

7.3.5 Niche Applications

Given an oscillating input signal, a phase-locked loop (PLL) generates an output signal with a matching frequency that is in phase with the input by automatically raising or lowering the frequency of an oscillator until it is matched to the input. PLLs are used in a wide range of analog and digital systems such as for clock and data recovery in high speed data streams or for clock generation in high frequency processors. Additionally, PLLs are notoriously difficult to simulate and validate. This emphasizes the need for improved verification tools. The work described in this dissertation focuses on developing general approaches for the formal verification of AMS circuits. However, by focusing on subsets of AMS circuits such as PLLs (or even particular subcategories of PLLs), formal verification methods can potentially exploit properties present in that category of circuit resulting in dramatically improved utility and performance. PLLs are just one of the many niche circuit categories where AMS formal verification methods developed specifically for that niche could be of great value.

7.3.6 Benchmark Development

During the development of the work described in this dissertation, a significant amount of time was devoted to developing relevant examples of systems to analyze. This can be a challenging task given that this work was developed in an academic setting, and the legal restrictions placed on industry in making their work public. As a result, it can be difficult for researchers to understand the true challenges including the important types of circuits and properties of interest that industrial AMS designers are faced with. Therefore, a suite of benchmark examples with real-world relevance would really benefit researchers and help to drive the direction of research.

7.3.7 Abstraction and Refinement

A common hurdle associated with model checking is the state space explosion problem. Therefore, it is necessary to apply abstraction techniques. Abstraction methods can be applied to the model or to the analysis procedure. In the first case, the full model is conservatively abstracted into a simpler model, which is then analyzed with the goal that the abstracted model generates a state space that can be represented using less memory. In the second case, as the state space is explored, the representation is stored in an abstract form where additional behavior may be introduced in order to minimize the state space representation.

One potential approach to abstracting the LHPN model is safe net transformations. This method has been previously applied to timed Petri nets [72]. The safe net transformations are used to reduce the size and the complexity of the timed Petri net without eliminating behavior that is critical to determining if the property is satisfied or violated. These methods could be similarly applied to LHPN models.

In the case of the BDD based model checker, analysis is impeded by the large number of BDD variables that are created during analysis. When analyzing a larger system, this can fully deplete the system resources preventing successful verification. Therefore, the second type of abstraction approach may be well-suited because it has the potential to directly reduce the number of BDD variables and thus the BDD explosion problem. Specifically, rather than creating a new inequality whenever necessary, an already existing inequality with similar values could be used instead. The decision about whether to create a new inequality or substitute an existing inequality can be based on the number of BDD variables that already exist, and/or the difference between the new inequality and the existing inequality. Both of these factors could be tuned to adjust the level of abstraction.

The use of abstraction techniques introduces the possibility of falsely reporting that properties are violated. This necessitates the use of algorithms for detecting if a property truly fails and if not, refining the abstraction and re-running the verification process.

One potentially promising abstraction and refinement approach is to combine the BDD based LHPN model checker and SMT based bounded model checker in such a way that their individual strengths are maximized. Specifically, the BDD based model checker is capable of performing an unbounded full state space exploration. However, do to the large number of BDD variables that are created, memory utilization quickly becomes an issue. This implies that the BDD based model checker would be well-suited for analyzing abstracted models. The SMT based model checker can efficiently determine if the full model violates the property given a number of iterations; however it can never fully guarantee that the property is not violated.

Figure 7.1 shows a potential tool flow that uses the model checkers in combination. The symbolic model is provided to both model checkers. In the case of the BDD based model checker, the symbolic model is first abstracted. If the BDD based model checker determines that the property is violated in the abstract model, the SMT based model checker is used with the full model to ensure that the failure is not a false negative. In order to accomplish this, it is necessary for the BDD based model checker to specify the number of iterations that are required for the abstract model to fail. If the SMT based model checker verifies that the full symbolic model does fail in that number of iterations, verification is complete. If the full symbolic model does not violate the property, the violation is determined to be a false negative and the unsatisfying core is used to refine the abstract model. The process then repeats. If at any point, the BDD based model checker determines that the abstract symbolic model does not violate the property, then the property is certainly not violated in the full model and the verification can terminate immediately.

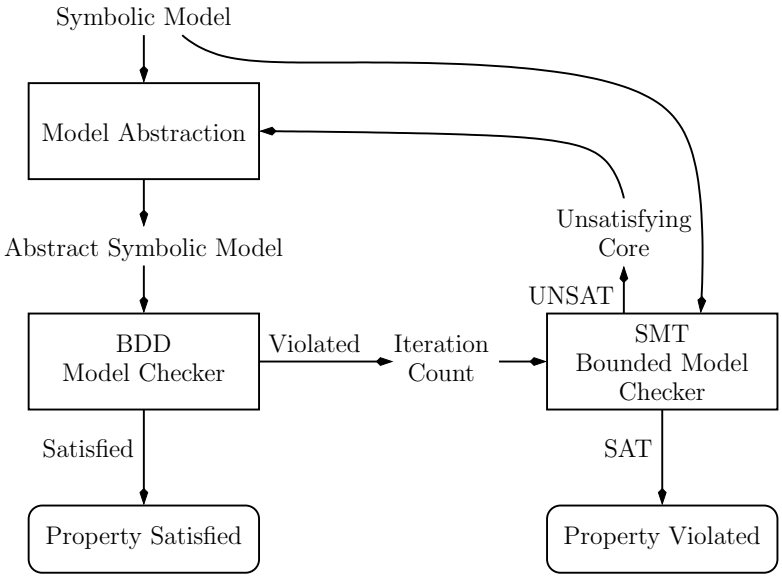


Figure 7.1. Using the SMT model checker in combination with the BDD model checker.

REFERENCES

[1] Abadi, M., and Lamport, L. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems 16*, 5 (September 1994), 1543–1571.

[2] Abdulla, P. A., Bouajjani, A., and Jonsson, B. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *Computer Aided Verification* (1998), pp. 305–318.

[3] Akers, S. B. Binary decision diagrams. *IEEE Transactions on Computers C-27*, 6 (June 1978), 509–516.

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

[5] Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P.-H. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems* (1992), pp. 209–229.

[6] Alur, R., Grosu, R., Hur, Y., Kumar, V., and Lee, I. Modular specification of hybrid systems in CHARON. In *HSCC* (2000), pp. 6–19.

[7] Alur, R., Henzinger, T. A., and Ho, P.-H. Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium* (1993), pp. 2–11.

[8] Alur, R., Henzinger, T. A., and Ho, P.-H. Automatic symbolic verification of embedded systems. In *IEEE Transactions on Software Engineering* (1996), pp. 181–201.

[9] Annichini, A., Asarin, E., and Bouajjani, A. Symbolic techniques for parametric reasoning about counter and clock systems. In *Computer Aided Verification* (2000), pp. 419–434.

[10] Annichini, A., Bouajjani, A., and Sighireanu, M. Trex: A tool for reachability analysis of complex systems. In *Computer Aided Verification* (2001), pp. 368–372.

[11] Armando, A., Castellini, C., and Giunchiglia, E. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning (Durham, UK)* (2000), S. Biundo and M. Fox, Eds., vol. 1809 of *Lecture Notes in Computer Science*, Springer, pp. 97–108.

[12] Armando, A., Castellini, C., Giunchiglia, E., and Maratea, M. A sat-based decision procedure for the boolean combination of difference constraints. In *SAT* (2004).

[13] Asarin, E., Bournez, O., Dang, T., and Maler, O. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control* (2000), vol. LNCS 1790, Springer-Verlag, pp. 20–31.

[14] Asarin, E., Dang, T., and Maler, O. d/dt: A verification tool for hybrid systems. In *Proc. of IEEE Conference on Decision and Control* (2001), pp. 2893–2898.

[15] Bagnara, R., Hill, P. M., and Zaffanella, E. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[16] Barrett, C., de Moura, L., and Stump, A. Design and results of the first satisfiability modulo theories competition (smt-comp 2005). *Journal of Automated Reasoning V35*, 4 (November 2005), 373–390.

[17] Barrett, C., de Moura, L., and Stump, A. SMT-COMP: Satisfiability Modulo Theories Competition. In *17th International Conference on Computer Aided Verification* (2005), K. Etessami and S. Rajamani, Eds., Springer, pp. 20–23.

[18] Barrett, C., Dill, D., and Stump, A. Checking satisfiability of first-order formulas by incremental translation to sat, 2002.

[19] Belluomini, W., and Myers, C. J. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design 19*, 5 (May 2000), 501–520.

[20] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Ranise, S., van Rossum, P., and Sebastiani, R. Efficient satisfiability modulo theories via delayed theory combination. In *CAV 2005* (2005), K. Etessami and S. K. Rajamani, Eds., vol. 3576 of *Lecture Notes in Computer Science*, Springer, pp. 335–349.

[21] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., and Sebastiani, R. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)* (2005), N. Halbwachs and L. D. Zuck, Eds., vol. 3440 of *Lecture Notes in Computer Science*, Springer, pp. 317–333.

[22] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (1986), 677–691.

[23] Chang, C.-L., and Lee, R. C.-T. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[24] Chen, H., and Hanisch, H.-M. Analysis of hybrid systems based on hybrid net condition/event system model. *Discrete Event Dynamic Systems: Theory and Applications 11* (Jan. 2001), 163–185.

[25] Chutinan, A., and Krogh, B. H. Computing approximating automata for a class of linear hybrid systems. *Lecture Notes in Computer Science 1567* (1999), 16–37.

[26] Chutinan, A., and Krogh, B. H. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. *Lecture Notes in Computer Science 1569* (1999), 76–90.

[27] Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. The MIT Press, 1999.

[28] Cohen, H. Buddy. Web page, March 2007. http://buddy.sourceforge.net.

[29] Coudert, O., and Madre, J. C. A unified framework for the formal verification of sequential circuits. In *ICCAD* (1990), pp. 126–129.

[30] Dang, T., Donze, A., and Maler, O. Verification of analog and mixed-signal circuits using hybrid systems techniques. In *Formal Methods for Computer Aided Design* (2004), pp. 21–36.

[31] Dang, T., and Maler, O. Reachability analysis via face lifting. In *HSCC* (1998), pp. 96–109.

[32] Dastidar, T. R., and Chakrabarti, P. P. A verification system for transient response of analog circuits using model checking. In *VLSID '05: Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 195–200.

[33] David, R. Modeling of hybrid systems using continuous and hybrid petri nets. In *Proceedings of the Seventh International Workshop on Petri Nets and Performance Models, June 3-6, 1997, Saint Malo, France* (Los Alamitos, California, June 1997), IEEE Computer Society, pp. 47–58.

[34] David, R., and Alla, H. On hybrid petri nets. *Discrete Event Dynamic Systems: Theory and Applications 11* (Jan. 2001), 9–40.

[35] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. In *Communications of the ACM, 5:394-397* (1962).

[36] Davis, M., and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the ACM 7* (1960), 201–215.

[37] de Moura, L., and Rue, H. Lemmas on demand for satisfiability solvers, 2002.

[38] Dutertre, B., and de Moura, L. A fast linear-arithmetic solver for DPLL(T). *Computer Aided Verification* (2006), 81–94.

[39] Filliâtre, J.-C., Owre, S., Ruess, H., and Shankar, N. ICS: Integrated Canonization and Solving (Tool presentation). In *Proceedings of CAV'2001* (2001), G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 246–249.

[40] Frehse, G. Phaver: Algorithmic verification of hybrid systems past HyTech. In *HSCC* (2005), M. Morari and L. Thiele, Eds., vol. 3414 of *Lecture Notes in Computer Science*, Springer, pp. 258–273.

[41] Frehse, G., Krogh, B. H., and Rutenbar, R. A. Verifying analog oscillator circuits using forward/backward refinement. In *Proc. Design, Automation and Test in Europe (DATE)* (2006), IEEE Computer Society Press, pp. 257–262.

[42] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., and Tinelli, C. DPLL(T): Fast Decision Procedures. In *16th International Conference on Computer Aided Verification, CAV'04* (2004), R. Alur and D. Peled, Eds., vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 175–188.

[43] Gupta, S., Krogh, B. H., and Rutenbar, R. A. Towards formal verification of analog and mixed-signal designs. TECHCON 2003, 2003.

[44] Gupta, S., Krogh, B. H., and Rutenbar, R. A. Towards formal verification of analog designs. In *International Conference on Computer-Aided Design* (2004), pp. 210–217.

[45] Hartong, W., Hedrich, L., and Barke, E. Model checking algorithms for analog verification. In *Design Automation Conference* (2002), pp. 542–547.

[46] Hartong, W., Hedrich, L., and Barke, E. On discrete modeling and model checking for nonlinear analog systems. In *Computer Aided Verification. 14th International Conference* (2002), pp. 401–413.

[47] Hedrich, L., and Barke, E. A formal approach to nonlinear analog circuit verification. In *International Conference on Computer-Aided Design* (1995), pp. 123–127.

[48] Hedrich, L., and Barke, E. A formal approach to verification of linear analog circuits with parameter tolerances. In *Design, Automation and Test in Europe Proceedings* (1998), pp. 649–654.

[49] Henzinger, T. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, New Jersey, 1996), pp. 278–292.

[50] Henzinger, T., Nicollin, X., Sifakis, J., and Yovine, S. Symbolic model checking for real-time systems. In *7th. Symposium of Logics in Computer Science* (Santa-Cruz, California, 1992), IEEE Computer Scienty Press, pp. 394–406.

[51] Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer 1*, 1–2 (1997), 110–122.

[52] IBS Corporation. Industry reports, 2003.

[53] Kurshan, R. P., and McMillan, K. L. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design 10*, 11 (Nov. 1991), 1356–1371.

[54] Litte, S., Walter, D., Myers, C., and Yoneda, T. Verification of analog and mixed-signal circuits using timed hybrid petri nets. In *Second International Symposium on Automated Technology for Verification and Analysis* (2004), pp. 426–440.

[55] Little, S., Seegmiller, N., Walter, D., and Myers, C. J. Verification of analog/mixed-signal circuits using labeled hybrid petri nets. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 2006), pp. 275–282.

[56] Miné, A. The octagon abstract domain. In *Analyzing, Slicing, and Transformation* (Oct. 2001), IEEE Computer Society Press, pp. 310–319.

[57] Myers, C. *Asynchronous Circuit Design*. Wiley, 2001.

[58] Myers, C. J., Harrison, R. R., Walter, D., Seegmiller, N., and Little, S. The case for analog circuit verification. In *The Workshop on Formal Verification of Analog Circuits* (Apr. 2005).

[59] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Abstract DPLL and Abstract DPLL Modulo Theories. In *11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'04* (2004), F. Baader and A. Voronkov, Eds., vol. 3452 of *Lecture Notes in Computer Science*, Springer, pp. 36–50.

[60] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM 53*, 6 (Nov. 2006), 937–977.

[61] Pastor, E., Roig, O., Cortadella, J., and Badia, R. M. Petri net analysis using boolean manipulation. In *Proc. of the 15th Int. Conf. on Application and Theory of Petri Nets (PNPM'94), Zaragosa, Spain* (June 1994), R. Valette, Ed., LNCS 815, Springer, pp. 416–435.

[62] Ranise, S., and Tinelli, C. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[63] Ranise, S., and Tinelli, C. The SMT-LIB Standard: Version 1.2. Tech. rep., Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[64] Seshia, S. A., and Bryant, R. E. A boolean approach to unbounded, fully symbolic model checking of timed automata. Tech. Rep. CMU-CS-03-210, CMU Computer Science Department, Dec. 2003.

[65] Seshia, S. A., and Bryant, R. E. Unbounded, fully symbolic model checking of timed automata using boolean methods. In *Proc. International Workshop on Computer Aided Verification* (2003), pp. 154–166.

[66] Silva, B. I., and Krogh, B. H. Formal verification of hybrid systems using CheckMate: A case study. In *American Control Conference* (June 2000), vol. 3, pp. 1679–1683.

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

[68] Somenzi, F. CUDD: CU Decision Diagram package release 2.4.1. Web page, March 2007. http://vlsi.colorado.edu/ fabio/CUDD/.

[69] Tuffin, B., Chen, D. S., and Trivedi, K. S. Comparison of hybrid systems and fluid stochastic petri nets. *Discrete Event Dynamic Systems: Theory and Applications 11* (Jan. 2001), 77–95.

[70] Wang, F. Symbolic parametric safety analysis of linear hybrid systems with bdd-like date-structures. In *Proc. International Workshop on Computer Aided Verification* (2004), pp. 295–307.

[71] Zheng, H. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.

[72] Zheng, H., Myers, C. J., Walter, D., Little, S., and Yoneda, T. Verification of timed circuits with failure directed abstractions. In *ICCD* (2003), IEEE Computer Society, pp. 28–35.