HIOA - A Specification Language for Hybrid Input/Output Automata

A Thesis

Submitted For the Degree of

Master of Science (Engineering)

in the Faculty of Engineering

by

Sayan Mitra

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JULY 2001

Acknowledgements

I deeply acknowledge the guidance provided by my research supervisor professor L.M. Patnaik. The freedom he offered in my pursuits allowed me to learn a lot about research. For this, and for painstakingly reading and commenting on all stages of my work, in spite of his busy schedule, I express my gratitude to professor Patnaik.

I thank the Indian Institute of Science(IISc) for providing an excellent environment for research and for the financial support.

During the course of my research I have been fortunate to have been able to discuss my work with professor Nancy Lynch of MIT. I am grateful to her for the comments and suggestions she gave.

I thank professor Tapan Kumar Ghoshal of Jadavpur University for his encouragements.

I would like to thank my Lab mates Baligar, Guru and Avneet for their encouragements, suggestions and feedbacks, MBK(Bharat) for sharing his views and experiences. I also thank all the staff members of the Microprocessor Applications Lab(MAL) and the Computer Science department for their help and cooperation.

I thank the Aeronautical Development Establishment, Bangalore for providing me with the data for one of the case studies presented in this thesis. In particular, I would like to thank Mr. Abdul Hamid and Mr. Ramakrishna for sparing their time to discuss with me.

Apart from courses, lectures and research, my stay at IISc has been fulfilling on account of having the pleasurable company of some truly extraordinary individuals, I would like to thank Sanjay, Ambi, Sameer, Nitin, Vinodh, Guru, Jaya, Anindya, Barda, Anartada and Niluda for enriching my being at IISc.

I thank my parents and my sister for their patience, encouragement, support, and the privilege they have given, which enabled me to pursue research.

Contents

List of Tables

List of Figures

Abstract

Various formal techniques have been developed for verifying hybrid systems but no single method provides a complete and adequate solution without restricting the class of systems to which it is applicable. It is accepted that successful adaptation of formal techniques in analyzing realistic problems can only follow the development and integration of suitable analysis tools and specification languages. Our main contribution is HIOA - a specification language for hybrid systems based on the hybrid input/output automaton model. We introduce the notion of activities for describing continuous behavior. Activities are syntactically similar to the discrete actions but unlike the *locations* of hybrid automata, more than one of them can be simultaneously operating, which gives flexibility to the user for developing the specifications. The conditions for the actions and the activities are written in first order logic and the evolutions of the continuous variables are expressed in terms of algebraic and differential equations. The HIOA front end checks the input specification for syntactic and semantic correctness, and generates an intermediate description of the automaton which can be used by other analyzing tools. Usually a complex hybrid system is described by several interacting component automata, whereas most analytical methods deal with primitive automata. So we have designed a *composer* for transforming composite HIOA specifications into equivalent primitive forms.

Deductive techniques like invariant assertion and induction over length of execution, are used to prove properties of hybrid I/O automata and these methods can be effectively applied using mechanical provers like PVS(Prototype Verification System). We have developed a scheme for generating PVS theories from HIOA specifications. The generated theories specify the system, and the PVS prover is used to derive the properties of the system from these specifications. We believe that automatic generation of the theories would expedite the whole process of verification by relieving the users from the routine work involved in writing specifications. We have tested HIOA by specifying representative hybrid systems given as functional block diagrams and MMT-specifications.

Chapter 1

Introduction

Developments in microelectronics and fabrication technologies have led to smaller and cheaper digital devices. Miniaturization and many-fold improvement in the performance of these devices have ushered a new paradigm in their application - *embedded systems*. Unlike the isolated monolithic computers of the past generation, today electronic devices are being increasingly used to control and interact with the environment in which they are embedded. Common examples abound in avionics, process control, robotics and consumer products.

1.1 Hybrid Systems

A physical plant or an environment is a *continuous time* system while the digital device acting on it, typically implements a *discrete time* control algorithm. The combined system, of a physical process being controlled by algorithms implemented in software, is one whose behavior exhibits both discrete and continuous changes. The combination of such a discrete program with an analog environment is called a *hybrid system*.

In recent years there has been widespread interest in hybrid systems, particularly in their modeling, verification and control. While the continuous evolution of variables has a unified description formalism in the form of algebraic and differential equations, there exists a diverse variety of formalisms for modeling discrete changes, most of which are derivatives of state transition models. Analysis of hybrid systems is possible through modeling which can capture both discrete and continuous behavior. By its very nature, hybrid systems is a subject of multi-disciplinary research, involving topics in the areas of system theory and computer science. Researchers in the area of control and system theory tend to view hybrid systems as a special class of dynamical systems with both real and boolean variables, while computer scientists have approached hybrid systems, by extending *formal techniques* which have proved to be effective in the analysis of discrete systems.

In this introductory chapter, first we discuss the rationale behind using formal methods in system design, then we present an overview of related research work in modeling and verification of hybrid systems. We discuss the current research trends which attempt to apply the theoretical developments in formal methods to practical industrial problems and in this light we outline the major goals undertaken in the research work presented in this thesis.

1.2 Formal Methods

Most of the models and methods developed by computer scientists for hybrid systems are derived from those which proved to be useful in the analysis of discrete systems. Formal techniques have been successful and hold promise as a methodology for system development for three main reasons. They

- detect design flaws at an early stage - cut costs by eliminating unproductive design cycles
- increase confidence in the reliability of the system - provably "correct" system design
- give a deeper understanding of the system by clarifying customer requirements and implementation tasks leading to precise and unambiguous specifications

The term *formal method* encompasses all approaches to specification and verification based on mathematical formalisms. Formal techniques are used to specify a system, check whether a specification is realizable, model the system, verify that an implementation satisfies its specification, prove properties of the system without necessarily executing the system. Applying formal methods for system design has two broad aspects; (1)*specification* describes the system and its desired properties, using a language with mathematically defined syntax and semantics, and (2)*verification* checks whether a given implementation of the system satisfies the required properties.

One area where formal methods have had major impact is hardware design. Companies like Intel have started using *model checking*, in complement with traditional design techniques like simulation, as a standard technique for detecting errors in hardware design. Some limited progress has also been made in applying formal methods to software development. Stringent reliability requirements of safety-critical software such as the NASA guideline [2] of less than 10^{-10} critical failures per hour of operation of aircraft flight control systems, has led to formal methods being included in an increasing number of software standards [3]. A detailed survey of applications of formal methods in specifying and verifying both hardware and software systems can be found in [4].

In the following section we briefly look at the formal models and the related verification techniques which have been developed for analysis of hybrid systems.

1.3 Formal Models for Hybrid Systems

The first extensions of discrete models towards incorporating continuous behavior considered real-numbered time. One such model is the timed automaton[5] – a finite automaton augmented with a finite number of real-valued clocks .The *hybrid automaton*[1, 6], is an extension of the timed automaton that allows continuous variables with more general dynamic behavior than clocks . Formalizations and terminology of hybrid automata vary to some extent, but in the most common form a hybrid automaton is a finite state automaton(states are called locations) with a set of real-valued variables associated with it. The hybrid automaton model for a thermostat is shown in Figure 1.1. In each location, the

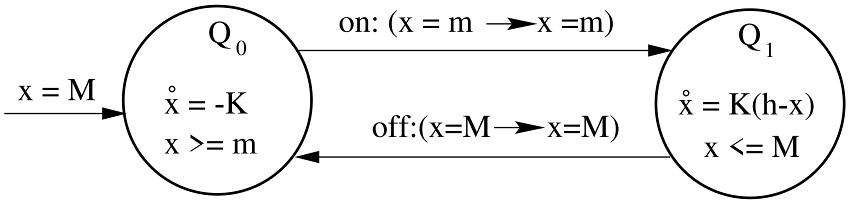

Figure 1.1: Hybrid automaton for thermostat

variables evolve continuously according to the rules for that location, stated by differential equations or rates. These continuous changes of the variables are referred to as the *flows* or the *trajectories* of the hybrid automaton. When some condition(guard) on the variables is satisfied, then an instantaneous transition may occur; if this happens in conjunction with the violation of one of the conditions (called *invariants*) then the transition *must* occur. An instantaneous transition is also called a *jump* or an *action*. In Figure 1.1 the jumps are shown by arrows between the locations with guards and assignments on their labels. The flows and the invariants are written in the respective states. If there are two or more automaton components modeling a complex system, then they are synchronized by the occurrence of their common actions.

The Hybrid Input/Output Automaton [7, 8] is another model for hybrid systems. This model is derived from the Lynch-Vaadrager model of timed automaton [9] and the phase transition model [10]. Unlike the hybrid automaton model[1], hybrid I/O automata components can communicate and synchronize via shared jumps as well as shared flows. The external interface is an integral part of each automaton component, defined in terms of input, output and internal variables and actions. The hybrid I/O automaton model is discussed in Chapter 2. An subclass of hybrid I/O automata can be conveniently specified by the MMT notation introduced by Merritt, Modugno and Tuttle[11].

A survey and classification of other, less commonly employed formalisms for specifying hybrid systems can be found in [12].

1.4 Verification Techniques

Verification is the process of finding if a given implementation satisfies the specifications of a system, that is checking if all possible behaviors of the implemented system are included in those allowed by the specifications. Behavorial specifications to be verified are usually *safety properties* which state that a certain predicate over the state space always holds. If the safety requirements are linear convex predicates then they can also be interpreted as a union of polytopes in the state space of the system.

There are two basic approaches to verification: algorithmic and deductive. For finite state systems and a restricted class of infinite state systems, verification of behavorial properties is decidable; that is, algorithms can be devised which would determine if a given implementation satisfies the specification. For a vast majority of infinite state systems including most hybrid systems, no such algorithm exists; the problem is undecidable and verification is based on deduction and heuristics.

1.4.1 Model Checking

Model checking is a method for automatic algorithmic verification by exhaustive exploration of state space. The search starts with a set of possible initial states and a finite union of convex polytopes corresponding to the safety property to be verified. If all the trajectories emanating from the initial states are enclosed within the polytope, then the safety condition is satisfied; else it is not. In the latter case, a counter-example is generated by the algorithm. Inclusion of real-valued variables in the hybrid automaton model makes its state space infinite and exhaustive search is not possible. This is an inherent problem of model checking and is known as the *state space explosion* problem. A possible solution to this problem is offered by *symbolic* model checking [13, 1], in which a set of states are abstracted to a *region* and further manipulations are performed on the regions instead of individual states. Symbolic model checking is possible for a subclass of hybrid automata called *linear* hybrid automata, where the flows are governed by linear differential equations and the guards and invariants represent linear convex predicates. But even for this restricted class, the fixpoint computation is not guaranteed to terminate.

Exact analysis of non-linear hybrid systems is extremely difficult at best and impossible in many cases, so they are approximated by linear hybrid automata[14]. Significant improvements can be made by approximating the regions by their convex-hulls and by enforcing the convergence of the iterative computation by extrapolation [15].
The most important tool for analyzing hybrid systems so far is **HyTech** [16]. It is a symbolic model checker for linear hybrid automata, it also allows parametric analysis, approximates non-linear automata and generates error traces. **Shift** [17] provides a simulation environment for generalized hybrid automata.

1.4.2 Theorem Proving

A second approach to verification is to use deductive techniques. In this approach the model of the system provides formal semantics for composition, abstraction, etc. and supports proof techniques such as induction on the length of executions, invariant assertions and simulation relations. In this approach the properties of the system are specified in standard logic with existential and universal quantifiers, making the notation simpler compared to model checking (which typically uses some variant of temporal logic CTL, ICTL[13], TCTL). More importantly, these techniques are free from the state space explosion problem. It has also been claimed[18] that higher/more abstract level of reasoning allows more powerful results to be proved and non-linearity to be handled more effectively. The hybrid I/O automaton model has been used in verifying a number of benchmark problems using deductive techniques [19, 20, 21].

However, unlike model checking which is automatic, deductive techniques involve interaction of the user with the mechanical prover, and this demands a certain level of skill on the part of the user. The fact that training personnel in theorem proving is expensive, poses an impediment for deductive methods from becoming widely applicable to industrial problems. This has led to the development of tools like **TAME**[22, 23] which is a customized interface for a particular framework(timed automaton) over a general purpose theorem prover (PVS). TAME helps the user to interact with the theorem prover by partially automating the process of specification and verification by utilizing templates and built-in proof strategies.

1.5 The Thesis

Even though significant theoretical advances have been made in specifying and verifying hybrid systems, application of formal methods in design and analysis of real industrial systems remains yet uncommon. It is believed that, suitable languages and tools are essential for system designers to derive benefit from these theoretical developments. In this context we mention the following important directions pointed out in [4] and [24] for making formal methods accessible and useful to system designers:

1. No single method provides a complete and adequate solution for the verification problem without restricting the class of systems to which it is applicable, hence an engineering approach has to be taken, preparing a suite of analysis tools and integrating methods.

2. Tools and specification languages should be easy to learn and use and their outputs easy to understand so as to minimize the effort and expertise needed to apply a method.

The HIOA(Hybrid Input Output Automaton) language, is our effort to provide an expressive and easy to learn specification language for hybrid systems, and for integrating analytical tools. HIOA is based on the hybrid I/O automata model[7]. The language primitives provide mechanisms for describing discrete and continuous changes. We introduce the notion of *activities* for describing continuous behavior. The activities are syntactically similar to the discrete transitions, but unlike the *locations* of hybrid automata [1], more than one activity can operate simultaneously, which gives the user more flexibility in developing the model.

We propose a *workbench* of tools for analyzing hybrid systems integrated around HIOA. The architecture of the workbench (Figure 1.2) is similar to that of the IOA Toolkit [25]. Language centric design reduces the translation efforts between languages and tools, and as the different tools are decoupled, they can be developed independently.

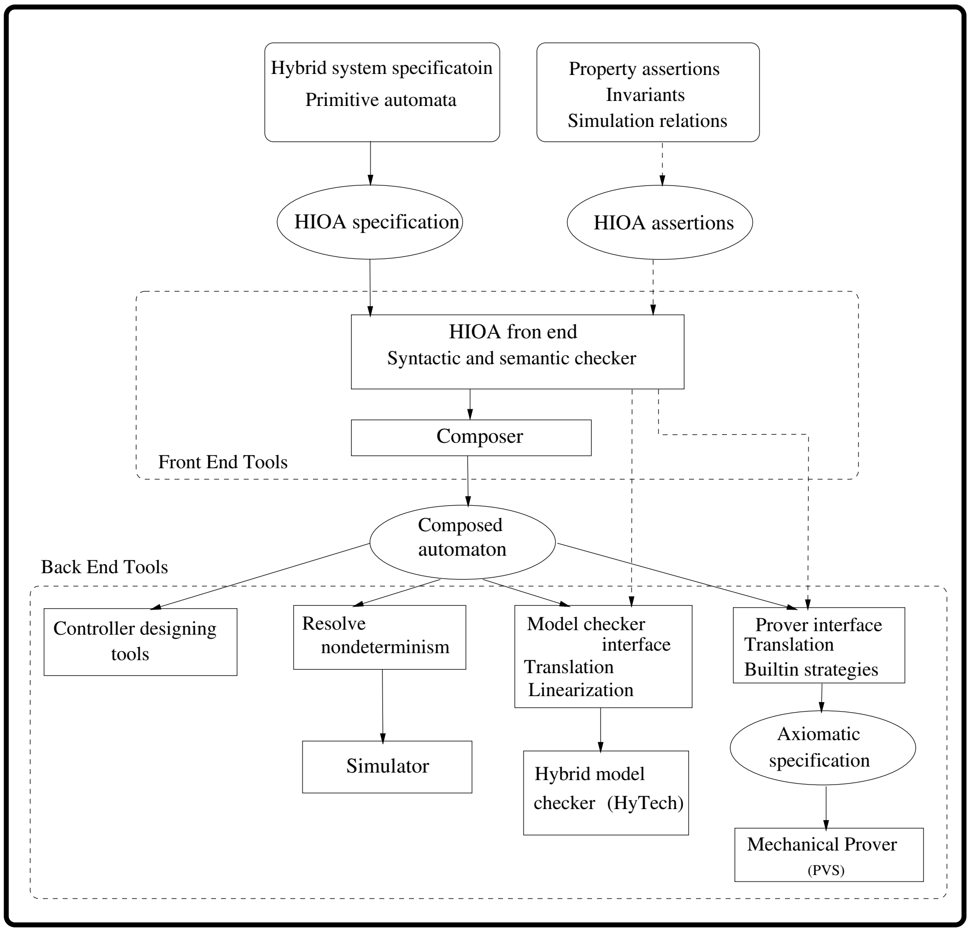

Figure 1.2: Proposed hybrid workbench

The specification of complex hybrid systems consist of several interacting hybrid I/O automata[7] each of which are separately specified in HIOA. The HIOA front end checks the input specification for syntactic and semantic correctness and generates a description of the system in an Intermediate Language(IL) which is used by the back end analysis tools. The composer puts together interacting components and produces an equivalent primitive automaton describing the whole system.

Separate interfaces and translators are to be built for various analysis tools which are already existing and also for those which shall be developed in the future. In this thesis we propose an interface to the Prototype Verification System(PVS), which would automatically convert HIOA specifications to PVS theories. PVS is a mechanical theorem prover for high order logic. Automatic translation of HIOA specifications to PVS would enable users to employ the powerful PVS prover for deducing properties of the system, without having to write the PVS specifications manually.

1.6 Organization of the Thesis

The rest of the thesis is organized into chapters as follows.

In **Chapter 2** we present the hybrid I/O automata model, which is the underlying mathematical model of the HIOA language.

In **Chapter 3** we discuss the language design issues and introduce the concept of *activities* for describing continuous behavior.

In **Chapter 4** we give a detailed description of the HIOA language.

In **Chapter 5** we describe the front end tools supporting HIOA. An overview of the implementation of the checker is given followed by a description of the intermediate language and the HIOA composer.

In **Chapter 6** we verify a leader election algorithm for a synchronous ring using PVS. This study serves as an introduction to the process of specification and verification in PVS.

In **Chapter 7** we develop a template theory in PVS for specifying HIOA components and other theories for defining hybrid executions.

In **Chapter 8** we evaluate the features provided by HIOA by studying the specification of (1) longitudinal axis controller of an aircraft and (2) deceleration control of a vehicle.

In **Chapter 9** we conclude with a discussion on the contributions made in this thesis and the directions along which this work can be extended.

Appendix A gives the BNF grammar of the HIOA language.

Appendix B gives the BNF grammar of the intermediate language(IL).

Appendix C gives the PVS theory for hybrid executions.

Chapter 2

Mathematical Model

2.1 Introduction

In this Chapter we present the *hybrid input output automaton* which is the underlying mathematical model of our specification language. The hybrid I/O automaton model [7, 8] is derived from Lynch-Vaadrager model of timed automaton [9] and the phase transition model[10]. It is an automaton model for description and analysis of discrete and continuous behavior of hybrid systems. This model specializes hybrid automata by an additional distinction between input and output, also the notion of trajectories is a primitive in this model. Some aspects of this model are still under development, our language design is based on the version of the model presented in [8].

2.2 Terminology

First of all, we introduce certain terms which are used for defining hybrid automata and their properties. Throughout the rest of this chapter, we assume *time axis* T to be a compact subgroup of $(\mathsf{R}, +)$, i.e., the real numbers with addition.

2.2.1 Static and Dynamic Types

The system consists of a universal set V of *variables*. Every variable is attributed with a *(static) type*, which gives the set of values it may assume, and a *dynamic type*, which gives the set of *trajectories* that it may follow. Since in a hybrid system a discrete transition can change the state at any time, elements of a dynamic type may contain (countably many) "discontinuities". Formally, each variable v has:

- $type(v)$, the *(static) type* of v, the set of values v can assume.
- $dtype(v)$, the *dynamic type* of v, the set of functions from left-closed intervals of T to $type(v)$ that is closed under *time shift, subinterval* and *pasting* [8].

2.2.2 Trajectories

The evolution of the continuous variables or the *flows* of the system are described as *trajectories* in the hybrid automaton model. Given V, a set of variables, a *valuation* $\mathbf{v}$ of V is a function that assigns to each variable $v \in V$ a value in $type(v)$. $val(V)$ is the set of all valuations for V. Let J be a left-closed interval of T with left endpoint equal to 0. Then a *J-trajectory* for V is a function $\tau : J \rightarrow val(V)$, such that for each $v \in V$, $\tau \downarrow v \in dtype(v)$. Here $\tau \downarrow v$ is the function with domain J defined by $(\tau \downarrow v)(t) = \tau(t)(v)$. A J-trajectory is *finite*(*closed*) if J is a finite(closed) interval, and *full* if $J = \mathsf{T}^{\geq 0}$.

2.2.3 Operations on Trajectories

The $\tau.ltime$, the *limit time* of trajectory τ, is the supremum of $dom(\tau)$. Similarly, $\tau.fval$, the *first valuation* of τ, is defined to be $\tau(0)$, and if τ is closed, then $\tau.lval$, the *last valuation* of τ, is defined to be $\tau(\tau.ltime)$.

For a trajectory τ, and $t \in \mathsf{T}^{\geq 0}$, $\tau \trianglelefteq t \triangleq \tau \lceil [0,t]$,$\tau \triangleleft t \triangleq \tau \lceil [0,t)$, and $\tau \trianglerighteq t \triangleq (\tau \lceil [t,\infty)) - t$. Note that the result of applying the above operations is always a trajectory, except when the result is a function with an empty domain. By convention, $\tau \trianglelefteq \infty \triangleq \tau$ and $\tau \triangleleft \infty \triangleq \tau$.

Trajectory τ is a *prefix* of trajectory υ, denoted by $\tau \leq \upsilon$, if τ can be obtained by restricting υ to a non-empty, downward closed subset of its domain.Formally, $\tau \leq \upsilon$ iff $\tau = \upsilon \lceil dom(\tau)$.

The concatenation of two trajectories is obtained by taking the union of the first trajectory and the function obtained by shifting the domain of the second trajectory until the start time agrees with the limit time of the first trajectory; the last valuation of the first trajectory, which may not be the same as the first valuation of the second trajectory, is the one that appears in the concatenation. Formally, let τ, υ be trajectories, with τ closed. Then the *concatenation* is the function given by $\tau ^\frown \upsilon \triangleq \tau \cup (\upsilon \lceil (0, \infty) + \tau.ltime)$.

2.2.4 Hybrid Sequences

A *hybrid sequence* is used to model a combination of changes that occur instantaneously and changes that occur over intervals of time. In the following definitions, A is a set of *actions*, which are used to model instantaneous changes and instantaneous synchronization with the environment, and V is a set *variables*, which are used to model continuous changes over intervals and continuous interaction.

An (A, V)-*hybrid sequence* is a finite or infinite alternating sequence $\zeta = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$, where (1) each τ_i is a trajectory in $trajs(V)$, (2) each a_i is an action in A, (3) if ζ is a finite sequence, then it ends with a trajectory, and (4) if τ_i is not the last trajectory in ζ, then $dom(\tau_i)$ is closed.

2.2.5 Operations on Sequences

If ζ is a hybrid sequence, then its *first valuation* $\zeta.fval$ is $\tau_0.fval$, and its *limit time* $\zeta.ltime$, is $\sum_i \tau_i.ltime$.

An (A, V)-sequence $\zeta = \tau_0\, a_1\, \tau_1 \ldots$ is a *prefix* of (A, V)-sequence $\zeta' = \tau'_0\, a'_1\, \tau'_1 \ldots$, denoted by $\zeta \leq \zeta'$, if either $\zeta = \zeta'$, or ζ is a finite sequence ending in some τ_k; $\tau_i = \tau'_i$, and $a_{i+1} = a'_{i+1}$ for every i, $0 \leq i < k$; and $\tau_k \leq \tau'_k$.

Suppose ζ and ζ' are (A, V)-sequences, with ζ closed. Then the *concatenation* is the (A, V)-sequence given by $\zeta \frown \zeta' \triangleq init(\zeta)\ (last(\zeta) \frown head(\zeta'))\ tail(\zeta')$.

2.3 Hybrid Automata

Here we give the definition of hybrid automata which is a more general model than the hybrid I/O automata. Hybrid automata classify actions and variables as external and internal, but do not further subdivide the external actions and variables into input and output.

2.3.1 Definition of Hybrid Automata

A *hybrid automaton (HA)* $\mathcal{A} = (W, X, \Theta, E, H, \mathcal{D}, \mathcal{T})$ consists of:

- A set W of *external variables* and a set X of *internal variables*, $W \bigcap X = \phi$. $V \triangleq W \cup X$.
- A nonempty set $\Theta \subseteq val(X)$ of *start states*.
- A set E of *external actions* and a set H of *internal actions*, $E \bigcap H = \phi$. $A \triangleq E \cup H$.
- A set $\mathcal{D} \subseteq val(X) \times A \times val(X)$ of *discrete transitions*. $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ is used as a shorthand for $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$.
- A set $\mathcal{T}$ of trajectories for V.

A valuation $\mathbf{x}$ for X determines a *state* of the automaton, and the set of all valuations of X, which is written as $val(X)$ is the set of states of $\mathcal{A}$. $V \triangleq W \cup X$. Given a trajectory $\tau \in \mathcal{T}$, $\tau.fstate \triangleq \tau.fval \lceil X$, and if τ is closed then $\tau.lstate \triangleq \tau.lval \lceil X$.

2.3.2 Executions, Traces and Comparability

An *execution fragment* of a HA $\mathcal{A}$ is an (A, V)-sequence $\zeta = \tau_0\ a_1\ \tau_1\ a_2\ \tau_2 \cdots$, where (1) each τ_i is a trajectory in $\mathcal{T}$, and (2) if τ_i is not the last trajectory in ζ, then

$\tau_i.lstate \overset{a_{i+1}}{\rightarrow} \tau_{i+1}.fstate$. An execution fragment records all the instantaneous, discrete state changes that occur during a specific evolution of a system, as well as the state changes and external variable changes that occur while time advances.

The *first state* of ζ, $\zeta.fstate$, is $state(\zeta.fval)$, or equivalently, $\tau_0.fstate$. An execution fragment ζ is an *execution* if $\zeta.fstate$ is a start state. If ζ is a closed execution fragment, then *last state* of ζ, $\zeta.lstate$, is $state(\zeta.lval)$, or equivalently, $last(\zeta).lstate$. A state of $\mathcal{A}$ is *reachable* if it is the last state of some closed execution of $\mathcal{A}$.

A trace is the externally visible part of an execution fragment i.e., it records the external actions and the evolution of external variables. The *trace* of an automaton $\mathcal{A}$ is a trace arising from the execution of $\mathcal{A}$. The set of traces of $\mathcal{A}$ is denoted by $traces_{\mathcal{A}}$.

Hybrid automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if they have the same external actions and variables, that is, if $W_1 = W_2$ and $E_1 = E_2$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, then we say that $\mathcal{A}_1$ *implements* A_2, denoted by $\mathcal{A}_1 \leq \mathcal{A}_2$, if the traces of $\mathcal{A}_1$ are included among those of $\mathcal{A}_2$; that is, if $traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$. Inclusion of sets of hybrid traces is the *implementation relation* of this model.

Simulation relations between hybrid automata are used to show that one automaton implements another. Let $\mathcal{A}$ and $\mathcal{B}$ be comparable hybrid automata. A *simulation* from $\mathcal{A}$ to $\mathcal{B}$ is a relation $R \subseteq val(X_{\mathcal{A}}) \times val(X_{\mathcal{B}})$ satisfying the following conditions, for all states $\mathbf{x}_A$ and $\mathbf{x}_B$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. If $\mathbf{x}_A \in \Theta_A$ then there exists a state $\mathbf{x}_B \in \Theta_B$ such that $\mathbf{x}_A \; R \; \mathbf{x}_B$.

2. If $\mathbf{x}_A \; R \; \mathbf{x}_B$, $\mathbf{x}_A \overset{a}{\rightarrow}_A \mathbf{x}'_A$ and $\tau = trace(\wp(\mathbf{x}_A) \, a \, \wp(\mathbf{x}'_A))$, then $\mathcal{B}$ has a closed execution fragment ζ with $\zeta.fstate = \mathbf{x}_B$, $trace(\zeta) = trace(\tau)$, and $\mathbf{x}'_A \; R \; \zeta.lstate$.

3. If $\mathbf{x}_A \; R \; \mathbf{x}_B$ and τ is a closed trajectory of $\mathcal{A}$ with $\mathbf{x}_A = \tau.fstate$ and $\mathbf{x}'_A = \tau.lstate$, then $\mathcal{B}$ has a closed execution fragment ζ with $\zeta.fstate = \mathbf{x}_B$, $trace(\zeta) = trace(\tau)$,

and $\mathbf{x}'_A \; R \; \zeta.lstate$.

It can be shown that, sates of two automata related by a simulation relationship have the same valuation of the external variables.

2.3.3 Composition

Composition allows an automaton representing a complex system to be constructed by putting together automata representing individual system components. The composition operation identifies actions and variables with the same name in different component automata. When any component automaton performs a step involving an action a, so do all component automata having action a. Similarly common variables are shared among the components.

Composition is defined as a partial, binary operation on hybrid automata. Since internal actions of an automaton $\mathcal{A}_1$ are intended to be unobservable by any other automaton $\mathcal{A}_2$, so they can be composed only if their internal actions are disjoint. Also, the internal variables of $\mathcal{A}_1$ should be disjoint from the set of variables of $\mathcal{A}_2$. Formally, we say that hybrid automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if for $i, j \in \{1, 2\}$ and $i \neq j$, $X_i \cap V_j = H_i \cap A_j = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible, then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the structure $\mathcal{A} = (W, X, \Theta, E, H, \mathcal{D}, \mathcal{T})$ where

- $W = W_1 \cup W_2$, $X = X_1 \cup X_2$, $E = E_1 \cup E_2$, $H = H_1 \cup H_2$
- $\Theta = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in \Theta_1 \wedge \mathbf{x} \lceil X_2 \in \Theta_2\}$
- For each $\mathbf{x}, \mathbf{x}' \in val(X)$ and each $a \in A$, $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ iff for $i = 1, 2$, either (1) $a \in A_i$ and $\mathbf{x} \lceil X_i \xrightarrow{a}_i \mathbf{x}' \lceil X_i$, or (2) $a \notin A_i$ and $\mathbf{x} \lceil X_i = \mathbf{x}' \lceil X_i$.
- $\mathcal{T} \subseteq trajs(V)$ is given by $\tau \in \mathcal{T} \Leftrightarrow \tau \downarrow V_1 \in \mathcal{T}_1 \;\wedge\; \tau \downarrow V_2 \in \mathcal{T}_2$.

In the following section we specialize the hybrid automaton model of Section 2.3, by adding a distinction between input and output, amongst external variables and actions.

2.3.4 Definition of Hybrid I/O Automata

A *hybrid I/O automaton (HIOA)* $\mathcal{A}$ is a tuple $(\mathcal{H}, U, Y, I, O)$ where

- $\mathcal{H} = (W, X, \Theta, E, H, \mathcal{D}, \mathcal{T})$ is a hybrid automaton.
- U and Y partition W into *input* and *output* variables, respectively. Variables in $Z \triangleq X \cup Y$ are called *locally controlled*; as before we write $V \triangleq W \cup X$.
- I and O partition E into *input* and *output actions*, respectively. Actions in $L \triangleq H \cup O$ are called *locally controlled*; as before we write $A \triangleq E \cup H$.
- The following additional axioms are satisfied:

  Input action enabling

  For all $\mathbf{x} \in val(X)$ and all $a \in I$ there exists $\mathbf{x}'$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

  Input flow enabling

  For all $\mathbf{x} \in val(X)$ and $v \in trajs(U)$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x}, \tau \downarrow U \leq v$, and either

  1. $\tau \downarrow U = v$, or
  2. there exist $t \in dom(\tau)$ and $l \in L$ such that l is enabled from $\tau(t)$.

Input action enabling ensures that an automaton does not have control over the occurrence of its input actions. Input flow enabling says that an hybrid I/O automaton should be able to accept any continuous input flow, either by letting time advance for the entire duration of the input flow, or by reacting with a locally controlled action after some part of the input flow has occurred.

An *execution* of an hybrid I/O automaton $\mathcal{A}$ is an execution of $\mathcal{H}$. Similarly, a *trace* of $\mathcal{A}$ is a trace of $\mathcal{H}$. Two hybrid I/O automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if their inputs and outputs coincide, that is, if $I_1 = I_2$, $O_1 = O_2$, $U_1 = U_2$, and $Y_1 = Y_2$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, then $\mathcal{A}_1 \leq \mathcal{A}_2$ is defined to mean that the traces of $\mathcal{A}_1$ are included among those of $\mathcal{A}_2$: $\mathcal{A}_1 \leq \mathcal{A}_2 \triangleq traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable automata, then the corresponding hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ are comparable and $\mathcal{A}_1 \leq \mathcal{A}_2$ iff $\mathcal{H}_1 \leq \mathcal{H}_2$.

2.3.5 Compatibility

The definition of composition for HIOAs builds on the corresponding definition for hybrid automata, and also takes the input/output structure into account. Just as in the definition of compatibility in 2.3.3, hybrid I/O automaton $\mathcal{A}_1$ is not allowed to be composed with $\mathcal{A}_2$ unless the internal actions and variables of $\mathcal{A}_1$ are disjoint from the actions and variables, respectively, of $\mathcal{A}_2$. In addition, in order that the composition operation satisfies certain properties, it is required that at most one component automaton "controls" any given action or variable; that is, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are to be composed, then the sets of output actions of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint and also the sets of output variables of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint. To summarize, two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible if, for $i, j \in \{1, 2\}$ and $i \neq j$, the following conditions hold:

1. $X_i \cap V_j = \phi$
2. $Y_i \cap Y_j = \phi$
3. $O_i \cap O_j = \phi$
4. $H_i \cap A_j = \phi$

2.3.6 Composition

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible, then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the tuple $\mathcal{A} = (U, Y, X, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$, where

1. $U = (U_1 \cup U_2) - (Y_1 \cup Y_2)$,
2. $Y = Y_1 \cup Y_2$,
3. $X = X_1 \cup X_2$,
4. $\Theta = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in \Theta_1 \wedge \mathbf{x} \lceil X_2 \in \Theta_2\}$.
5. $I = (I_1 \cup I_2) - (O_1 \cup O_2)$,
6. $O = O_1 \cup O_2$.

7. $H = H_1 \cup H_2$,

8. For each $\mathbf{x}, \mathbf{x}' \in val(X)$ and each $a \in A$, $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ iff for $i = 1, 2$, either (1) $a \in A_i$ and $\mathbf{x} \lceil X_i \xrightarrow{a}_i \mathbf{x}' \lceil X_i$, or (2) $a \notin A_i$ and $\mathbf{x} \lceil X_i = \mathbf{x}' \lceil X_i$.

9. $\mathcal{T} \subseteq trajs(V)$ is given by $\tau \in \mathcal{T} \Leftrightarrow \tau \downarrow V_1 \in \mathcal{T}_1 \ \wedge \ \tau \downarrow V_2 \in \mathcal{T}_2$.

This definition of compatibility is not sufficient to ensure that the composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ is always a hybrid I/O automaton. Thus, a stronger notion of compatibility involving input flow enabling is introduced in [8].

2.4 Summary

In this chapter, we have given an overview of the theory of hybrid I/O automata. Definitions of *Hiding, Receptiveness* and the proofs of the important results are given in [7] and [8]. Based on the hybrid I/O automaton model, we have designed our specification language - HIOA. The guiding factors in the design of HIOA and the basic design issues are discussed in the next chapter.

Chapter 3

Language Design Issues

3.1 Introduction

The specification language is the central element of the workbench, in that it binds the analyzing tools in a common platform. *Hybrid Input Output Automata (HIOA)* is a formal language for defining hybrid I/O automata and stating their properties. The design of HIOA is derived from the *IOA language*[26] which is a Larch specification language for asynchronous concurrent systems. In the Larch style of writing specifications, the specification of a system consists of two parts, (1) the actual system description is written in an *interface language*, specially tailored for the underlying mathematical model of the system and (2) the system independent elements like data types and operators are specified in the *Larch Shared Language or LSL*, which is independent of any mathematical model.

HIOA is an interface language for specifying hybrid systems. HIOA extends a subset of the IOA language by introducing mechanisms to specify continuous change of variables. In this chapter we discuss the issues involved in the design of the HIOA language and in the next chapter we present the architecture of the complete language with examples.

3.2 Requirements of a Hybrid Specification Language

The effectiveness of any specification language is measured by its expressive power and the kind of tools which can exploit the specifications written in it. Whereas its practical utility is determined by the ease with which it can be learnt and used. We begin the discussion on the design of HIOA by considering a list of desirable properties which a specification language for hybrid systems should have. The following features have been identified as the main requirements, and they have influenced the design of HIOA.

1. The language should be easy to learn; expressive for writing both discrete and continuous behavior of hybrid systems.
2. The language should support the two commonly employed approaches of modeling
   - A hierarchical model is developed incrementally i.e., initially a coarse description is given and then, successively more refined descriptions are developed by adding constraints.
   - A modular/partitioned model describes the behavior of the system in different *operational modes* as distinct modules.
3. For for deriving properties of the system from its specifications, the language should be easily interfaced with analytical tools like
   - Mechanical theorem provers
   - Symbolic model checkers
   - Simulators

3.3 Language Primitives

In the hybrid input/output automata model, discrete and continuous changes of variables are described by *transitions* and *trajectories* respectively. The discrete transitions in the HIOA language are handled more or less in the same fashion as in the IOA language, whereas the trajectories being primitive types demand new language constructs.

3.3.1 Representing the Trajectories

The continuous evolution of variables in hybrid I/O automata is described in terms of trajectories. Consider a hybrid automaton $\mathcal{H}$ with variable set V, $val(V)$ denotes the set of all possible assignments or valuations to the variables in V. Thus $val(V)$ represents a multi-dimensional space, each point in which corresponds to a particular valuation of all the variables of $\mathcal{H}$; we call this the *variable space* of $\mathcal{H}$. A trajectory τ of $\mathcal{H}$ (section 2.2.2) is a function $\tau : J \rightarrow val(V)$, where J is a left-closed interval of T . The trajectory τ, describes the continuous behavior of the variables in V over the interval J in terms of a *progress function*. A progress function gives the valuation of the locally controlled variables as algebraic definitions of the form $v_i = F_{i,\tau}(V)$ or in terms of first order ordinary differential equations like $\dot{v_i} = F_{i,\tau}(V)$. The functions $F_{i,\tau}(V)$ on the right hand sides of these assignments are either algebraic terms or nondeterministic *choice* expressions.

3.3.2 Organizing the Trajectories : Activities

As the continuous variables evolve differently in different regions of the variable space, each region is represented by an *activity*, consisting of an *activation condition* defining the limits of the region and a *progress function* describing the continuous changes of the local variables in the region.

> **Definition** An *activity* α is defined as a mapping $P \rightarrow E$, where P, the *activation condition* is a predicate over V and E is a progress function for the locally controlled variables.

This means that the evolution of the continuous variables is governed by the progress function E over a region $[P]$ of the variable space defined by the activation condition P. The activity α is said to be *operating* in the region $[P]$.

The complete continuous nature of the automaton is described by a set of activities. Let us assume that the hybrid automaton $\mathcal{H}$ is described by a set of activities $\mathcal{A}= \{\alpha_1, \ldots, \alpha_n\}$, where each activity $\alpha_i : P_i \rightarrow E_i$.

Simple Activities

If the activation conditions (P_i) of these activities are mutually exclusive, as shown in Figure 3.1, then the set $\mathcal{A}$ is said to be *simple*. In case of a simple set of activities, each of its member activities (α_i) describes the evolution of local variables in a particular part of the variable space ($[P_i]$) independent of other activities and the set of activities satisfies the condition :

$$\forall\, i, j \;\; i \neq j \Rightarrow [P_{\alpha_i}] \cap [P_{\alpha_j}] = \Phi$$

Such a scenario is typical of *multi-mode* control systems where the dynamics of the system in a particular *mode* is governed by a set of differential equations and the modes change by discrete jumps. The disjoint regions of a set of simple activities correspond to locations of the hybrid automaton model proposed by Alur et al. [1].

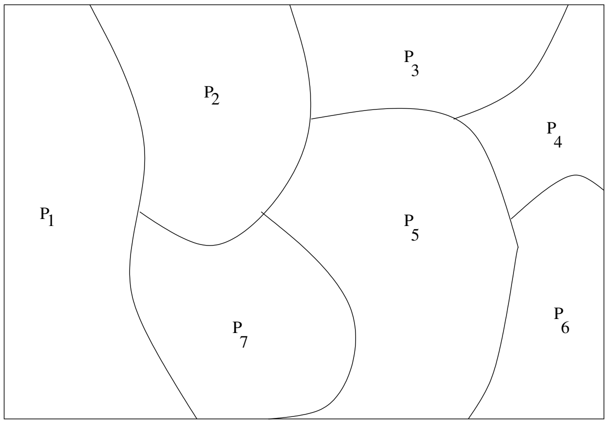

Figure 3.1: Disjoint activities

Complex Activities

In general however, the activation conditions need not be disjoint, and if $\exists\, i, j \;\; i \neq j$ such that $[P_{\alpha_i}] \cap [P_{\alpha_j}] \neq \phi$, then the set of activities $\mathcal{A}$ is said to be *complex*. Over a particular region in the variable space, more than one activation condition could be satisfied. Consequently, in such regions more than one progress function would define the continuous changes. As seen in Figure 3.2, the activities α_1, α_2 and α_3 are operating in the innermost region, and hence the three operating progress functions in this region are E_1, E_2 and E_3.

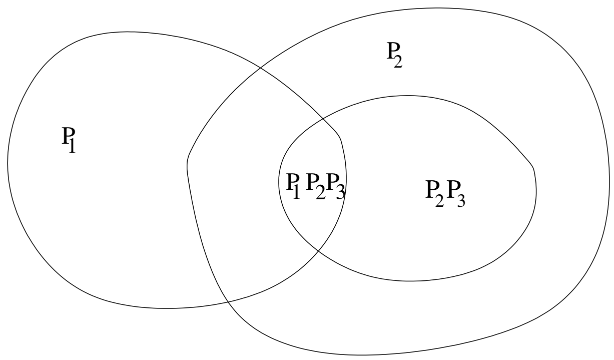

Figure 3.2: Overlapping activities

Overlapping activities and nondeterminism in the progress functions allow the user to develop the model of the system incrementally by writing an initial coarse description and then adding more precise activities to make the model adequately refined.

3.3.3 Simplifying Complex Activities

Provision for overlapping activities gives the user more freedom to develop the model, he need not take extra effort in ensuring that the activation conditions of all the activities are disjoint. However, for the purpose of analysis, in many cases, it is easier to deal with simple activities. Hence we devise methods to simplify a complex set of activities.

Form Figure 3.2 it is clear that the entire variable space can be viewed as a union of several disjoint regions, P_1P_2', $P_1'P_2P_3$, $P_1'P_2P_3'$, $P_1P_2P_3'$ and $P_1P_2P_3$; each of which can be represented as an activity. In general if there are n activities comprising a complex set, then 2^n simple activities can be formed. The activation conditions of these new activities are obtained by conjugating the original activation conditions or their negations. The progress function of a resultant simplified activity is formed by introducing a nondeterministic choice between the progress functions of the corresponding complex activities.

For example, the complex set of activities α_1, α_2 and α_3 of Figure 3.2, can be equivalently represented by the following simple set:

$$\beta_1 : (P_1 \wedge \neg P_2) \rightarrow E_1$$
$$\beta_1 : (\neg P_1 \wedge P_2 \wedge P_3) \rightarrow (E_2 \mid E_3)$$
$$\beta_1 : (\neg P_1 \wedge P_2 \wedge \neg P_3) \rightarrow E_2$$
$$\beta_1 : (P_1 \wedge P_2 \wedge \neg P_3) \rightarrow (E_1 \mid E_2)$$
$$\beta_1 : (P_1 \wedge P_2 \wedge P_3) \rightarrow (E_1 \mid E_2 \mid E_3)$$

The notation $(P_i \wedge P_j) \rightarrow (E_i \mid E_j)$ means that the evolution of the continuous variables in the region defined by $(P_i \wedge P_j)$, is entirely guided by either E_i or E_j but not a combination of both.

3.4 Summary

In this chapter we first looked at the desirable properties of a specification language for hybrid systems. The evolution of the continuous variables in hybrid automata is described by the trajectories, we introduced the notion of activities for writing and organizing the trajectories. An activity comprises of (1) an activation condition, which is a predicate defining a region of the variable space over which the activity operates, and (2) a progress function, which is a set of algebraic and differential equations governing the evolution of the variables as long as the activity is operating. A set of activities could have disjoint or overlapping activation conditions. For the purpose of analysis, a set of overlapping activities can be converted to simple activities.

Having introduced these basic concepts and the terminology, in the next chapter we discuss the specification of hybrid I/O automata using HIOA.

Chapter 4

The HIOA Language

4.1 Introduction

HIOA - Hybrid Input Output Automaton is a specification language for hybrid systems. It is based on the hybrid I/O automaton model described in Chapter 2. In this chapter we present the structure of the language and discuss the features it provides. The syntactic and lexical details of the language are given in Appendix A. We begin with a typical hybrid system and develop its specifications in the HIOA language, illustrating the important features. Then we discuss the constructs of the language for specifying primitive and composed automata in detail.

4.2 Introduction to HIOA

The hybrid system we present here is a water level controller. The HIOA specification for this system is developed in three stages; first, we describe the physical system, then the hybrid automaton for the system is developed and finally the corresponding HIOA code is written.

4.2.1 A Water Level Controller

A schematic diagram of the water level controller is given in Figure 4.1. The water level $\mathcal{Y}$ in the tank is maintained by the *Controller* by operating the pump. When the pump is `On`, it adds water at a constant rate which raises the level at m units/sec in the absence of any leakage. The drain at the bottom of the tank removes water at a constant rate of n units/sec. The Controller reads the output of two sensors at levels u and l respectively. There exist certain constant time delays between the sensor outputs and the actuation of the pump, and these delays are lumped into two constant quantities ld and ud for the l and the u sensors respectively.

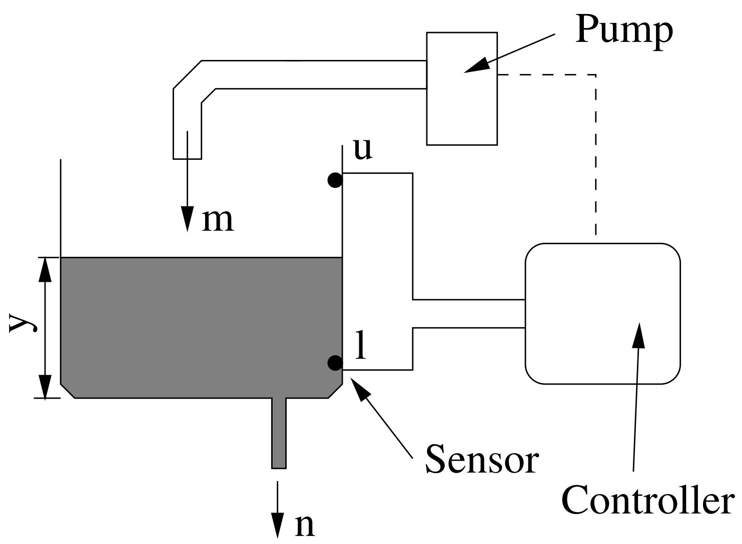

Figure 4.1: The water level controller

4.2.2 Hybrid Automaton Model of the *Controller*

The hybrid automaton model of the Controller is shown as a phase transition system in Figure 4.2. The model consists of two continuous variables; x, a stopwatch and y the tank level. The continuous evolution of these two variables is described in the four *locations*. At `On` and `Flood`, the pump is running; hence the level rises at a rate $\dot{y} = m - n$, while at `Off` and `Starve` the pump is stopped, and $\dot{y} = -n$. The variable x is a clock, so $\dot{x} = 1$

in all the four locations.

The automaton has four actions, of which *Udelay* and *Ldelay* correspond to the triggering of the u and l sensors respectively. The action *TurnOn*(*TurnOff*) corresponds to the actual switching `On`(`Off`) of the pump after the elapse of the delay period $dl(du)$. The delays are measured by the stopwatch x.

At the starting state of the automaton(indicated by the single arrow-head) the pump is `On` and the level y is anywhere between l and u.

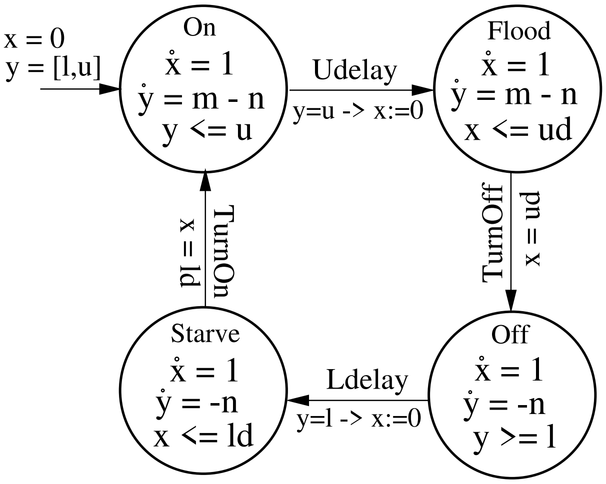

Figure 4.2: Controller automaton

4.2.3 HIOA Specification for Hybrid Automaton

For writing the HIOA specification from the phase transition diagram, the locations of the hybrid automaton are translated to simple activities by the relationship shown in Figure 4.3. Similarly the jumps of the automaton correspond to the transitions of the HIOA code as illustrated in Figure 4.4.

4.2.4 HIOA Specification of the *Controller*

The HIOA specification of the controller is given in Figure 4.5. The first line declares the name of the hybrid automaton along with its formal parameters and the remaining lines of

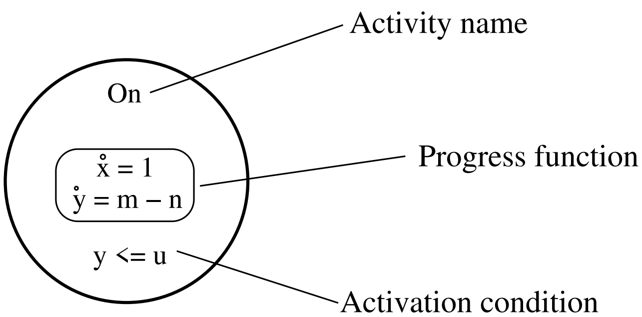

Figure 4.3: Equivalence of locations and activities

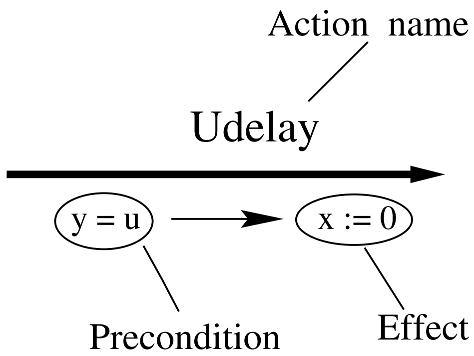

Figure 4.4: Equivalence of discrete jumps and actions

the specification define its components. The `CONTROLLER` automaton is parameterized by the upper(u) and the lower(l) sensor levels, the pump in-flow rate(m), the drain out-flow rate(n) and the lumped delays in the upper(ud) and the lower(ld) sensor circuits. All these parameters are typed `Real`, which is a built-in of HIOA.

The **signatures** section of the specification declares two internal actions *Udelay, Ldelay* and two output actions *TurnOn* and *TurnOff*. We declare *TurnOn* and *TurnOff* as output actions because these actions might be required to be visible externally. For instance, these actions could be used to send the `On/Off` signals to a pump automaton. On the other hand, *Udelay* and *Ldelay* are declared as internal actions as their occurrence need not be visible outside the `CONTROLLER`.

The automaton `CONTROLLER` has three internal(state) variables. The variable *loc* is of a user-defined enumeration type `Locations` and it encodes the location or the mode of the automaton. The continuous variables x and y are defined to be *analog*, which means that they have continuous dtypes[1]

The variables *loc* and x(time) are initialized to *On* and *0.0* respectively. The level y is set by a non-deterministic *choice* to any value in the closed interval [*l*,*u*].

The **transitions** of automaton `CONTROLLER` are given in the precondition-effect style. For instance, the action *Udelay* occurs if $loc = On \wedge y = u$, and as a consequence of it, the *loc* becomes *Flood* and the stopwatch x is reset.

The `CONTROLLER` automaton has four locations, so according to the mapping shown in Figure 4.3 we may write four corresponding activities for describing the flow of the locally controlled analog variables x and y. However, as the variables behave identically in the locations *On* and *Flood*, so these two are merged into a single activity *rising*. Similarly, the locations *Off* and *Starve* are merged into the activity *falling*. Each activity is headed by an activation condition, which is a predicate on the variables of the automaton. The body of the activities consist of the progress functions governing the evolution of x and y with time. Here the progress functions are simple constant rate expressions like, $\dot{x} = 1; \dot{y} = m - n$. The `"'"` operator is used to denote the first derivative of an analog variable.

4.3 Structure of HIOA Specifications

In this section we discuss the features of the HIOA language in detail.

[1] dtype of a variable v or the dynamic type is the set of functions from left-closed intervals of T to *type*(v) that is closed under *time shift, subinterval* and *pasting*. See Section 2.2.1.

```
hybridautomaton CONTROLLER(u,l,m,n,ld,ud:Real )
  type Locations = enumeration of On, Flood, Off, Starve
  signatures
    internal Udelay, Ldelay
    output TurnOn, TurnOff
  variables
    internal loc : Locations := On,
    internal analog x : Real := 0.0,
    internal analog y : Real := choose [l,u]
  transitions
    TurnOn
      pre loc = Starve ∧ x = ld
      eff loc := On
    Udelay
      pre loc = On ∧ y = u
      eff loc := Flood; x := 0.0
    TurnOff
      pre loc = Flood ∧ x = ud
      eff loc := Off;
    Ldelay
      pre loc = Off ∧ y = l
      eff loc := Starve; x := 0.0
  trajectories
    activity rising
      precon loc = On ∨ loc = Flood
      evolve x‘ := 1.0;
        y‘ := m−n
    activity falling
      precon loc = Off ∨ loc = Starve
      evolve x‘ := 1.0;
        y‘ := −n
```

Figure 4.5: HIOA code for *Controller*

4.3.1 Lexical Notes

Names

HIOA is case sensitive. In HIOA an identifier is a sequence of alphanumeric characters or underscores starting with alphabets, except those sequences which are keywords of the language[2]. Identifiers are used to *name* automata, types, variables, actions, and activities.

Real Literals

Real literal constants are customarily represented as two sequence of numerals separated by a `"."`; e.g., *9.81, 0.741*.

Comments

Any text beyond the `"%"` symbol in a line of HIOA code is treated as comment.

4.3.2 Data Types

HIOA has a set of built-in data types which can be used without explicit declarations, these are : `Bool, Int, Nat, Real, Char and String`. Complex data types like arrays, sequences and maps can be constructed out of these basic data types.

Users can either define new data types as enumerations/tuples of primitive types or they can refer to auxiliary specifications written in the *Larch Shared Language (LSL)* giving the syntax and the semantics of the new data types.

Example: **type** `opMode` = **enumeration of** *stop, accel, cruise, retard*

Example: **type** `state` = **tuple of** *mode* : `opMode`, *dist* : `Real`, *t* : `Time`

4.4 Primitive Automaton

The definition of a primitive hybrid automaton starts with a declaration and is followed by four sections namely signatures, variables, transitions, and trajectories. The declaration

[2] The list of HIOA keywords is given in the Appendix A.

starts with the keyword **hybridautomaton** and it consists of a unique name for the automaton followed by an optional list of formal parameters. Each formal parameter in the list is an identifier with an associated type. When composing several automata, the primitive definitions are instantiated by providing an actual parameter for each formal parameter.

Example: **hybridautomaton** `A`

Example: **hybridautomaton** `B(`x`:Int,`y`:Real,`b`:Bool)`

4.4.1 Signature

The actions of an automaton are declared using the **signatures** keyword followed by a list of its input, output and internal actions. Each item in the list consists of a name followed by an optional list of formal parameters or expressions following the **const** keyword. Each entry in the list corresponds to a set of actions, one for each assignment of values to its non-**const** parameters.

The range of values assumed by the parameters for an entry in the signature, can be restricted by **where** clauses. A **where** clause is a boolean-valued expression with the formal variables.

Example: **signatures**

internal *update*
input *read*(i, j :Int)
output *write*(k :Int) **where** k $\leq$ 20

4.4.2 Variables

The variables of the automaton are declared using the **variables** keyword followed by a list of input, output or internal variables and their types separated by commas. In HIOA, discrete and continuous variables are distinguished by the keyword **analog** in the variable declaration.

Locally controlled variables (internal and output) can be initialized during declaration using the assignment operator `:=` followed by an expression or by a *nondeterministic choice*. Input variables cannot be initialized.

Example : **input** x : `Int` ;
internal y : `Bool` := false;
output analog z : `Real` := **choose** [*0.0, 5.8*)

4.4.3 Choice

HIOA allows variables to be assigned nondeterministically with the **choose** clause. A nondeterministic choice is indicated by the keyword **choose** following the assignment operator. The choice is expressed by a sequence of open or closed intervals separated by semicolons.

Example : x := **choose** [*4.0, 10.0*) ; (*10.5, 16.0*]
Example : y := **choose** (*a,b*)
Example : z := **choose** [*9.81*, `\infty`)

Such assignments indicate that the variable on the left hand side of the assignment assumes any value contained in any of the intervals on the right hand side. Special symbol `"\infty"` is used to denote infinity.

4.4.4 Transitions

This section of the automaton definition begins with the **transitions** keyword and it consists of the transitions for the actions in the automaton's signature. A transition definition consists of an action name with parameters matching those of the action definition and an optional **where** clause. More than one definition can be given for a particular entry in the automaton's signature.

```
Example : hybridautomaton TOGGLE
            signatures
              output Set,
              input Reset
            variables
              input b : Bool := true,
              output a : Int ,
              output c : Real := 0.0
            transitions
              Set
                pre b = true
                eff a := 5 ;
                    c := choose (−\infty, -5.0 )
              Reset
                eff a := 0 ;
                    c := 5.0
            trajectories
              ...
```

Figure 4.6: Typical transitions

Preconditions

A precondition for a locally controlled action's transition is defined with the **pre** keyword followed by a predicate. An action is said to be *enabled* at a particular valuation of the variables if the precondition predicate for one of its transitions is true for that valuation. In the absence of a precondition it is tacitly assumed that the transition is always enabled. Since the occurrence of the input actions cannot be controlled by an automaton(see Section 2.3.4), so they are always enabled and do not have preconditions.

Effects

The effect of a transition, if any, is defined following the keyword **eff** . The effect is defined in terms of a *hybrid program*, which is a collection of statements assigning(possibly nondeterministically) values to the variables of the automaton. Whenever an action occurs, the corresponding hybrid program executes and the new values resulting from the execution of are assigned to the variables of the automaton instantaneously. Although in general hybrid programs assign values to variables as well as their first derivatives(for continuous variables), the *effect* of a transition can only assign values to variables and not their derivatives. Figure 4.6 shows the transitions of a `TOGGLE` automaton.

4.4.5 Hybrid Programs

The actions and the activities of a hybrid automaton describe discrete and continuous change of its variables respectively. Discrete changes are simple algebraic assignments to the variables which are effected when a particular action occurs. The continuous changes, are also described (see section 3.3.1) by first order ordinary differential equations(ODE) which are written as assignments to the first derivatives or *rates* of the variables.

These assignments are written in the HIOA specification as hybrid programs. A hybrid program is a sequence of statements separated by semicolons and they collectively specify the valuation of the variables when an action occurs or an activity operates. Statements are either direct or conditional assignments to variables or their first derivatives. An assignment consists of a locally controlled variable followed by the assignment operator `":="` and either an expression or a nondeterministic choice. The expression or the choice must have the same type as the variable being assigned. Input variables cannot be assigned. In the `TOGGLE` automaton of Figure 4.6 the output action *Set* occurs when the input variable b is `true` and it sets the variable a to *5* and variable c is nondeterministically assigned a value less than *-5.0*. The occurrence of the input action *Reset* assigns *0* and *5.0* to variables a and c respectively.

Conditional statements allow assignment of values to the variables depending on the a predicate. A conditional statement has the common **if** – **then** – **else** structure with any number of **elseif** branches in between as shown in Figure 4.7.

Syntactically the hybrid programs for transitions and trajectories are identical and there are no side-effects for the assignments. Semantically however, for transitions the

Example:
transitions
starting
pre $mode = accel$
eff
if $carType = power$ **then** a :=[*30.0*,*33.5*]
elseif $carType = low$ **then** a :=[*10.0*,*15.6*]
else a :=*20.0* **fi**
cruising
pre $mode = cruise$
eff $a := 0.0$
if $carType = low$ **then** v :=*5.8*
else v :=*9.65* **fi**

Figure 4.7: **if then else** in hybrid programs

assignments are made sequentially while for trajectories the assignments are made simultaneously(in parallel), so in the latter case the order of appearance of the statements do not matter.

4.4.6 Trajectories

The trajectories section is present in HIOA specifications with locally controlled analog variables. This section defines the continuous behaviour of the system. It begins with the keyword **trajectories** and is followed by a list of activities. An activity consists of a name, an optional activation condition, and a progress function.

Activation Conditions

The activation condition, if present, is a predicate over the variables, preceded by the keyword **precon** or **postcon** and the predicate defines the set of valuations of the variables(region) over which the particular activity is operating. Absence of an activation condition implies that the activity is always operating.

```
Example : trajectories
            flow1
              precon b := true
              evolve x' := 9.81;
                     s := a
            flow2
              postcon b := true
              evolve x' := choose [6.0,10.0 ];
                     s := a
```

Figure 4.8: Typical trajectory

A predicate following **precon** implies that the activity concerned is operating as long as the predicate is true. During the operating period of the activity, the values of the variables are determined according to the progress function of the activity. A predicate following **postcon** has a slightly different semantic interpretation, it means that the activity is operating until the predicate *becomes* true. So the activity *flow2* is operating as long as b is false, and in the final state of the execution under the activity *flow2*, $b =$ true; and at that point *flow1* becomes operating.

Progress Function

In every activity, the evolution of each locally controlled continuous variable is described by algebraic definitions or a set of first order ODEs constituting the progress function of the activity. The progress function is stated by the keyword **evolve** followed by a hybrid program consisting of assignment and conditional statements. Locally controlled variables of the automaton evolve according to the algebraic or differential equations specified in the hybrid program. Differential equations are written as assignments to the derivative of variables using the "‘" operator. Input variables cannot be assigned in any activity. Locally controlled variables which are not assigned are assumed to remain constant.

4.5 Composed Automaton

The composition operation on hybrid automata allows the user to define complex systems as an interaction of several simpler components. A composite automaton is constructed from primitive automata or other composite automata. The utility of composition and its semantics are dealt with in Section 5.4. In this section, we discuss the syntax of a composite automaton.

Component : **hybridautomaton** `SERVER(`i`:Int , `j`:Int )`
  signatures
    input *read*,
    output *write*
  variables
    input *someOutput* : `Real` ,
    internal k : `Int`
  transitions
    . . . ;
  trajectories
    . . . ;

Component : **hybridautomaton** `CLIENT(`a`:Int )`
  signatures
    output *read*,
    input *write*
  variables
    output *someOutput* : `Real` ,
    internal l : `Int`
  transitions
    . . . ;
  trajectories
    . . . ;

Figure 4.9: Component automata `SERVER` and `CLIENT`

The segments in Figure 4.9 define two primitive automata, apparently a `SERVER` and a `CLIENT`. It is to be noted that these two automata are compatible, according to the definition of compatibility given in Section 2.3.5. The executions of these two automata are discretely synchronized by the external actions *read* and *write* and synchronization over continuous intervals is achieved by the external variable *someOutput*.

In Figure 4.10, the composite `SYSTEM` automaton is constructed by instantiating and composing the primitive `CLIENT` and `SERVER` automata. For instantiation, every formal parameter in the automaton definition is replaced by a suitable actual parameter. The *handles* *s*, *c1* and *c2*, etc are used to refer to the state variables of the components.

Example : **hybridautomaton** `SYSTEM`
compose
s : `SERVER(`*550,13*`)` ;
c1 : `CLIENT(`*1*`)` ;
c2 : `CLIENT(`*2*`)` ;
...

Figure 4.10: Composed automaton

4.6 Assertions

HIOA can be used to state the properties of a system as automata invariants. An invariant is a predicate which is satisfied by every valuation of the variables of the automaton that is ever reached in any executions. An invariant assretion begins with the keywords **invariant of** followed by the name of the automaton and the predicate. For example, the following invariant asserts a certain upper and lower bound on the variable y for all possible instances of the `CONTROLLER` automaton (Figure 4.5).

Example : **invariant of** `CONTROLLER`

$$\forall u, l, m, n, ld, ud : \texttt{Real}$$

$$(\texttt{CONTROLLER}(u, l, m, n, ld, ud).y \leq u + (m - n) * ud$$

$$\wedge\ \texttt{CONTROLLER}(u, l, m, n, ld, ud).y \geq l + (-n) * ld)$$

4.7 Summary

In this chapter we have presented the structure of the HIOA language for specification of primitive and composed hybrid I/O automata. In Chapter 8 we evaluate HIOA by using it to specify hybrid systems given as functional blocks and as MMT-specifications and our experiences indicate that writing specifications in HIOA is simple and intuitive. In the next chapter we describe the front end tools supporting HIOA.

Chapter 5

HIOA Front End Tools

5.1 Introduction

The HIOA specification of the system written by the user has to be checked for syntactic and semantic correctness before it can be used for analysis. The HIOA front end performs this task and also transforms the HIOA specification into a simplified intermediate form which is used as input to the other tools in the workbench. In this chapter we give an overview of the implementation of the HIOA front end, we describe the intermediate representation language (IL) and the composer for hybrid automata.

5.2 Overview of HIOA Front End Implementation

The HIOA front end consists of a parser, syntactic and semantic checker and printer. The HIOA parser parses, reports syntactic errors and constructs an internal representation of the input specification.

The lexical analyzer and the tokenizer of HIOA are built by modifying the lexer of the IOA Toolkit [25]. In the first of the two passes, the parser internally constructs an Abstract Syntax Tree (AST) for the input specification and performs the semantic checks on the AST. In the second pass, the AST is traversed to generate the code of the hybrid automaton in the Intermediate Language(IL). The simplified representation of the automaton in IL serves as an input to the other tools as shown in Figure 1.2.

Standard compiler techniques (as in [27, 28]) have been employed for implementing the parser. We have used PolyJ [29], an extension of Java which allows parameterized polymorphism and Java CUP [30], an LALR parser generator for Java, for writing the parser. The implementation has been done in a pentium based system running Red Hat Linux 6.2 using JDK 1.1.8. The grammars of both HIOA and IL are given in Appendix A and Appendix B respectively.

5.3 Intermediate Language

The specification of a system in the intermediate language(IL) is the input to the back end tools of the hybrid workbench. The IL is a simplified version of HIOA conserving all the information given in the HIOA specification. The BNF grammar of the intermediate language is given in appendix B. The intermediate representation consists of a list of declarations followed by the primitive automaton in the intermediate form. There are no composed automata in the intermediate form, all automata described as compositions are converted to their primitive form by the *Composer*. The transformation of composed automata to their equivalent primitive form has been discussed in Section 5.4.

The declarations include types and operators which are used by all automaton definitions, for example the builtin types and the predefined operators. Primitive automata in their intermediate forms are represented by lists of all their subsections with resolved references to the variables, types and operators.

In the list of declarations, each variable, type and operator is assigned a unique *internal name* along with the *external name*, which is used in the original HIOA specification. Only the internal names are used in the body of the automaton specification. The IL representation of the `CONTROLLER`(Figure 4.5) automaton is given in Figure 5.1. The list of type declarations begins with the keyword **sorts** followed by the entries in the

```
(trait "DecimalLiterals" (formals s2) (ops (op13 (id "succ") (s2) s2)))
(hioa
 ((sorts
   (s0 Bool ())
   (s1 "type" ())
   (s3 "Int" () lit)
   (s5 "Real" () lit)
    ...
  (ops
   (op1 (infix "~=") ((s0 s0) s0))
   (op4 (id "false") (() s0))
   (op9 (id "true") (() s0))
   (op14 (infix "=") ((s0 s0) s0))
    ...
   (op295 (select "x") ((s21) s5))
   (op296 (select "y") ((s21) s5)))
  (vars
   (v0 "u" s5)
   (v1 "l" s5)
   (v2 "m" s5)
   (v3 "n" s5)
    ...
 (hybridautomaton "CONTROLLER" (formals v0 v1 v2 v3 v4 v5)
  ((actions (a1 internal "Udelay") (a2 internal "Ldelay")
    (a3 output "TurnOn") (a4 output "TurnOff"))
   (variables (internal DISCRETE v6) (internal ANALOG v7 (lit s5 0.0))
    (internal ANALOG v8 (choose (( [ v1 , v0 ])))))
   (transitions
    (a3 (pre (apply op3 (apply op282 v6 op289) (apply op65 v7 v4)))
     (eff ((assign (v6) op286))))
    (a1 (pre (apply op3 (apply op282 v6 op286) (apply op65 v8 v0)))
     (eff ((assign (v6) op287) (assign (v7) (lit s5 0.0)))))
    (a4 (pre (apply op3 (apply op282 v6 op287) (apply op65 v7 v5)))
     (eff ((assign (v6) op288))))
    (a2 (pre (apply op3 (apply op282 v6 op288) (apply op65 v8 v1)))
     (eff ((assign (v6) op289) (assign (v7) (lit s5 0.0))))))
   (trajectories
    (f1 "rising"
     (preCon (apply op8 (apply op282 v6 op286) (apply op282 v6 op287)))
     (evolve
      ((assign (d( v7 )) (lit s5 1.0))
       (assign (d( v8 )) (apply op72 v2 v3)))))
    (f2 "falling"
     (preCon (apply op8 (apply op282 v6 op288) (apply op282 v6 op289)))
     (evolve
      ((assign (d( v7 )) (lit s5 1.0))
       (assign (d( v8 )) (apply op69 v3))))))))))
```

Figure 5.1: Intermediate representation of level controller

list, each consisting of an internal name beginning with the letter `"s"` and an external name. Similarly the list of variable declarations begins with **vars** and the entries consist of internal names beginning with `"v"`, external names and the type of the variable. The keyword **lit** in a type declaration or an assignment indicates that the member is of literal type (decimal literals are constructors of the member). Operator declarations begin with the keyword **ops** and each new operator is represented by an internal name starting with `"o"`, external name and its signature. The special operator **d()** is used to denote the first derivative of variables.

The intermediate representation of an automaton is a list of its actions, variables, transitions and activities ordered according to their appearance in the HIOA code. The intermediate code uses a subset of the HIOA keywords and the internal names for the variables, types and operators. Assignments are represented by the keyword **assign** followed by the left and the right sides of the assignment. An operator declared in the list of the automaton is used with the keyword **apply** followed by the internal name of the operator and the parameters.

5.4 Composer

A complex hybrid system is described as an interaction of several component hybrid I/O automata using the **compose** keyword as it has been shown in Section 4.5. However most analytical methods deal with primitive automata, hence a *composing* tool is necessary for transforming the composite description of the system to an equivalent primitive automaton by eliminating the **compose** operator. The semantics of this composition transformation conforms with the composition operation described in Section 2.3.6.

We describe the composition operation for a generalized composite hybrid I/O automaton $\mathcal{A}$ (Figure 5.2) with n primitive components $A_1 \ldots A_n$ as given in Figure 5.3. If the components are not primitive automata, then the composition transformation is applied on them recursively. If the definition of a component has formal parameters, then they are replaced by the actual parameters provided in the instantiation of that particular automaton in the composite.

hybridautomaton $\mathcal{A}$
compose
$A_1(p_{11}, \ldots, p_{1a_1})$
$A_2(p_{21}, \ldots, p_{2a_2})$
$\ldots$
$A_n(p_{n1}, \ldots, p_{na_n})$

Figure 5.2: Generalized composed HIOA

The generalized component A_i is a primitive automaton parameterized by $p_1, \ldots, p_{a_i}$ parameters. Without any loss of generality, all the actions are assumed to have k parameters the values of which are constrained by the corresponding **where** predicates. Each action signature may have several corresponding transitions; for example, the signature $I_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $IPred_i(m_1, \ldots, m_k)$ defines input actions for those values of $m_1, \ldots, m_k$ which satisfy the predicate $IPred_i$ and the transitions for these actions are given by e_i number of $I_i(m_1, \ldots, m_k)$ entries in the **transitions** section. Each I_i entry defines the transitions for a set of actions determined by its $IPred_{ik}$ predicate, where $1 \leq k \leq e_i$. The component A_i has b_i, c_i and d_i numbers of input, output and internal variables respectively. A variable is either continuous (**analog**) or it is discrete. The progress functions of the activities $F_{i1} \ldots F_{ih_i}$ assign values to the locally controlled continuous variables or their first derivatives. The activation conditions $P_{i1} \ldots P_{ihIi}$ are predicates on the variables of the automaton.

5.5 Compatibility Conditions

The conditions for compatibility of two automata have been stated in Section 2.3.5. Now we restate these conditions in terms of the generalized component automaton A_i. The first four conditions correspond to those in Section 2.3.5 in addition to which two other conditions are required to prevent duplicate declaration of actions and undeclared actions.

hybridautomaton $A_i(p_1 : P_1, \ldots, p_{a_i} : P_{a_i})$
  signatures
    input $I_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $IPred_i(m_1, \ldots, m_k), \ldots$
    output $O_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $OPred_i(m_1, \ldots, m_k), \ldots$
    internal $H_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $HPred_i(m_1, \ldots, m_k), \ldots$
  variables
    input (**analog**) $U_{i1} : S_{11}, \ldots,$ **input** $U_{ib_i} : S_{1b_i},$
    output (**analog**) $Y_{i1} : S_{21}, \ldots,$ **output** $Y_{ic_i} : S_{2c_i},$
    internal (**analog**) $X_{i1} : S_{31}, \ldots,$ **internal** $X_{id_i} : S_{3d_i}$
  transitions
    $I_i(m_1, \ldots, m_k)$ **where** $IPred_{i1}(m_1, \ldots, m_k)$ **eff** $IProg_{i1}$
    $\ldots$
    $I_i(m_1, \ldots, m_k)$ **where** $IPred_{ie_i}(m_1, \ldots, m_k)$ **eff** $IProg_{ie_i}$
    $O_i(m_1, \ldots, m_k)$ **where** $OPred_{i1}(m_1, \ldots, m_k)$ **pre** $OGuard_{i1}$ **eff** $OProg_{i1}$
    $\ldots$
    $O_i(m_1, \ldots, m_k)$ **where** $OPred_{if_i}(m_1, \ldots, m_k)$ **pre** $OGuard_{if_i}$ **eff** $OProg_{if_i}$
    $H_i(m_1, \ldots, m_k)$ **where** $HPred_{i1}(m_1, \ldots, mk_)$ **pre** $HGuard_{i1}$ **eff** $HProg_{i1}$
    $\ldots$
    $H_i(m_1, \ldots, m_k)$ **where** $HPred_{ig_i}(m_1, \ldots, m_k)$ **pre** $HGuard_{ig_i}$ **eff** $HProg_{ig_i}$
  trajectories
    activity F_{i1} **precon** P_{i1} **evolve** $Y_{i1}` := \ldots; \ldots; X_{i1} := \ldots; \ldots$
    $\ldots$
    activity F_{ih_i} **precon** P_{ih_i} **evolve** $Y_{il}` := \ldots; \ldots; X_{ik} := \ldots; \ldots$

Figure 5.3: Generalized HIOA component

The compatibility conditions on the component A_is are as follows:

1. Set of internal variables is disjoint with variables of all other components:

$$\forall\, i, j \in \{1, \ldots, n\} \wedge i \neq j, \quad \{X_{im} | 1 \leq m \leq d_i\} \bigcap \begin{pmatrix} \{U_{jm} | 1 \leq m \leq b_j\} \\ \{Y_{jm} | 1 \leq m \leq c_j\} \\ \{X_{jm} | 1 \leq m \leq d_j\} \end{pmatrix} = \phi$$

2. Set of output variables is disjoint with output variables of all other components:

$$\forall\, i,j \in \{1,\ldots,n\} \wedge i \neq j, \quad \{Y_{im} | 1 \leq m \leq c_i\} \bigcap \{Y_{jm} | 1 \leq m \leq c_j\} = \phi$$

3. Set of output actions is disjoint with output actions of all other components:
   $\forall\, i,j \in \{1,\ldots,n\} \wedge i \neq j, \;\; OPred_i(m_1,\ldots,m_k) \cap OPred_j(m_1,\ldots,m_k) = \phi$

4. Set of internal actions is disjoint with actions of all other components:

$$\forall\, i,j \in \{1,\ldots,n\} \wedge i \neq j, \;\; HPred_i(m_1,\ldots,m_k) \bigcap \begin{pmatrix} IPred_j(m_1,\ldots,m_k) \\ OPred_j(m_1,\ldots,m_k) \\ HPred_j(m_1,\ldots,m_k) \end{pmatrix} = \phi$$

5. There exists at most one transition for every action:

$$\begin{aligned} &\forall\, i \in \{1,\ldots,n\} \\ &\quad \forall\, i_1,i_2 \in \{1,\ldots,e_i\}, \quad IPred_{ii_1}(m_1,\ldots,m_k) \wedge IPred_{ii_2}(m_1,\ldots,m_k) \Rightarrow i_1 = i_2 \text{ and} \\ &\quad \forall\, i_1,i_2 \in \{1,\ldots,f_i\}, \quad OPred_{ii_1}(m_1,\ldots,m_k) \wedge OPred_{ii_2}(m_1,\ldots,m_k) \Rightarrow i_1 = i_2 \text{ and} \\ &\quad \forall\, i_1,i_2 \in \{1,\ldots,g_i\}, \quad HPred_{ii_1}(m_1,\ldots,m_k) \wedge HPred_{ii_2}(m_1,\ldots,m_k) \Rightarrow i_1 = i_2 \end{aligned}$$

6. Every action should have a corresponding transition:

$$\begin{aligned} &\forall i \in \{1,\ldots,n\}, \\ &\quad \forall m_1,\ldots,m_k, \\ &\qquad IPred\_i(m_1,\ldots,m_k) \Rightarrow \exists\, j \in \{1,\ldots,e_i\} \quad IPred_{ij}(m_1,\ldots,m_k) \text{ and} \\ &\qquad OPred_i(m_1,\ldots,m_k) \Rightarrow \exists\, j \in \{1,\ldots,f_i\} \quad OPred_{ij}(m_1,\ldots,m_k) \text{ and} \\ &\qquad HPred_i(m_1,\ldots,m_k) \Rightarrow \exists\, j \in \{1,\ldots,g_i\} \quad HPred_{ij}(m_1,\ldots,m_k) \end{aligned}$$

In the following sections we develop the equivalent primitive automaton $\mathcal{B}$ for the composed automaton $\mathcal{A}$. The formal definition of composition has been given in Section 2.3.6. In most parts we extend the composing rules of IOA [31] to develop the composition rules for hybrid I/O automaton .

5.6 Signature of $\mathcal{B}$

The signature of the transformed automaton is obtained by replacing the formal parameters in the definition of the components by the actual parameters provided in the **compose** clause. For example, the automaton $\mathcal{B}$ in Figure 5.2, instantiates the component $A_1(p_1 : P_1, \ldots, p_{a_1} : P_{a_1})$ by the actuals $p_{11}, \ldots, p_{1a_1}$, so in the signature of $\mathcal{B}$, all occurrences of p_1 in A_1's signature are replaced by p_{11}.

5.6.1 Internal Actions and Output Actions of $\mathcal{B}$

The set of output(internal) actions for $\mathcal{B}$ is the union of the set of output(internal) actions of all the component A_i's. This means that a particular output action header
$O_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $OPred_i(m_1, \ldots, m_k)$ is included in the signature of $\mathcal{B}$ iff

- The pattern $O_i(m_1, \ldots, m_k)$ appears in the output action list of one of the A_i's, and
- $OPred_i = OPred_1 \vee OPred_2 \vee \ldots$ where $OPred_1, OPred_2, \ldots$ are the **where** clauses of the actions with the pattern $O_i(m_1, \ldots, m_k)$.

The same rule applies for the inclusion of an internal action
$H_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $HPred_i(m_1, \ldots, m_k)$ in the signature of $\mathcal{B}$.

5.6.2 Input actions of $\mathcal{B}$

An input action of a component which is *not* an output action of any other component is included in the set of input actions of $\mathcal{B}$. In other words, the action header
$I_i(m_1 : M_1, \ldots, m_k : M_k)$ **where** $IPred_i(m_1, \ldots, m_k)$ is included in the signature of $\mathcal{B}$ iff

- The pattern $I_i(m_1, \ldots, m_k)$ appears in the input action list of one of the A_i's.
- $IPred_i = IPred_1 \vee IPred_2 \vee \ldots$ where $IPred_1, IPred_2, \ldots$ are the **where** clauses of the actions with the pattern $I_i(m_1, \ldots, m_k)$.

- $OPred_1 \vee OPred_2 \vee \ldots = \mathsf{false}$ where $OPred_1, OPred_2, \ldots$ are the **where** clauses of the output actions(if any) with the pattern $I_i(m_1, \ldots, m_k)$.

5.7 Variables of $\mathcal{B}$

The set of variables of $\mathcal{B}$ is the union of all the variables of the components(A_is), with their references modified. The variables references are modified by prepending the original variable names with a list of defining component names and actual parameters instantiating the component. For example, the output variable U_{12} of component A_1 instantiated with parameters $p_{11}, \ldots, p_{1a_1}$ is referred to as $A_1(p_{11}, \ldots, p_{1a_1}).U_{12}$.

The set of output(internal) variables of $\mathcal{B}$ is the union of all output(internal) variables of its components. An output(internal) variables of $\mathcal{B}$ is initialized if the corresponding variable in the component is initialized. The compatibility assumptions 1 and 2 ensure that there are no conflicting initializations. The set of input variables of $\mathcal{B}$ consists of those input variables of its components which are not output variables for some other component. Input variables are not initialized.

5.8 Transitions of $\mathcal{B}$

For each input or output action entry in the signature of the automaton, a single transition is defined. For input actions, the resulting transition is a chain of **if** statements. The transition for the action $I_i(m_1, \ldots, m_k)$ is formed by grouping together all transitions for $I_i(m_1, \ldots, m_k)$ of the component automata. Each branch of the chain corresponds to a transition of one of the components for this particular action. The condition following **if** is the **where** predicate and the action following **then** is the corresponding effect of the component's transition. In Figure 5.4 the I_1 transition for the composed automaton has been shown[1].

[1] We have removed the parameters $m_1, \ldots, m_k$ of the **where** predicates for clarity

$$
\begin{array}{lll}
\multicolumn{3}{l}{I_1(m_1,\ldots,m_k)} \\
& \textbf{eff} & \textbf{if } IPred_{11} \ \wedge \ IPred_1 \textbf{ then } IProg_{11} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } IPred_{1e_1} \ \wedge \ IPred_1 \textbf{ then } IProg_{1e_1} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } IPred_{n1} \ \wedge \ IPred_n \textbf{ then } IProg_{n1} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } IPred_{ne_n} \ \wedge \ IPred_n \textbf{ then } IProg_{ne_n} \textbf{ fi } ;
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{O_1(m_1,\ldots,m_k)} \\
& \textbf{pre} & \textbf{if } OPred_{11} \ \wedge \ OPred_1 \textbf{ then } OGuard_{11} \\
& & \ldots \\
& & \textbf{elseif } OPred_{1f_1} \ \wedge \ OPred_1 \textbf{ then } OGuard_{1f_1} \\
& & \ldots \\
& & \textbf{elseif } OPred_{n1} \ \wedge \ OPred_n \textbf{ then } OGuard_{n1} \\
& & \ldots \\
& & \textbf{elseif } OPred_{nf_1} \ \wedge \ OPred_n \textbf{ then } OGuard_{nf_1} \textbf{ fi} \\
& \textbf{eff} & \textbf{if } OPred_{11} \ \wedge \ OPred_1 \textbf{ then } OProg_{11} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } OPred_{1e_1} \ \wedge \ OPred_1 \textbf{ then } OProg_{1e_1} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } OPred_{n1} \ \wedge \ OPred_n \textbf{ then } OProg_{n1} \textbf{ fi } ; \\
& & \ldots \\
& & \textbf{if } OPred_{ne_n} \ \wedge \ OPred_n \textbf{ then } OProg_{ne_n} \textbf{ fi } ;
\end{array}
$$

Figure 5.4: Input action I_1 and output action O_1 of $\mathcal{B}$

A similar procedure is followed to construct the output actions for $\mathcal{B}$. The preconditions of the output actions of the components are combined in an **if** -**elseif** statement (Figure 5.4). The order of the conditional statements in the transitions of $\mathcal{B}$ is not important, because the components are compatible and each component defines only one transition for every action. Further for the output actions, according to the compatibility assumptions, at most one **if** condition is true at a particular valuation, and therefore at most one of the effect programs is executed.

5.9 Trajectories of $\mathcal{B}$

The activities of $\mathcal{B}$ are the combination of the activities of the components. We assume that all the component automata have simple activities[2], i.e.,
$\forall\, i \in \{1, \ldots, n\}, \forall\, j, k \in \{1, \ldots, h_i\}, j \neq k \Longrightarrow P_{ij} \cap P_{ik} = \phi$, where h_i is the number of activities of the i^{th} component and P_{ij} is the activation condition of the j^{th} activity.

hybridautomaton A_1
  signatures
    ...
  variables
    input analog u_0, u_1 : `Real` ,
    output analog $y_0, y1$: `Real` ,
    internal analog x_{11}, x_{12} : `Real` ,
  transitions
    ...
  trajectories
    activity F_1 **precon** $\mathcal{P}_{F_1}$
      evolve y_0‘ $:= E_{F_1 y_0}$; y_1‘ $:= E_{F_1 y_1}$; x_{11}‘ $:= E_{F_1 x_{11}}$; x_{12}‘ $:= E_{F_1 x_{12}}$;
    activity F_2 **precon** $\mathcal{P}_{F_2}$
      evolve y_0‘ $:= E_{F_2 y_0}$; y_1‘ $:= E_{F_2 y_1}$; x_{11}‘ $:= E_{F_2 x_{11}}$; x_{12}‘ $:= E_{F_2 x_{12}}$;

hybridautomaton A_2
  signatures
    ...
  variables
    input analog y_0, u_2 : `Real` ,
    output analog u_0, y_2 : `Real` ,
    internal analog x_{21}, x_{22} : `Real` ,
  transitions
    ...
  trajectories
    activity G_1 **precon** $\mathcal{P}_{G_1}$
      evolve u_0‘ $:= E_{G_1 u_0}$; y_2‘ $:= E_{G_1 y_2}$; x_{21}‘ $:= E_{G_1 x_{21}}$; x_{22}‘ $:= E_{G_1 x_{22}}$
    activity G_2 **precon** $\mathcal{P}_{G_2}$
      evolve u_0‘ $:= E_{G_2 u_0}$; y_2‘ $:= E_{G_2 y_2}$; x_{21}‘ $:= E_{G_2 x_{21}}$; x_{22}‘ $:= E_{G_2 x_{22}}$

Figure 5.5: Component automata A_1 and A_2

So the activation conditions of the i^{th} automaton partitions its variable space into h_i

[2]The procedure for converting a complex set of activities to a simple set has been discussed in 3.3.3

parts. Superposing the partitions created by all the components, at most $h_1 \times h_2 \times \ldots \times h_n$ parts of the total variable space are created. Each of these parts is represented as activities of $\mathcal{B}$, the whole partition constituting a set of simple activities.

hybridautomaton $\mathcal{B}$
  signatures
    ...
  variables
    input analog u_1, u_2 : `Real` ,
    output analog u_0, y_0, y_1, y_2 : `Real` ,
    internal analog $x_{11}, x_{12}, x_{21}, x_{22}$: `Real` ,
  transitions
    ...
  trajectories
    activity F_1G_1 **precon** $\mathcal{P}_{F_1} \wedge \mathcal{P}_{G_1}$
      evolve $y_0` := E_{F_1y_0}$; $y_1` := E_{F_1y_1}$; $x_{11}` := E_{F_1x_{11}}$; $x_{12}` := E_{F_1x_{12}}$;
        $u_0` := E_{G_1u_0}$; $x_{21}` := E_{G_1x_{21}}$; $x_{22}` := E_{G_1x_{22}}$
    activity F_1G_2 **precon** $\mathcal{P}_{F_1} \wedge \mathcal{P}_{G_2}$
      evolve $y_0` := E_{F_1y_0}$; $y_1` := E_{F_1y_1}$; $x_{11}` := E_{F_1x_{11}}$; $x_{12}` := E_{F_1x_{12}}$;
        $u_0` := E_{G_2u_0}$; $x_{21}` := E_{G_2x_{21}}$; $x_{22}` := E_{G_2x_{22}}$
    activity F_2G_1 **precon** $\mathcal{P}_{F_2} \wedge \mathcal{P}_{G_1}$
      evolve $y_0` := E_{F_2y_0}$; $y_1` := E_{F_2y_1}$; $x_{11}` := E_{F_2x_{11}}$; $x_{12}` := E_{F_2x_{12}}$;
        $u_0` := E_{G_1u_0}$; $x_{21}` := E_{G_1x_{21}}$; $x_{22}` := E_{G_1x_{22}}$
    activity F_2G_2 **precon** $\mathcal{P}_{F_2} \wedge \mathcal{P}_{G_2}$
      evolve $y_0` := E_{F_2y_0}$; $y_1` := E_{F_2y_1}$; $x_{11}` := E_{F_2x_{11}}$; $x_{12}` := E_{F_2x_{12}}$;
        $u_0` := E_{G_2u_0}$; $x_{21}` := E_{G_2x_{21}}$; $x_{22}` := E_{G_2x_{22}}$

Figure 5.6: Composed automaton $\mathcal{B}$

We illustrate the composition of activities with an example. The continuous variables and the activities of two component automata A_1 and A_2 are shown in Figure 5.5. Each component has two disjoint activities F_1, F_2 and G_1, G_2 respectively, with corresponding progress functions assigning values to their continuous variables. The right side of the assignments (either terms or **choose** expressions) are represented by distinct symbols of the form E_{F_i*} and E_{G_i*}. The composed automaton $\mathcal{B}$, shown in Figure 5.6, has four activities which are the result of superposing the partitions created by the activation conditions of the component activities. The activation condition of F_iG_j is the conjunction of the activation conditions of F_i and G_j. The progress function of F_iG_j is obtained by combining the progress functions of F_i and G_j. Note that u_0 and y_0 are input variables and the disjointness of the locally controlled varibales ensures that the assignments to the variables (or their first derivatives) are not conflicting.

5.10 Summary

In this chapter we have presented the front end tools supporting the HIOA language. An overview of the implementation of the HIOA checker was given followed by a description of the intermediate language and the HIOA composer.

So far we have been concerned with the HIOA language, its design features and its front end; in the remaing part of this thesis we look into the usage of deduction tools with HIOA specifications and we propose a framework for translating HIOA specifications to a form suitable for theorem proving using the Prototype Verification System(PVS). To this end, in the next chapter we suspend the discussion on hybrid systems for a while and study the process of specification and verification in PVS and in Chapter 7 we develop the framework for translating HIOA specifications to PVS theories.

Chapter 6

Verifying Properties with PVS

6.1 Introduction

In this chapter and the next, we look at interfacing the HIOA language with deductive tools for verifying properties of the systems specified in HIOA. Powerful general purpose mechanical theorem provers like PVS [32], HOL [33] and Larch [34], have been developed over the years, which can be used for rigorously checking and organizing proofs and also for systematically proving new properties by administering predefined proof methods. PVS is a theorem prover for high order logic, which can be used to specify and verify hybrid systems. It provides a combination of a rich specification language and a powerful prover.

In this chapter we present the verification of a leader election algorithm in a synchronous ring as an introduction to the process of using PVS for specification and verification. Even though this system does not have continuous variables(activities), the techniques for proving invariants are similar to those used for hybrid systems. The LCR leader election algorithm, which we verify in this chapter is simple and the proofs involved in establishing its correctness are intuitive; hence we can get a clear view of the process of using PVS without getting mired into the details of the problem.

6.1.1 The Prototype Verification System(PVS)

The following description of PVS is taken from [35]:

> PVS [32] is an environment for specification and verification that has been developed at SRI International's Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [33] and the Boyer-Moore prover [36], the distinguishing character of PVS is that it supports both a highly expressive specification language and a very effective interactive theorem prover in which most of the low-level proof steps are automated. The system consists of a specification language, a parser, a type checker, and an interactive proof checker. The PVS specification language is based on high-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught by the type checker. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based boolean simplification.

6.2 The LCR Algorithm

The necessity of electing a leader arises under various circumstances in distributed systems, the most commonly cited scenario is in a ring network where a *token* circulates around the network and only the process possessing the *token* has the right to communicate with its neighbors. If for some reason the token is lost, then a new token has to be generated, which amounts to electing a leader amongst the processes. The uniqueness of the token, prevents the possibility of multiple processes trying to communicate simultaneously and causing interference.

We assume a synchronous system, where the processes are in a unidirectional ring consisting of n nodes, with indices numbered between 1 to n. Each process is associated with one of the n nodes and with a *unique identifier (UID).* The set of UIDs is *totally ordered.* It is also assumed that the UIDs of all the processes are distinct, but there are no constraints on which UIDs actually appear in the ring. A simple solution of the leader election problem in this system is given by the so called *LCR algorithm* named after Le Lann, Chang, and Roberts. The following is an informal description of the algorithm as given in [37].

> **Algorithm Description**
> Each process sends its UID around the ring. When a process receives an incoming UID, it compares that to its own. If the incoming UID is greater than its own, it keeps passing the UID; if it is less than its own, it discards the incoming UID; if it is equal to its own, the process declares itself the leader.

After the initiation of this algorithm, the process with the largest UID is eventually elected as the leader, and it is the only process to be elected. The following formal description of the algorithm is also taken from [37].

> **Algorithm Pseudo code**
> For each i, $states_i$ denotes the state of the i^{th} process, which has three components :
>
> u, a UID , initially $i's$ UID
> $send$, a UID or $null$, initially $i's$ UID
> $status \in \{unknown, leader\}$, initially $unknown$
>
> For each i, the message-generation function sends the current value of $send_i$ to the process $i + 1$. All additions in this definition are *modulus n* . For each i, the transition function is defined by the following pseudo code:
>
> *send* := *null*
> if the incoming UID is v, then
>   case
> $v > u$: $send := v$
> $v = u$: $status := leader$
> $v < u$: do nothing

6.2.1 Correctness of the algorithm

A *correct* leader election algorithm is one which when invoked causes *exactly one* of the processes in the system to perform a *leader* output eventually.

Let us denote the index of the process with maximum UID by i_{max}, and let u_{max} denote its UID. Then, to prove that a given algorithm is correct it is enough to show that (1) process i_{max} outputs *leader* at the end of round n, and (2) no other process ever gives such an output. These conditions are stated formally in terms of theorems later in Section 6.4 and are proved in section 6.5.

6.3 Specification in PVS

In the remaining part of this chapter we model the system, specify the algorithm and prove its correctness using PVS. We present the essential parts of this exercise; the complete PVS dump file can be found at:

`www2.csa.iisc.ernet.in/~mitras/main/research/ringleader.dmp`.

We model the whole system as a synchronous network where the above algorithm defines the state transition function. The specification consists of the main theory `RingLeader` and a theory called `RING` which defines a set of node indices forming a ring.

6.3.1 The Ring

The `RING[k:posnat]` is a parameterized theory describing the properties of a ring with k elements. First we define a non-empty type `INDEX`, for the set of node indices and the functions necessary for accessing the neighboring nodes in a ring. Note that the indices are numbered from $1...k$ and not from $0...(k-1)$ and the following functions simply provide a modified version of modulo k addition and subtraction.

```
INDEX:NONEMPTY_TYPE = {x:posnat | 1<= x AND x <=k}
CCW(i:INDEX):INDEX  =  (IF i = 1 THEN k ELSE i - 1 ENDIF)
CW(s:nat, i:INDEX):INDEX = (IF s + i <= k THEN s + i
     ELSE s + i - k  ENDIF)
```

The `CCW(i)` function is straightforward, it gives the index of the node immediately next to one with index i in the *counter-clockwise(CCW)* direction. The `CW(s, i)` function is more general, it gives the index of a node s places from i in the *clock-wise(CW)* direction. Throughout this chapter we shall use the abbreviations CW and CCW and also the functions `CW` and `CCW`, the context will clarify their meanings. Several useful lemmas regarding properties of such a ring are also stated as a part of this theory, we mention them as and when they are used in proofs.

6.3.2 The UIDs

The set of UIDs of the processes is defined as a non-empty type `UID` satisfying strict total order. The map `name(i:index)` gives the UID of the process residing at the node with index i. This mapping is unique, no two nodes are mapped to the same UID:

```
UNIQUE_NAMES: LEMMA FORALL(i,j): name(i) = name(j) IMPLIES i = j
```

A special `UID` called `null` is defined, which is same as the *null* value in the algorithm and is used as a placeholder indicating absence of any message. We use an index i to denote both the i^{th} node in the ring and also the process residing therein. As the UIDs are totally ordered, there must exist a maximum element. Let the maximum element be `nameMax`, and the node index at which this process resides be `max`.

```
nameMax : UID = name(max)
MAX_EXISTS: LEMMA
     FORALL(j:INDEX): j/=max IMPLIES name(j) < nameMax
```

6.3.3 Ring Segments: The `Away` Function

In proving the uniqueness of the elected leader, we will have to use subsets of the nodes in the ring starting from `max` upto a given node k. These subsets are segments of the ring with the common form [max,...,k].

Definition:$[max, k] \triangleq \{max, CW(1, max), \ldots, CW(x, max)\}$, with $CW(x, max) = k$.

As the node indices form an ordered ring structure, for specifying the above segments in PVS, we simply use the distance of the last element k from `max`. We define a recursive function `Away` which gives the *distance* of a node from `max` in terms of the number of node hops in the CW direction.

```
Away(k:INDEX):RECURSIVE upton = IF k = max THEN 0
     ELSE 1 + Away(CCW(k)) ENDIF
         MEASURE (LAMBDA(x:INDEX):x)
```

where `upton` is a sub-type of natural numbers restricted to $0 \ldots (n-1)$.

Several lemmas derived from the `Away` function are stated in the theory, of which the following establishes the relationship between `CW` and `Away` functions.

```
Away_prop5: LEMMA CW(Away(k),max) = k
```

`Away(k)` implicitly represents the set

$$\texttt{[max,k]} = \{\texttt{CW(p,max)} \mid p \geq 0 \text{ and } p \leq \texttt{Away(k)}\}.$$

From the above definition it is clear that if `Away(i) < Away(j)`, then `[max,i]` $\subset$ `[max,j]`. For all nodes except `max`, `Away(CCW(i))` $<$ `Away(i)`; hence the set [`max, CCW(i)`] is a subset of [`max, i`]. We write this in a convenient way:

```
Away_subset: LEMMA FORALL(i,j):
    Away(i) <= Away(CCW(j)) AND j /= max
        IMPLIES Away(i) <= Away(j)
```

6.3.4 The System

A process state is a 3-tuple, consisting of three state variables.

```
PROCESS: TYPE = [UID, UID, STATUS_TYPES ]
initProcess (i:INDEX) : PROCESS =  (name(i),name(i),UNKNOWN)
```

where `STATUS_TYPES` is {`UNKNOWN, LEADER`} corresponding to the state component $status_i$ of Section 6.2. The first and second components of the `PROCESS` tuple correspond to state variables u_i and $send_i$ respectively. The `initProcess` function gives the initial state of a process. According to the algorithm, each process is initialized by setting u and *send* to its own UID and *status* to *unknown*.

The system state is defined as an array of n process states with their indices forming a ring. `SYSTEM(i)` denotes the i^{th} process in the ring. A map `systemState` gives the state of the system after the t^{th} round of computation.

```
SYSTEM:TYPE =  ARRAY [i:INDEX -> PROCESS]
systemState(t:nat) : SYSTEM
```

6.3.5 Model of Computation

In the synchronous network model [37], an *execution* of the system is defined to be an infinite sequence $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$, where each C_r is a state assignment, i.e., assignment of state to each process in the system and each of M_r and N_r are message assignments, i.e., an assignment of messages to each channel. C_r is the system state after r rounds, while M_r and N_r represent the messages that are sent and received at the r^{th} round, respectively.

We assume an ideal and failure-free communication system, i.e, the channels are lossless and without any delay. This implies that no additional information is stored in the channels, the state of the system is entirely determined by the state of the processes and the system execution can be represented as a sequence $C_0, C_1, C_2, C_3...$, of state assignments alone.

The unidirectional ring architecture of the network makes the system model neat; as each process can send a message only to its immediate CW neighbor, the message sent is stored in the *send* variable of the process state. As we have assumed the channels to be without loss or delay, we need not store the message received by a process separately, it can be directly read from the *send* variable of its CCW neighbor.

6.3.6 Algorithm

The algorithm describes how the system state evolves in successive rounds of execution. As the message passing information is encapsulated in the process state itself, the `systemState(t)` i.e, the state after the t^{th} round is a function of `systemState(t-1)` alone. This transition function is not defined explicitly, but its nature is captured in the following axioms which describe the LCR algorithm. The initial condition is written as the system state assignment at $t = 0$. The allowed state transitions are stated as three axioms, corresponding to three cases in the algorithm.

```
INITIALLY: AXIOM FORALL(i) :
    systemState(0) = LAMBDA (i:INDEX): initProcess(i)

incoming = systemState(t)(CCW(i))‘2 ,
past = systemState(t)(i)
present = systemState(t+1)(i)

MORE: AXIOM : ( incoming /= null AND  past‘1 < incoming)
      IMPLIES present = (this‘1 , incoming ,  this‘3 )
```

```
EQUAL: AXIOM : (incoming /= null AND past`1 = incoming)
       IMPLIES present = (this`1 , null , LEADER )

LESS: AXIOM : (incoming = null or incoming < this`1 )
      IMPLIES present = (this`1 , null,  this`3 )
```

6.4 Executions and Properties

For modelling executions in PVS, we use sequences as defined by Devilliers and Griffioen[38]. This formalization is compact, intuitive and it does not require the user to distinguish between finite and infinite sequences. A sequence is defined as a mapping from downward-closed subset of natural numbers($\mathbf{N}$) to a data set, which is `systemState` in our case. A sequence is defined as a record with fields `dom` and `map`. We model executions of the system as sequences of system state assignments called `TRACE`.

```
seq: TYPE = [# dom:index_sets, map[(dom)->T] #]
```

where, the range type of `map` is a parameter of the theory, and `index_set` is a downward closed subset of natural numbers($\mathbf{N}$).

```
TRACE : seq  = (# dom := nat , map:= systemState #)
```

Finally, we state the theorems which establish the correctness of the LCR algorithm in terms of execution sequences. These theorems are direct translations of the two correctness conditions given in section 6.2.1.

```
ELECTED : THEOREM Trace`map(n)(max)`3 = LEADER

NONE_ELSE: THEOREM FORALL(t:nat) : FORALL(i):
    i /= max IMPLIES TRACE`map(t)(i)`3 = UNKNOWN
```

The theorem `ELECTED` states that the status of the process with index `max` is set to `LEADER` at the end of the n^{th} round. The second theorem, `NONE_ELSE` states the uniqueness condition, i.e., the status of no process other than `max` is ever set to `LEADER` during the execution of the algorithm. These two theorems together imply the correctness of the LCR leader election algorithm. In the following section we outline the proofs of the above theorems.

6.5 Proving Correctness of the LCR Algorithm

In this section we establish the correctness of the LCR leader election algorithm by proving the above theorems in PVS. The formalizations of the proofs are based on the hand proofs given in [37].

6.5.1 Theorem `ELECTED`

In order to prove the theorem `ELECTED`, we begin by proving the following lemma which states that the value of the first component of the `PROCESS` tuple remains unaltered throughout the execution of the algorithm, for all the processes.

```
FIXED_C1 : LEMMA systemState(t)(i)`1 = systemState(t+1)(i)`1
INIT_UID_STAYS: COROLLARY TRACE`map(t)(i)`1 = name(i)
```

For any given process, we consider all the three possible relationships $>, =$ or $<$ between the `incoming` UID and the first component of the process, and in each case, using the appropriate axiom `MORE`, `EQUAL` or `LESS` respectively, we show that the first component is not altered by the algorithm. The corollary `INIT_UID_STAYS`, states that the unchanging first component of a process is indeed its initial UID.
Next, we need to show that if some `INDEX` is sent by a process to its neighbor then it must be the UID of some process in the ring. This is stated as the lemma `SEND_NAME`.

```
SEND_NAME: LEMMA systemState(t)(i)`2 = null OR
   EXISTS(j:INDEX): systemState(t)(i)`2 = name(j)
```

This lemma is proved by an induction on t. The base case is resolved by the `INITIALLY` condition, the induction step is broken into cases. We show that if the `incoming` is less than or equal to `this`1`, then nothing (`null`) is sent and the theorem is trivially true; but if `incoming` is greater than `this`1`, then the `incoming` value is sent. By the induction hypothesis it is already known that the `incoming` value, which was sent by the CCW neighbor of `this` in the previous round, *is* a UID of some process. Hence it follows that the value sent by `this` process in the present round is a valid UID.□

Now we prove the most important lemma of this section which states that at the end of t rounds, the maximum UID `nameMax`, moves t places CW, from the node with index `max`.

```
INTERMEDIATE: LEMMA FORALL(t:nat| t<=n) :
    TRACE`map(t)(CW(t,max))`2= nameMax

PREV_GETS: COROLLARY TRACE`map(n-1)(CCW(max))`2 = nameMax
```

This is also proved by induction on t. The base case `t=0`, is resolved using initialization condition. In the induction step, we make use of the axiom `MORE` which changes the *send* component of the process to the `incoming` UID. Using lemmas `INIT_UID_STAYS` and `MAX_EXISTS`, the induction step of the proof is completed. □

The corollary `PREV_GETS`, which is used later, is proved by simply instantiating `INTERMEDIATE` with `"(n-1)"`.

Having proved the necessary results, the proof of the theorem `ELECTED` can be accomplished in a few steps. First, we use the lemma `PREV_GETS` which states that the process with index `max` receives `nameMax` after `n-1` rounds. Using the rule `EQUAL` and simplifying with the lemma `INIT_STAYS`, we complete the proof of `ELECTED`. □

This theorem establishes that at the end of the n^{th} round, the process with maximum UID sets its *status* to `LEADER`. For proving the correctness of the LCR algorithm, another theorem is required which is discussed in the next section.

6.5.2 Theorem `NONE_ELSE`

This theorem establishes the uniqueness of the elected leader, it states that no other process except the one with the maximum UID, ever declares itself as the *leader*, i.e. their *status* is always set to `UNKNOWN`. This can be proved if we can somehow show that none of the other processes, ever receives its own UID in any round of computation. This is because a process is elected as the leader only when it receives its own UID from its CCW neighbor. This is stated as the following lemma:

```
NEVER_GETS: LEMMA
    FORALL(t:nat, i:INDEX|i /= max):
        TRACE`map(t)(CCW(i))`2 /= name(i)
```

This is proved using another lemma which states that in any round of computation, a process residing at a node, which is at a *distance* k from `max`, can only send the UIDs of a processes residing in one of the nodes in the set [max,k][1] .

```
SEND_WHAT_BELONGS: LEMMA
    FORALL(t:nat, p:INDEX,k:nat | k<n):
        TRACE`map(t)(CW(k,max))`2 = name(p)
            IMPLIES Away(p) <= Away(CW(k,max))
```

This lemma is proved by an induction on the round of computation t, followed by an induction on the *distance* k, of a node from `max`. The base cases are trivial. For the induction step, the `Away_subset` lemma is used to show that whatever is sent by a process at `CW(k,max)` in the t^{th} round is either `null`, in which case it does not correspond to any UID; or else it is an UID of another process sent to `this` process by its *CCW neighbor* i.e. `CCW(CW(k,max))`. Now, by the induction hypothesis, whatever is sent by `CCW(CW(k,max))` in the $(t-1)^{st}$ round is known to belong to `[max,CCW(CW(k,max))]`; hence the UID sent by `CW(k,max)` also belongs to `[max,CCW(CW(k,max))]`. The assertion that `[max,CCW(CW(k,max))]` is a subset of `[max,CW(k,max)]`, as stated in `Away_subset`, completes the proof of the above lemma. □

Next we prove another lemma which states that the CCW neighbor of a process `CW(k,max)`, for all k $\neq$ 0, can never send the UID of `CW(k,max)`.

```
NEVER_GETS_CW: LEMMA FORALL(k:nat | k > 0 AND k<n) :
    TRACE`map(t)(CCW(CW(k,max)))`2 /= name(CW(k,max))
```

The proof of this lemma is broken into two cases, the first case `(k=0)` is trivial, for the second case `(k>0)` we use the lemma `SEND_WHAT_BELONGS`. Use of lemma `SEND_NAME` completes the proof. □

[1] as defined in Section 6.3.2

Using `NEVER_GETS_CW`, The lemma `NEVER_GETS` is proved easily as the former states the same relation as the latter in terms of bound `INDEX` variable `i` instead of `(k:nat)`.

Finally, the theorem `NONE_ELSE` is proved by induction on t. The base case is easily resolved using the `INITIALLY` axiom which states that the initial *status* of all the processes is `UNKNOWN`. For the induction step, we use `NEVER_GETS`, which rules out the possibility that the incoming UID is equal to the UID of the receiving process. Then we are left with the cases where the incoming UID is `null` or not equal to the receiver UID. We consider all cases, one by one; if `incoming` is < `this‘1`, we use `LESS` to show that this does not lead to setting of *status*. For the other cases, namely `incoming` = `null` or `incoming` > `this‘1`, we use `LESS` and `MORE` respectively to show that in all these cases the *status* of the receiving process remains `UNKNOWN`. □

6.6 Summary

In this chapter we have presented the verification of a distributed leader election algorithm using PVS. Type checking in PVS enabled us to capture mis-specifications at an early stage and the prover takes care of some of the proof subgoals automatically. However, we found that the entire process of specification and verification requires considerable effort. Some twenty-five lemmas were proved to complete the proofs of the theorems which establish the correctness of the LCR algorithm. Apart from producing rigorous and readable proofs, another benefit derived from this exercise is the insight gained into the system and the working of the algorithm. In the next chapter we construct a customized interface for translating HIOA specifications to PVS theories which would simplify the usage of the PVS prover by partially automating the specification process.

Chapter 7

Generating PVS Theories from HIOA Specifications

7.1 Introduction

In the previous chapter we had introduced the process of specification and verification in PVS, now we present a scheme for automatic translation of HIOA specifications to PVS theories. But first, we briefly discuss the necessity of automating this process.

In Chapter 1 we had mentioned the importance of deductive techniques in verification of hybrid systems. Real life hybrid systems are complex, proving their properties involve a lot of detail. Establishing even seemingly simple results typically involves proving many lemmas as it is evident from the experience of researchers [39, 40, 41, 20, 19, 21, 42]. To organize and keep track of all the proofs, using an automatic theorem prover is desirable. Moreover common proof techniques can be administered automatically using proof strategies. On the other hand, for using a mechanical theorem prover the user is required to learn the input language of the prover, develop skills for writing specifications and for proving properties. Often it is regarded that the training required to acquire these skills are expensive, thus posing an impediment towards the adoption of deductive methods.

If the PVS specification of the system is generated automatically from the HIOA code then the user no longer has to spend time mastering the specification language of PVS. Secondly, being relieved of the extra effort involved in writing specifications the user can concentrate on the interactive proofs which demand understanding of the system and ingenuity.

The idea of customized interface over general purpose theorem prover, in itself is not new, a similar interface of PVS for the Lynch-Vaadrager timed automaton model called TAME (Timed Automaton Modeling Environment) has been developed [23, 22]. TAME automatically generates PVS specifications by instantiating predefined theory templates and provides a set of customized proof strategies for timed automata.

7.2 Revisiting Actions and Activities

Hybrid executions have been defined in Section 2.3.2 as alternating sequences of actions and trajectories: $\tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$, and a trace is the externally visible part of an execution. Most of the properties of hybrid automata are proved by inductively reasoning over the set of reachable states. This is accomplished by showing that an invariant which captures the property, say $\mathcal{I}$, is satisfied at all the start states $\mathbf{v} \in \Theta$ and then inductively showing that all the transitions(or trajectories) which are enabled(or operating) at $\mathbf{v}$ respect the invariant $\mathcal{I}$.

hybridautomaton `A`
  signatures
    input *Reset*
  variables
    internal *Counter* : `Int`
  transitions
    Reset
      eff *Counter* := *0.0*
    ...

Figure 7.1: *Reset* action

7.2.1 Actions

The actions in a HIOA specification define discrete jumps between points in the variable space of the automaton. The $effect$ part of an action defines the resulting valuation after the occurrence of the action in terms of the prior valuation. For example, the *Reset* action of automaton `A` (Figure 7.1) defines a set of jumps for each value of the internal variable *Counter* as shown in Figure 7.2. In this case, the individual jumps can be identified by initial values of *Counter* as shown by the labels $Result(1), Reult(2), \ldots$ etc. Had the

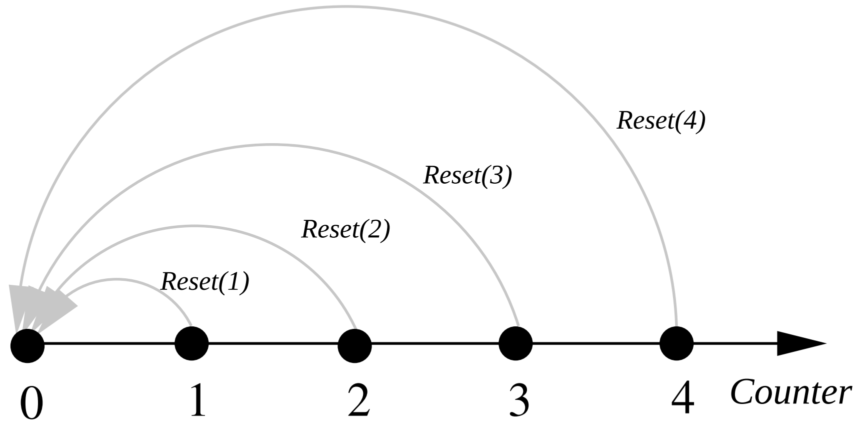

Figure 7.2: Actions defined by *Reset*

effect been a nondeterministic assignment, then each initial value of *Counter* would have given rise to a set of jumps consisting of all possible final values of the nondeterministic assignment.

7.2.2 Activities

The Lynch-Vaadrager model[43] of timed automaton involves only one kind of continuous evolution and that is the progress of the real-valued clock. Consequently the formalization of timed automaton in [23] incorporates a special parameterized *time elapse* action $\nu(\Delta t)$, the occurrence of which corresponds to progress of real time by Δt units. The hybrid automata model allows more general continuous evolutions. As we have seen in Section 4.4.6 each activity is associated with a progress function which determines a set of allowed trajectories over a particular region in the variable space defined by its activation condition. The progress function of a particular activity can be very specific or *tight* allowing only a single value for each variable at a particular instant; else it can be *loose* in which case the valuation is nondeterministically determined form a set. For example

trajectories
  activity *tight*
    precon $x \geq 0.0 \wedge doTight$
    evolve $x' = 2.0$
  activity *loose*
    precon $x \geq 0.0 \wedge \neg doTight$
    evolve $x' = [0.5, 2.5]$

Figure 7.3: *Tight* and *loose* activities

consider the two activities in Figure 7.3, the valuation of the variable x is determined by either one of the activities depending on the boolean variable $doTight$. For a trajectory starting with $x = x_0$, at every successive instant, the activity *tight* assigns a particular value $x(\Delta t) = x_0 + 2 \times \Delta t$ whereas the activity *loose* nondeterministically chooses a value from the set $[x_0 + 0.5 \times \Delta t, x_0 + 2.5 \times \Delta t]$.

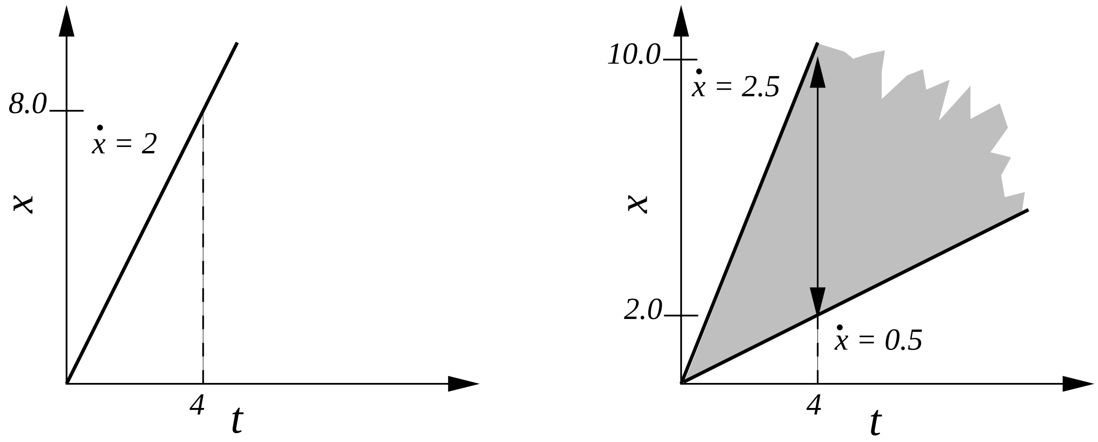

Figure 7.4: *Tight* and *loose* trajectories

An activity defines a set of trajectories or *flows* over a region in the state space either loosely or tightly depending on its progress function. For example, the activity *tight* defines a different trajectory for each value of $x_0 >= 0$ and for each different length of the interval. Figure 7.5 shows the set of *flows* defined by *tight*, (a) with same Δt but different x_0's and (b) with different Δt's and constant x_0. So, a particular *flow* of a tight activity can be specified by these two parameters; the starting state(x_0) and the duration (Δt) as shown in Figure 7.5(c). In the case of the activity *tight*, the assignment being

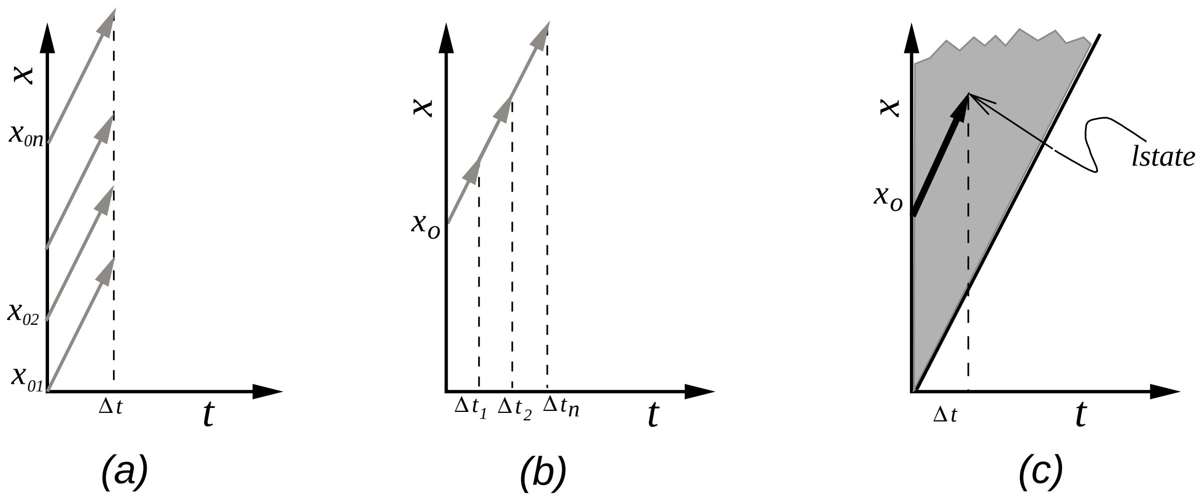

Figure 7.5: Trajectories defined by *tight*

deterministic, the parameters x_0 and Δt determine the *lstate* of the trajectory according to the equation $x(\Delta t) = x_0 + 2 \times \Delta t$. Whereas for the activity *loose*, the parameters determine a set of flows and instead of a single final state, a set of *lstates* which satisfy the nondeterministic assignment is obtained(see Figure 7.6).

7.3 Revisiting Executions

In terms of the jumps and flows defined above, a hybrid execution fragment can be written as a sequence : $\zeta = \alpha_0(\mathbf{v}_0, \Delta t_0), a_1(\mathbf{v}_1), \alpha_2(\mathbf{v}_2, \Delta t_2), a_3(\mathbf{v}_3), \alpha_4(\mathbf{v}_4, \Delta t_4), \cdots$, where each $\alpha_i(\mathbf{v}_i, \Delta t_i)$ is a flow or a trajectory defined by the activity α_i and each $a_i(\mathbf{v}_i)$ is a jump defined by the action a_i. An execution fragment is an execution if $\mathbf{v}_0$ is a start

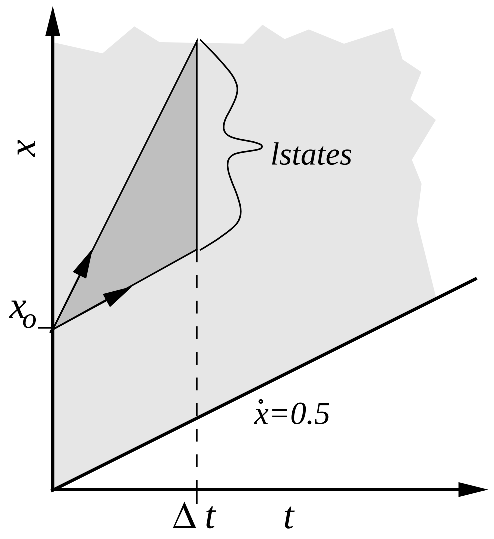

Figure 7.6: A trajectory defined by *loose*

state of the automaton. Now we shall define a wrapper over the elements of the execution sequence and redefine the hybrid execution in terms of these *execution elements*.

Let us denote the effect of a jump by eff and the progress function of a flow by *evolve*. Also the first and the last sates of a jump or a flow are denoted by $fstate$ and $lstate$ respectively. In case of nondeterministic assignments, the lstate is a set of states. Obviously,

- $fstate(\alpha_i(v_i, \Delta t_i)) = v_i$
- $lsate(\alpha_i(v_i, \Delta t_i)) = \{v \mid v \in evolve(\alpha_i(v_i, \Delta t_i))(\Delta t_i)\}$
- $fstate(a_i(v_i)) = v_i$
- $lstate(a_i(v_i)) = \{v \mid v \in eff(a_i(v_i))(v_i)$

An execution element e of an automaton is either a flow $\alpha_i(v_i, \Delta_i)$ or a jump $a_i(v_i)$ such that α_i is an activity and a_i is an action of the automaton. An execution of the hybrid automaton is a sequence $\zeta = e_0, e_1, e_2, \cdots$, of execution elements satisfying the following conditions.

- $fstate(e_0) \in \Theta$
- $\exists i \; e_0 = \alpha_i$
- $\forall i \; lstate(e_i) = fstate(e_{i+1})$
- $\forall i \, (e_i = jump \Rightarrow e_{i+1} = flow) \; \wedge \; (e_i = flow \Rightarrow e_{i+1} = jump)$

7.4 PVS Specification

In this and the following sections, we develop the PVS specification for hybrid automata. First we give the template theory with data objects defining the actions and the activities of the hybrid automaton. This template is instantiated by the user specifications given in HIOA. We also develop the underlying theories in PVS which define the execution elements, the executions and the lemmas required to prove theorems by invariant assertions.

Prior to developing a theory for the automaton, we need to define time. In the hybrid automata model time is taken as $\mathsf{R}^{\geq 0}$ and since the terminal trajectory in an execution sequence can have an infinitely long period, thus we take time as $\mathsf{R}^{\geq 0} \cup \{\infty\}$. This union type is defined in the *time* data type:

```
time: datatype
  begin
    fintime(dur: {r: real | r ≥ 0}): fintime?
    infinity: inftime?
  end time
```

In PVS, data types are defined by *constructors* and *recognizers*. The data type *time* has two constructors; *fintime*, has a non-negative real parameter *dur* and a recognizer *fintime*? and the second constructor *infinity* has no parameter and a recognizer *infinity*?.

The algebraic and comparison operations on *time* are defined in the PVS theory *Time* given in Figure 7.7. A parameterized type $interval(t_1 : (fintime?), t_2 : time)$ is defined,

$Time$: **theory**
begin
importing *time*
t_1, t_2, z: **var** *time*
$zero$: *time* $=$ $fintime(0)$;

$\leq(t_1,\ t_2)$: *bool* $=$ **if** $fintime?(t_1) \wedge fintime?(t_2)$ **then** $dur(t_1) \leq dur(t_2)$
else $inftime?(t_2)$ **endif**;

$\geq(t_1,\ t_2)$: *bool* $=$ **if** $fintime?(t_1) \wedge fintime?(t_2)$ **then** $dur(t_1) \geq dur(t_2)$
else $inftime?(t_1)$ **endif**;

$<(t_1,\ t_2)$: *bool* $=$ **if** $fintime?(t_1) \wedge fintime?(t_2)$ **then** $dur(t_1) < dur(t_2)$
else $\neg\ (inftime?(t_1)) \wedge inftime?(t_2)$ **endif**;

$>(t_1,\ t_2)$: *bool* $=$ **if** $fintime?(t_1) \wedge fintime?(t_2)$ **then** $dur(t_1) > dur(t_2)$
else $\neg\ (inftime?(t_2)) \wedge inftime?(t_1)$ **endif**;

$t_1 + t_2$: *time* $=$ **if** $fintime?(t_1) \wedge fintime?(t_2)$ **then** $fintime(dur(t_1) + dur(t_2))$
else *infinity* **endif**;

$t_1 - t_2$: $(fintime?)$: *time* $=$ **if** $fintime?(t_1) \wedge\ dur(t_1) > dur(t_2)$
then $fintime(dur(t_1) - dur(t_2))$ **else** *infinity* **endif**;

$interval?(t_1,\ t_2)(z)$: *bool* $=$ $(t_1 \leq z \wedge z \leq\ t_2)$

$interval(t_1{:}(fintime?),\ t_2{:}time)$: **type** = $\{z \mid\ interval?(t_1,\ t_2)(z)\}$

end *Time*

Figure 7.7: *Time* theory

it represents an interval $[t_1, t_2]$ in $\mathsf{R}^{\geq 0}$. We shall use this *interval* type to define domain of trajectories. If the second parameter of an interval is $infinity$, then it represents an open interval of time $[t, \infty)$.

7.4.1 The Template

Figure 7.8 shows the template theory we have developed for defining hybrid I/O automata in PVS. This template is instantiated by filling out the details in the parts indicated by "$\langle\cdots\rangle$". The template imports the $Time$ theory along with other theories(if any) which are used to define its components.

First the *stateVector* type is defined, which is typically a tuple or a record of the

```
⟨hybrid_automaton_name⟩: theory
 begin
  importing Time

  stateVector: type =  ⟨state_tuple⟩
  vectors: type = [# basic: stateVector, now: (fintime?) #]
  validState?(v: vectors): bool = ⟨predicate⟩

  jumps: datatype
   begin
    ⟨action_name⟩(fstate:vectors,⟨parameters⟩) : ⟨recognizer⟩
   end jumps

  enabled_general(a: jumps, v: vectors): bool =  fstate(a) = v
  enabled_specific(a: jumps, v: vectors): bool =
    cases a of
        ⟨action_name⟩:⟨pre_condition⟩ endcases

  eff(a: jumps, v: vectors): vectors =
    cases a of
        ⟨action_name⟩:⟨assigments⟩  endcases;

  enabled(a: jumps, v: vectors): bool = enabled_general(a, v)
                  ∧ enabled_specific(a, v) ∧ validState?(eff(a, v));

  lstate(a):vectors =  eff(a,fstate(a))

  trajectories: datatype
   begin
    ⟨activity_name⟩(fstate: vectors, period: time):⟨recognizer⟩
   end trajectories

  sameActivity?(f, f1: trajectories): bool = ⟨recognizer_matching_function⟩

  ftime(f: trajectories): time = now(fstate(f))
  ltime(f: trajectories): time =  ftime(f)+period(f)

  operating?(f: trajectories, v: vectors):  bool = validState?(v) ∧
   cases f of
      ⟨activity_name⟩:⟨activation_condition⟩  endcases

  evolve(f: trajectories): [(interval(ftime(f), ltime(f))) → vectors] =
    cases fof
        ⟨activity_name⟩:⟨assigments⟩ endcases;

  lstate(f: trajectories): vectors =  evolve(f)(ltime(f))

  possibleFlow?(f: trajectories): boolean =  ∀ (t: (interval?(ftime(f), ltime(f)))):
         (∀ (a): ¬ enabled(a, evolve(f)(t)) ∧
         (∀ (f1): ¬ sameActivity?(f1, f)): ¬ operating?(f1, evolve(f)(t)) ∧
         validState?(evolve(f)(t)))) ∧  operating?(f, evolve(f)(t)

  validFlow?(v1: vectors, f: trajectories,v2: vectors): boolean =
   fstate(f) = v1 ∧ lstate(f) = v2 ∧ possibleFlow?(f,v1)  ∧ operating?(f,v1)

  start(s: vectors): boolean = ⟨initial valuation predicate⟩

end ⟨hybrid_automaton_name⟩
```

Figure 7.8: HIOA specification in PVS : Template theory

variables of the hybrid automaton. The record type *vectors* is defined by augmenting the *stateVector* with a variable *now* which keeps the temporal information of the state. All other time dependent variables in the definition of the automaton are matched with *now*. A constraint on the set of allowed valuations of the variables can be imposed by the *validState?* predicate on the vectors.

The *jumps* data type defines the jumps for the actions of the HIOA specification. The jumps corresponding to each action in the signature of the HIOA specification are defined by a separate constructor and a corresponding recognizer. For simple (unparameterized) actions, the constructor has only one parameter *fstate* of type *vectors*, corresponding to the initial valuation at which that particular jump occurs and the constructor defines a set of jumps for different values of *fstate*. For value-passing(parameterized) actions, the constructors have one parameter for each of the parameters of the corresponding action, in addition to *fstate*.

A jump a can occur at a particular valuation v if $\mathit{fstate}(a) = v$. This condition, which is common for the occurrence of any jump is stated in *enabled_general*. And the specific preconditions of the individual actions are defined by the *enabled_specific* function. The *eff* function states the effects of the jumps. If we use the symbols v and v' to denote the valuations before and after the occurrence of a jump, respectively, then the function *eff* defines the mapping $v \xrightarrow{a} v'$ for every jump a of the automaton. The predicate *enabled(a,v)* determines if a particular jump a can occur at a given valuation v. This definition uses *enabled_general*, *enabled_specific* and a third condition which ensures that the valuation resulting from the occurrence of a, i.e., $\mathit{eff}(a, v)$, is valid.

The flows of the automaton are defined by the *trajectories* data type. Each activity of the HIOA specification has a corresponding constructor with parameters *fstate* and *period*. Different values of *fstate* and *period* generate the whole set of flows of the particular activity as shown in Section 7.2.2. The functions *ftime* and *ltime* return the start and the terminal times of a flow and *sameActivity?* checks if two flows are for the same activity.

The activation conditions of the different activities are defined by the *operating?* function which takes into account the fact that a trajectory can operate only in valid states. A flow f maps the interval [*ftime(f)*, *ltime(f)*] to points in the variable space. These mappings are given by the *evolve* function, for each individual activity.

A given flow f is a $possible\ flow$ if it satisfies the following conditions:

1. The trajectory remains operating over its domain interval $\mathcal{I}$.
2. None of the actions is enabled in $\mathcal{I}$.
3. No *other* activity operates in $\mathcal{I}$; i.e., none of the trajectories created by a different activity [1] is operating in $\mathcal{I}$.
4. All valuations to which f maps $\mathcal{I}$ are valid states.

A possible flow f is a valid flow between two valuations v_1 and v_2 if $v_1 = fstate(f)$ and $v_2 \in lstate(f)$. These conditions are expressed in the predicates *possibleFlow?* and *validFlow?* respectively. The set of starting states of the automaton is specified by the predicate *start*.

7.4.2 Theory for Execution Elements

The *execution_element* theory(Figure 7.9) defines execution elements for an automaton and the wrapper functions. It imports the instance of the template theory containing the definitions of the automaton's components. The data type *execEle* defines two constructors for two kinds of elements, jumps and flows. The *guard(e, v)* function evaluates if the execution element e can occur at a given valuation v. The valuations which are ever reached in a particular execution element are given by the *touched* function. For a jump $a(v)$, the reachable valuations constitute the set $\{v, eff(a(v))\}$. And for a flow, $f : \mathcal{I} \longrightarrow val(V)$, it is given by the set $\{v \mid \exists\, t \in \mathcal{I}, evolve(f)(t) = v\}$. The functions

[1] This naturally does not happen if all the activities are disjoint

```
execution_element: theory
 begin

  importing ⟨hybrid_automaton_theory⟩

  execEle: datatype
   begin
    flow(fl: trajectories): flow?
    jump(jp: actions): jump?
   end execEle

  guard(ee: execEle, v: vectors): boolean =
      cases ee of flow(f): possibleFlow?(fl(ee), v), jump(a): enabled(jp(ee), v)
        endcases;

  touched(ee: execEle, v: vectors): boolean =
      cases ee
        of jump(a): fstate(a) = v ∨ lstate(a) = v,
           flow(f): ∃ (t: (interval?(ftime(f), ltime(f)))): v = evolve(f)(t)
        endcases;

  fstate(ee: execEle): vectors =
      cases ee of flow(f): fstate(fl(ee)), jump(a): fstate(jp(ee))
        endcases;

  lstate(ee: execEle): vectors =
      cases ee of flow(f): lstate(fl(ee)), jump(a): lstate(jp(ee))
        endcases;

  ftime(ee: execEle): time =
      cases ee of flow(f): ftime(fl(ee)), jump(a): now(fstate(jp(ee)))
        endcases;

  ltime(ee: execEle): time =
      cases ee of flow(f): ltime(fl(ee)), jump(a): now(fstate(jp(ee)))
        endcases;

  finiteEle?(ee: execEle): boolean = ltime(ee) ≠ infinity

  terminalFlow?(ee: execEle): boolean =
      flow?(ee) ∧ (∀ (a: execEle | jump?(a)): ¬ guard(a, lstate(ee)))
 end execution_element
```

Figure 7.9: Theory exec_Element

fstate, lstate, ftime and *ltime* give the first and the last states of an execution element and the corresponding times.

A flow f is regarded as a terminal flow if none of the jumps of the automaton are enabled at its *lstate*. This is captured in the *terminalFlow?* predicate. For an execution sequence of finite length, the last execution element has to be a terminal flow.

7.4.3 Theory for Executions

Executions of automata were defined as sequences of execution elements satisfying certain properties. In this section we develop the PVS specification of executions. We also state and prove a theorem which is used to prove invariants by induction on the length of executions. The complete *executions* theory is given in Appendix C.

We use the notion of a sequence in PVS as given in [44] i.e., a record type consisting of an index-set *dom*, and a mapping *map* from *dom* to execution elements. A *fragment* is a sequence with alternating jump and flow elements with connected first and last states.

$$\begin{array}{l}
\mathit{fragment}\colon \textbf{type} = \ \{c\colon \mathit{seq} \mid \neg\ \mathit{empty?}(c`\mathit{dom}) \wedge \\
\quad (\forall\ (k\colon \mathit{nat} \mid c`\mathit{dom}(k) \wedge c`\mathit{dom}(k+1))\colon \\
\quad\quad \mathit{lstate}(c`\mathit{map}(k)) = \mathit{fstate}(c`\mathit{map}(k+1)) \wedge \mathit{flow?}(c`\mathit{map}(k)) \supset \\
\quad\quad\quad \mathit{jump?}(c`\mathit{map}(k+1)) \wedge \mathit{jump?}(c`\mathit{map}(k)) \supset \ \mathit{flow?}(c`\mathit{map}(k+1)))\}
\end{array}$$

An *execution* is a fragment satisfying two conditions; (1) it begins with a flow which has one of the start states as its *fstate*, and (2) if the fragment sequence is finite[2] then it must end with a terminal flow.

$$\begin{array}{l}
\mathit{execution}\colon \textbf{type} = \ \{c\colon \mathit{fragment} \mid (c`\mathit{dom}(0) \wedge \mathit{flow?}(c`\mathit{map}(0)) \wedge \\
\quad \mathit{start}(\mathit{fstate}(c`\mathit{map}(0)))) \wedge \\
\quad\quad (\mathit{finiteSeq?}(c) \supset (\forall\ (x\colon \mathit{nat} \mid \mathit{largest?}(c`\mathit{dom})(x))\colon \mathit{terminalFlow?}(c`\mathit{map}(x))))\}
\end{array}$$

The behavior of a specified automaton is studied in terms of its executions. Various types of executions $finite$, *admissible*, and $Zeno$ for timed systems have been defined in [9]. An

[2]The function *finiteSeq?(c)*, returns true if the domain of the argument sequence c is finite.

execution is *finite* or *closed* if the domain of its sequence it finite and its final trajectory has a finite period.

$$\textit{finiteExec?}(c\colon \textit{execution})\colon \textit{boolean} = \textit{finiteSeq?}(c) \wedge \\ (\forall\ (x\colon \textit{nat} \mid \textit{largest?}(c`\textit{dom})(x))\colon \textit{finiteEle?}(c`\textit{map}(x)))$$

An admissible execution is one in which the time progresses to infinity. There are two subclasses of admissible executions; (1) the domain set of the execution is finite but its final trajectory has an infinite period and (2) the domain set is infinite (infinite sequence) and the sequence satisfies the *greatest lower bound* property. The greatest lower bound property ensures that arbitrarily large number of execution elements cannot occur within a finite time interval. These two conditions are captured in the predicates *finSeqInfExec?* and *infSeqAdmissExec?* respectively, and are used to define admissible executions, as given below.

$$\textit{finSeqInfExec?}(c\colon \textit{execution})\colon \textit{boolean} = \textit{finiteSeq?}(c) \wedge \\ (\forall\ (x\colon \textit{nat} \mid \textit{largest?}(c`\textit{dom})(x))\colon \neg\ \textit{finiteEle?}(c`\textit{map}(x)))$$

$$\textit{infiniteExec?}(c\colon \textit{execution})\colon \textit{boolean} = \neg\ \textit{finiteExec?}(c)$$

$$\textit{is\_glb}(z\colon \textit{time},\ c\colon (\textit{infiniteSeq?}),\ k\colon \textit{nat})\colon \textit{boolean} = \\ \textit{ltime}(c`\textit{map}(k)) \le z \wedge (\forall\ (n)\colon n > k \supset z \le \textit{ltime}(c`\textit{map}(n)))$$

$$\textit{has\_glb}(z\colon \textit{time},\ c\colon (\textit{infiniteSeq?}))\colon \textit{boolean} = \exists\ (k)\colon \textit{is\_glb}(z,\ c,\ k)$$

$$\textit{infSeqAdmissExec?}(c\colon \textit{execution})\colon \textit{boolean} = \\ \textit{infiniteSeq?}(c) \wedge (\forall\ (z\colon \textit{time})\colon \textit{has\_glb}(z,\ c))$$

$$\textit{admissExec?}(c\colon \textit{execution})\colon \textit{boolean} = \\ \textit{finSeqInfExec?}(c) \vee \textit{infSeqAdmissExec?}(c)$$

Of all possible executions, only the *admissible* ones are of practical importance as they represent executions of realizable systems. Behavorial properties of hybrid automata are proved by asserting invariants over reachable valuations. Now we present a set of definitions which help us to prove invariants for infinite execution sequences. The corresponding set of definitions for finite-admissible executions are given in Appendix C.

The set of reachable valuations of an execution is inductively defined by the function *n_reachable* in terms of the domain index. The total set of reachable valuations, given by *c_reachable*, consists of valuations which are reached by at least one execution element of the sequence. An invariant over the variables is a predicate and it defines a set of valuations over which the the predicate holds. In order to prove that an automaton satisfies a certain invariant, the standard technique is to show that the invariant holds in the starting states and that every transitions of the automaton respects the invariant. Which means for a given transition, if the invariant holds in the state before the occurrence of the transition, then it must also hold for the state which results after the transition. This technique of induction is translated into two conditions which together imply the validity of any invariant.

$$
\begin{array}{l}
\mathit{n\_reachable}(v\texttt{:}\ \mathit{vectors}\texttt{,}\ c\texttt{:}\ (\mathit{infSeqAdmissExec?})\texttt{,}\ n\texttt{:}\ \mathit{nat})\texttt{:}\ \ \textbf{inductive}\ \mathit{bool} =\\
\quad \textbf{if}\ n = 0\ \textbf{then}\ \mathit{touched}(c\text{`}\mathit{map}(0)\texttt{,}\ v)\\
\quad \textbf{else}\ \mathit{n\_reachable}(v\texttt{,}\ c\texttt{,}\ n-1) \vee \mathit{touched}(c\text{`}\mathit{map}(n)\texttt{,}\ v)\ \textbf{endif}\\[1em]
\quad \mathit{c\_reachable}(v\texttt{:}\ \mathit{vectors}\texttt{,}\ c\texttt{:}\ (\mathit{admissExec?}))\texttt{:}\ \mathit{boolean} = \exists\ (n)\texttt{:}\ \mathit{n\_reachable}(v\texttt{,}\ c\texttt{,}\ n)\\[1em]
\quad \mathit{satisfies}(I\texttt{:}\ \mathit{invariant}\texttt{,}\ c\texttt{:}\ (\mathit{infSeqAdmissExec?}))(k)\texttt{:}\ \mathit{boolean} =\\
\qquad \forall\ (v)\texttt{:}\ \mathit{n\_reachable}(v\texttt{,}\ c\texttt{,}\ k) \supset I(v)\\[1em]
\quad \mathit{base}(I\texttt{:}\ \mathit{invariant})\texttt{:}\ \mathit{boolean} =\ \forall\ (c\texttt{:}\ (\mathit{infSeqAdmissExec?}))\texttt{:}\ \mathit{satisfies}(I\texttt{,}\ c)(0)\\[1em]
\quad \mathit{inductstep}(I\texttt{:}\ \mathit{invariant})\texttt{:}\ \mathit{boolean} = \forall\ (c\texttt{:}\ (\mathit{infSeqAdmissExec?}))\texttt{:}\\
\qquad\quad \forall\ (k\texttt{:}\ \mathit{nat})\texttt{:}\ \mathit{satisfies}(I\texttt{,}\ c)(k) \supset \mathit{satisfies}(I\texttt{,}\ c)(k+1)\\[1em]
\quad \mathit{induction\_lemma}\texttt{:}\ \textbf{lemma}\ \ \forall\ (I\texttt{:}\ \mathit{invariant})\texttt{:}\ \ \mathit{base}(I) \wedge \mathit{inductstep}(I) \supset\\
\qquad\quad (\forall\ (c\texttt{:}\ (\mathit{infSeqAdmissExec?}))\texttt{:}\ \ \forall\ (v\texttt{:}\ \mathit{vectors})\texttt{:}\ \mathit{c\_reachable}(v\texttt{,}\ c) \supset I(v))
\end{array}
$$

First the *satisfies(k)* function is defined, which holds for a given invariant and execution, if all the valuations touched by the k^{th} element in the execution satisfy the invariant. Using the *satisfies* function, the *base* and the *inductstep* conditions are stated. Finally we state the induction lemma, which says for any invariant, if the the *base* and the *inductstep* conditions hold for all executions of an automaton, then the invariant holds for all reachable valuations of the automaton. The complete proof of *induction_lemma* and the corresponding lemma for finite length admissible executions can be found in the PVS dump file: http://kohinoor.csa.iisc.ernet.in/ mitras/main/research/hybrid-theory.dmp

7.5 Summary

In this chapter we have developed theories in PVS to specify hybrid automata and to deduce their properties. Proving behavorial properties of hybrid automata, generally involves induction over the length of the executions of automata. In order to use theorem provers like PVS, we have to specify the possible executions of the automaton from the HIOA specification, which is basically a state transition model. The elements of an execution are the jumps and the flows which are defined by the actions and the activities of HIOA respectively. The template theory, which can be instantiated automatically from a HIOA specification and the *execution_elements* theory together define the possible execution elements of a given automaton. Once we have the execution elements defined, the *executions* theory can be to prove invariants of the automaton by inductive reasoning. Specialized strategies for proving such invariants can also be incorporated into the PVS prover.

With this chapter we conclude the discussion on HIOA, its front end tools and its interface to PVS which we have developed. In the next chapter we shall take up two case studies to illustrate the use of the language and some of the tools.

Chapter 8

Case Studies

8.1 Introduction

In this chapter we study the specification of two hybrid systems using HIOA. In the first problem, we are given the functional diagram of the Longitudinal Axis Controller of a combat aircraft, from which we develop its HIOA specification. The second problem is a vehicle-controller system given as an MMT-specification [11].

8.2 Longitudinal Axis Controller

The functional block diagram of the Longitudinal Axis Controller(LAC) of a typical combat aircraft is shown in Figure 8.1. The primary input, U1 is the stick movement made by the pilot for commanding the normal acceleration/pitch rate of the aircraft and the output Y corresponds to the shaped pilot command which is used in the control law. U2 is a discrete input indicating the undercarriage position in the aircraft. In this section we develop the HIOA specification of this system.

The entire system is made up of two kinds of basic blocks: (1) the nonlinearities $G_1 \ldots G_5$ and (2) the lead-lag filters, F_1 and F_2. The blocks of the same kind are identical in all

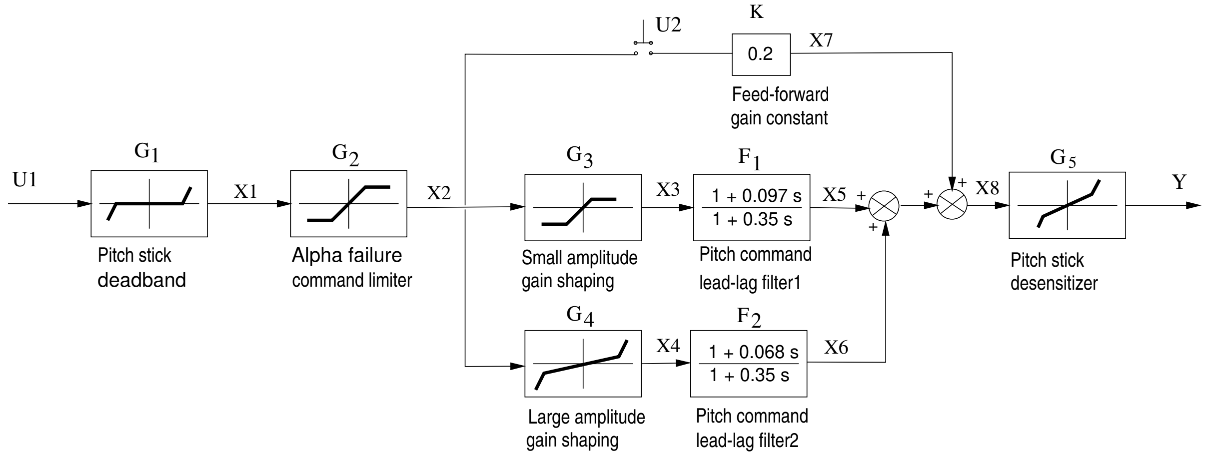

Figure 8.1: Longitudinal Axis Controller

respects, except for their input/output variables and their characteristic constants [1]. First we develop HIOA specifications for a generic nonlinearity and a generic filter. Then, by appropriately altering the input/output variables and the characteristic constants of the generic specifications, we get the specifications of the individual blocks. The entire system is specified by composing the specifications of the individual blocks.

8.2.1 Generic Nonlinearity

The blocks G_1 to G_5 in Figure 8.1 are nonlinearities, their output signals are determined from their inputs according to a nonlinear function. The nonlinear functions are piecewise linear and they have a common form as shown in Figure 8.2. The domain of the generic nonlinear function is: $[b2, a2]$, and this region is divided into three parts $[b2, b1), [b1, a1]$, and $(a1, a2]$, having different slopes.

The HIOA specification of a generic nonlinearity G_i has been given in Figure 8.3. We use symbols with suffix i to denote the characterizing constants of the nonlinear function and the input/output variables of the i^{th} block. Table 8.1 gives the correct meaning of these

[1] By characteristic constants we mean the constants in the transfer function for the lead-lag filters and the constants defining the nonlinear function for the nonlinearity blocks.

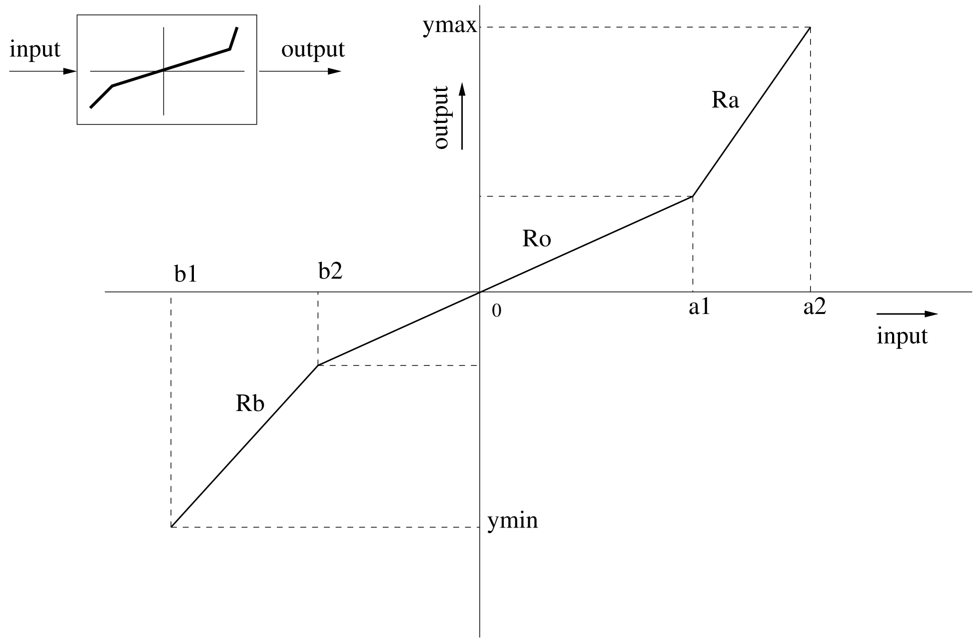

Figure 8.2: Generic nonlinear function

symbols for each nonlinearity in the system.

The input variable x_i is an analog signal i.e., it changes continuously with time, or at most has finite number discontinuities. The variable *now* records the real time for the block, and the variable y_i is the output signal.

The solitary action, *blankAction* has no effect and is never enabled. It is written as present HIOA grammar does not allow complete elimination of actions and transitions. We have defined three activities *b2b1,b1a1* and *a1a2* corresponding to the three sections of the nonlinear function, with the progress function of each activity defining the evolution of the output variable y_i in terms of the input variable x_i. The fourth activity *b2a2* defines the output when the input variable is beyond the domain of the function. According to the given specification of the nonlinearities, out-of-range inputs are frozen at the limit values; hence when the input x_i is greater than $a2_i$, the output is $ymax_i$ and when x falls below $b2_i$, the output is $ymin_i$.

The HIOA specification of the nonlinearities G1-4 are obtained by substituting the correct variables and constants from table 8.1 into the generic specification of Figure 8.3.

hybridautomaton G_i `% i = 1,2,3,4,5`
  signatures
    internal *blankAction* `% dummy action`
  variables
    input analog now, x_i : `Real` ,
    output analog y_i : `Real` $:= 0.0$
  transitions
    blankAction **pre** false
  trajectories
    activity *b2b1*
      precon $(x_i \geq b2_i) \wedge (x_i < b1_i)$
      evolve $y_i := (b1_i * Ro_i) + (x_i - b1_i) * Rb_i$
    activity *b1a1*
      precon $(x_i \geq b1_i) \wedge (x_i \leq a1_i)$
      evolve $y_i := x_i * Ro_i$
    activity *a1a2*
      precon $(x_i > a1_i) \wedge (x_i \leq a2_i)$
      evolve $y_i := (a1_i * Ro_i) + (x_i - a1_i) * Ra_i$
    activity *b2a2*
      precon $(x_i > a2_i) \vee (x_i < b2_i)$
      evolve
        if $(x_i > a2_i)$ **then** $y_i := a1_i * Ro_i + (a2_i - a1_i) * Ra_i$
        elseif $(x_i < b2_i)$ **then** $y_i := b1_i * Ro_i + (b2_i - b1_i) * Rb_i$ **fi**

Figure 8.3: HIOA code for generic Nonlinearity

Component	Variables		Constants						
	Input(x_i)	Output(y_i)	$a1_i$	$a2_i$	$b1_i$	$b2_i$	Ra_i	Rb_i	Ro_i
G_1: deadband	U1	X1	*0.7*	*35.0*	*-0.7*	*-40.0*	*0.874*	*1.043*	*0.0*
G_2: limiter	X1	X2	*30.0*	*40.0*	*-41.0*	*-50.0*	*0.0*	*0.0*	*1.0*
G_3: gain shaper	X2	X3	*10.0*	*30.0*	*-10.0*	*-41.0*	*0.0*	*0.0*	*1.0*
G_4: gain shaper	X2	X4	*10.0*	*30.0*	*-10.0*	*-41.0*	*1.0*	*1.0*	*0.0*
G_5: desensitizer	X8	Y	*23.0*	*30.0*	*-27.0*	*-41.0*	*1.714*	*1.419*	*0.783*

Table 8.1: Variables and constants for Nonlinearities $G_1 - G_5$

```
hybridautomaton G5
  signatures
    internal U2On , U2Off
  variables
    input analog now, X2, X5, X6, : Real  ,
    input U2 : Bool  ,
    output analog Y : Real  :=0.0,
    internal analog K, X7, X8 : Real  ,
  transitions
    U2On   pre U2 ∧ ¬(K =2.0)   eff K :=2.0
    U2Off  pre ¬ U2 ∧ ¬(K =0.0)   eff K :=0.0
  trajectories
    activity always
      evolve X7 := K * X2 ; X8 := X5 + X6 + X7
    activity b2b1
      precon (X8 ≥ -41.0) ∧ (X8 <-27.0) evolve Y:= -22.8 + (X8 + 27.0)*1.429
    activity a1a2
      precon (X8 ≤ 30.0) ∧ (X8 > 23.0) evolve Y:= 18.0 + (X8 - 18.0)*1.714
    activity b1a1
      precon (X8 ≥ -27.0) ∧ (X8 ≤ 23.0) evolve Y := X8 * 0.783
    activity b2a2
      precon (X8 > 30.0) ∨ (X8 < -41.0)
      evolve if (X8 > 30.0) then Y := 30.0
            elseif (X8 < 41.0) then Y := -41.0 fi
```

Figure 8.4: HIOA code for Desensitizer (G_5) block

The specification of the G_5 block (Figure 8.4) is slightly different from that of the other nonlinearities; it takes as input the small amplitude signal(*X5*), the large amplitude signal(*X6*), the undercarriage state (U2), and generates the final output (*Y*) of the system.

When the state of U2 changes from `Off` to `On`, the action *U2On* occurs setting the value of internal variable *K* to the feedforward gain(*Fg*). Similarly, when U2 changes from `On` to `Off`, the action *U2Off* sets *K* to *0.0*, opening the feedforward path.

The *aibj* activities of G_5 are identical to those of the other shaping blocks, they compute the output(*Y*) for four different regions of the input(*X8*). In addition to these four activities G_5 has an activity called *always*, determining the contribution of the feedforward path(*X7*) and the actual input(*X8*), in terms of the different input variables.

8.2.2 Generic Filter

In this section we develop the HIOA specification of a generic lead-lag filter, which will be used later to specify the pitch command filters of Figure 8.1. The input-output relation

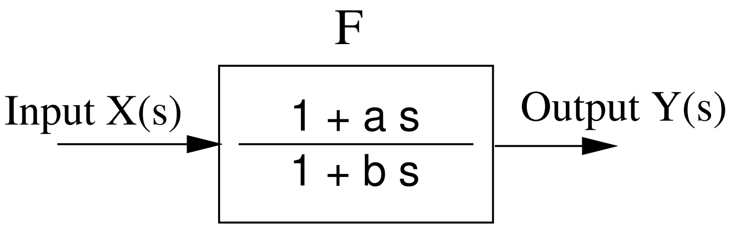

Figure 8.5: Generic lead-lag filter

of the filter is given by a transfer function, which is transformed into the time domain and expressed in terms of linear ordinary differential equations.

$$\frac{Y(s)}{X(s)} = \frac{1+as}{1+bs} = \frac{a}{b} + \left(1 - \frac{a}{b}\right)\frac{1}{1+bs}$$

$$Y(s) = Y_1(s) + Y_2(s) \ \ where, \ \ Y_1(s) = \frac{a}{b}X(s), \ \ Y_2(s) = \left(1 - \frac{a}{b}\right)\frac{1}{1+bs}X(s)$$

Taking the Laplace inverse transform, we get

$$y_1 = \frac{a}{b}x \ ; \ \dot{y_2} = \frac{1}{b}\left(1 - \frac{a}{b}\right)x - \frac{1}{b}y_2 \ \ and \ \ y = y_1 + y_2$$

The above equations describe the continuous evolution of the output variable y as a combination of algebraic equations and linear ODEs which can be written as progress function of an activity.

The filter blocks F_1 and F_2 are identical, excepting the constants in their transfer functions and their input/output variables. The HIOA code for a generic filter F_i is given

```
hybridautomaton F_i                      % i = 1, 2
  signatures
    internal blankAction                 % dummy action
  variables
    input analog x_i : Real   ,
    output analog y_i : Real   :=0.0,
    internal analog y1_i, y2_i : Real
  transitions                            % dummy transition
    blankAction   pre false              % never occurs
  trajectories
    activity always                      % always operating
      evolve y1_i := (a_i/b_i)*x_i ;
             y2_i':=(((b_i - a_i)/(b_i * b_i)) * x_i) - ((1.0/b_i) * y2_i);
             y_i := y1_i + y2_i
```

Figure 8.6: HIOA code for generic lead-lag filter

in Figure 8.6, the specifications for F_1 and F_2 are obtained by replacing the symbolic constants a_i, b_i and the external variables x_i and y_i according to Table 8.2. F_i represents a primitive hybrid automaton with input and output variables x_i and y_i respectively. The only action *blankAction* is a syntactic filler and the activity *always* defines the evolution of the locally controlled variables in terms of the equations derived above. Solving these equations would yield the set of allowed trajectories or flows of the locally controlled variables of the automaton.

Component	Variables		Constants	
	Input(x)	Output(y)	a	b
F1: lead-lag filter-1	X3	X5	*0.055*	*0.350*
F2: lead-lag filter-2	X4	X6	*0.061*	*0.468*

Table 8.2: Variables and constants for filters F1 and F2

8.2.3 Complete Specification

The individual blocks of Figure 8.1 have been specified in the previous sections, the complete system is a combination of these blocks and is defined by the composition operation of hybrid automata. The code for the composed automaton `LAC` is shown in Figure 8.7. The composition operation identifies the common variables in the different components and synchronizes them.

```
hybridautomaton LAC
  compose
    deadband: G1; limiter: G2;     % components
    sags: G3; lags: G4; dsnctzr: G5;
    llf1: F1; llf2: F2
```

Figure 8.7: HIOA code for the complete LAC system

8.3 Observations from LAC Specification

The Longitudinal Axis Controller of an aircraft is a typical multi-mode control system with continuous input-output and discrete switches. It is a *real-life* hybrid system which is represented in the form of a functional block diagram. The system consists of two kinds of basic blocks: lead-lag filters and nonlinearities. We have developed generic specifications for these two kinds of blocks, created specific instances of these generic blocks and have composed the instances to get the complete specification of the system. The following observations are made from this study, regarding the application of HIOA.

- Converting a block diagram representation of a system to a HIOA specification is straightforward and involves very little learning apart from the syntax of the specification language. The composition operation can be effectively used to put together the specification of the individual blocks.

- In block diagram representation of common control systems, certain kinds of blocks occur frequently (for example the lead-lag filter and the nonlinearities in this case study). Hence, a library of HIOA template specifications for the frequently occurring blocks could be provided as part of the specification environment. Such templates would have to be *instantiated* with the proper characteristic constants and the proper variables prior to their usage. We have indirectly used this technique by first developing the generic specifications and then tabulating the specific variables and the characterizing constants.

- Typical hybrid systems (like the LAC system) have linear ODEs describing the continuous behavior of the system. Linear ODEs can be expressed in HIOA along with algebraic definitions. For reasoning with the exact trajectories of the automaton, these differential equations are to be solved. Hence we need to combine a numerical tool with HIOA for solving ODEs.

8.4 Simple Deceleration

In this section we discuss the specification of the **simple deceleration** problem which consists of a vehicle moving on a track and a controller retarding the vehicle so as to keep the final velocity of the vehicle within a given range. The vehicle has two *modes* of operation: (1) in the *braking mode* the vehicle decelerates nondeterministically, and (2) in the *non-braking mode*, the vehicle continues to move with a constant velocity. The non-deterministic deceleration is bounded within a given range.Two simplifying assumptions about this model are :

- No delay: The controller can instantaneously affect a change in the mode of operation of the vehicle.
- No feedback: The controller receives no information from the vehicle.

The MMT-specification of the simple deceleration problem has been given in [45], based on which we need to specify the system using HIOA. The MMT-model [11], named after Merritt, Modugno and Tuttle, is a convenient notation to describe a subclass of hybrid input/output automata. In the remaining part of this section, we develop the HIOA specification of the vehicle and the controller as two interacting hybrid automata, we compose them to describe the entire system, the composed automaton is then transformed to an equivalent primitive automaton, and finally we develop the PVS specification of the entire system.

8.4.1 Problem Description

The simple deceleration problem defined above involves the following constants: the initial position(si), the initial velocity(vi), the final position(sf), the target final velocity range $[vfmin, vfmax]$, the strongest($amin$) and the weakest($amax$) decelerations produced by the application of brakes. The following constraints are imposed on these constants in order to ensure that the deceleration maneuver is actually possible:

1. $si < sf$ — initial position is before final position.

2. $vi > vfmax \geq vfmin > 0$ — initial velocity is more than the target range.

3. $ai = 0$ — initially the vehicle is moving at a constant velocity.

4. $amin \leq amax < 0$ — $amin$ is the strongest possible deceleration

5. $sf - si \geq \frac{vfmax^2 - vi^2}{2amax}$ — the weakest deceleration can bring the vehicle to the highest allowed velocity in the given distance.

6. $\frac{vfmax-vi}{amax} \leq \frac{vfmin-vi}{amin}$ — the least amount of time for which the controller must brake is less than the greatest amount of time for which the controller can brake.

In the next two sections we define the hybrid automata for the vehicle and the controller.

8.4.2 The `VEHICLE` Automaton

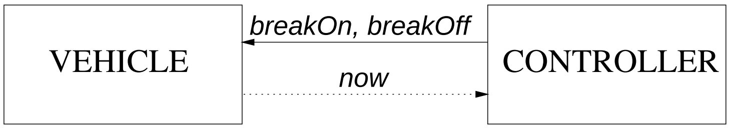

Figure 8.8: Vehicle and Controller

The vehicle is modeled by a hybrid automaton called `VEHICLE`(Figure 8.9). The acceleration, and hence the entire dynamics of the vehicle is guided by the controller, which communicates with the vehicle via external actions *brakeOn* and *brakeOff*. These actions are input actions to the `VEHICLE` automaton.

The state of the vehicle is defined by its internal variables position(s), velocity(v) and deceleration(a). These variables are declared as real analog variables and are initialized to their starting values. The fourth state variable b indicates the *operating mode* of the automaton. The output variable *now* records the progress of real time in the entire system.

The actions *brakeOn* and *brakeOff* are input actions, so they do not have preconditions. The *breakOn* action sets the braking mode, it makes $b =$ `true` and nondeterministically

```
hybridautomaton VEHICLES(S0,V0,Amin,Amax:Real)
  signatures
    input brakeOn, brakeOff
  variables
    internal analog s : Real := S0,
    internal analog v : Real := V0,
    internal analog a : Real := 0.0,
    internal b : Bool := false,
    output analog now: Real := 0.0
  transitions
    brakeOn
      eff b := true;   a := choose [Amax, Amin]
    brakeOff
      eff b := false;   a := 0.0
  trajectories
    activity always
      precon true  evolve s‘ := v; v‘ := a; now‘ := 1.0
    activity braking
      precon b = true  evolve a := choose [Amax, Amin]
    activity non_braking
      precon b = false  evolve a := 0.0
```

Figure 8.9: HIOA code for the VEHICLE automaton

sets a within the range $[Amax, Amin]$. *breakOff* sets the non-braking mode by resetting b and making $a =$ zero.

In the **trajectories** section, the activity *always* defines the rates of s, v and *now*. The other two activities, namely *braking* and *non_braking*, set the value of a depending on the operating mode (b).

8.4.3 The ONE_SHOT Controller Automaton

A simple *brake-controller* called ONE_SHOT has been described in [45] which achieves the deceleration maneuver by applying the brakes only once. This means that the *brakeOn* and *brakeOff* actions can occur exactly once. The HIOA specification of this controller is given in Figure 8.10. The `ONE_SHOT` automaton executes in three phases *idle*, *braking* and

```
hybridautomaton ONE_SHOT(A,B,C:Real )
  type Phases = enumeration of idle, braking, done
  signatures
    output brakeOn, brakeOff
  variables
    internal phase : Phases := idle,
    internal brake_time : Real ,
    input analog now: Real
  transitions
    brakeOn
      pre phase = idle ∧ now ≥ 0 ∧ now ≤ A
      eff phase := braking; brake_time := now
    brakeOff
      pre phase = braking ∧
        now−brake_time ≥ B ∧ now−brake_time ≤ C
      eff phase := done
```

Figure 8.10: HIOA code for the `ONE_SHOT` controller automaton

done as limited by the three parameters A,B and C. The input variable *now* is a measure of real-time and it synchronizes the executions of `ONE_SHOT` with the `VEHICLE` automaton. The `ONE_SHOT` automaton waits in the *idle* phase for a duration of 0 to A as measured by *now*, after which the *brakeOn* action occurs. The occurrence of this action has two effects: firstly, it brings the automaton to the *braking* phase and the time of occurrence of the action is recorded in an internal variable called *brake_time*. The automaton can continue in the *braking* phase for a duration of B to C units as measured by *now*, beyond which the *brakeOff* action occurs, bringing the automaton to its final *done* phase. It is to be noted that the `ONE_SHOT` automaton has only one continuous variable(*now*), which is not a locally controlled variable, and hence the automaton does not have any activities.

In [45] a *correct* controller has been defined as an automaton which when composed with `VEHICLE`, yields a HIOA whose hybrid traces satisfy the following:

> **Safety property**: In all states of the hybrid traces : $s = sf \Rightarrow vfmin \leq v \leq vfmax$
>
> **Timeliness property**: For every hybrid trace : $\exists\, t \geq \Re^{\geq 0}$ such that, if $now = t$ for some state, then there is a state of the trace at which $s = sf$.

Also, it has been proved that if the parameters A,B and C are chosen as :

$$A = \frac{1}{vi}\left(sf - si - \frac{vfmax^2 - vi^2}{2amax}\right)\ ;\ B = \frac{vfmax - vi}{amax}\ ;\ C = \frac{vfmin - vi}{amin} \quad ...(8.1)$$

then the `ONE_SHOT` controller *is* correct. In the next section we define the complete system as the composition of the `VEHICLE` automaton and a *correctly* instantiated `ONE_SHOT` automaton.

8.4.4 The Composed `SYSTEM`

The `SYSTEM` automaton defines a composition of the `VEHICLE` automaton instantiated by the constants mentioned in Section 8.4.1 and the `ONE_SHOT` automaton instantiated according to values given in equation (8.1).

```
hybridautomaton SYSTEM(si, vi, amin,amax, sf, vfmin, vfmax :Real)
  compose
    v : VEHICLE( si, vi, ai, amin, amax);
    c : ONE_SHOT(((sf - si -(vfmax*vfmax - vi*vi)/(2*amax))/vi),
        (vfmax - vi)/amax,(vfmin - vi)/amin)
```

This composite automaton is transformed into an equivalent primitive automaton called `System_Primitive`(Figure 8.11) by the composition transformation(see section 5.4).

A few points are to be noted regarding the `System_Primitive` automaton. First of all the actions *brakeOn* and *brakeOff* are output actions with preconditions corresponding to the

```
hybridautomaton System_Primitive( si, vi, amin, amax, vfmax, vfmin, sf: Real )
  type Phases = enumeration of idle , braking , done
  signatures
    output brakeOn , brakeOff
  variables
    internal analog s : Real := si , internal analog v : Real := vi ,
    internal analog a : Real :=0.0, internal b : Bool := false ,
    output analog now : Real :=0.0, internal phase : Phases:= idle ,
    internal brake_time : Real
  transitions
    brakeOn
      pre phase = idle ∧ (now ≥ 0.0)
        ∧ (now ≤ ((sf-si-(((vfmax*vfmax)-(vi*vi))/(2*amax)))/ vi ))
      eff phase := braking ; b := true ;brake_time := now;
            a := choose [ amax , amin ]
    brakeOff
      pre phase = braking ∧ (( now - brake_time ) ≥ (( vfmax - vi )/ amax ))
        ∧ (( now - brake_time ) ≤ (( vfmin - vi )/ amin ))
      eff phase := done ; b := false ; a :=0.0
  trajectories
    activity braking
      precon b   evolve now':=1.0; s':= v ; v':= a ; a := choose [ amax , amin ]
    activity non_braking
      precon ¬b   evolve now':=1.0; s':= v ; v':= a ; a:=0.0
```

Figure 8.11: HIOA code for `System_Primitive` automaton

actions of `ONE_SHOT` with the formal parameters A, B and C replaced by the actual parameters used for instantiating `ONE_SHOT` in the definition of `SYSTEM`. Secondly, the effects of the actions are obtained by concatenating the hybrid programs of the corresponding actions of `ONE_SHOT` and `VEHICLE`. Third, as the `ONE_SHOT` automaton does not have any locally controlled continuous variable, the activities of `System_Primitive` are same as the simplified activities of `VEHICLE`.

8.5 PVS Specification of Simple Deceleration

Now we develop the PVS theories specifying the simple deceleration problem from the HIOA specification of the `System_primitive` automaton, using the scheme presented in Chapter 7. First, we present important portions of the auxiliary theories which are required for defining the elements of the state vector of the automaton and then in Section 8.5.2 the instance of the template theory is discussed. The complete theories specifying the simple deceleration problem can found be in the following dump file:

`http://kohinoor.csa.iisc.ernet.in/~mitras/main/research/hybrid-theory.dmp`

8.5.1 Auxiliary Theories

If we look at the continuous (**analog**) variables of `System_Primitive`, there are four physical quantities : time, distance, velocity and acceleration. Definition of the data type for time and the basic operations on it have already been developed in Section 7.4. In an identical manner the data types and the operators for displacement and velocity are defined, as shown in Figure 8.12 and Figure 8.13 respectively.

```
disp: datatype
 begin
  disp(dis: {r: real | true}): disp?
  dispinf: dispinf?
 end disp

Disp: theory
 begin
  importing disp
  ≤(s1, s2: disp): bool = if disp?(s1) ∧ disp?(s2) then dis(s1) ≤ dis(s2)
    else dispinf?(s2) endif;
  ...
end Disp
```

Figure 8.12: Data type and theory for displacement

In the *Velo* theory, two additional operators are defined for multiplying a *velo* with a real constant and for dividing a *velo* with a non-zero real constant.

```
velo: datatype
 begin
  velo(vel: {r: real | true}): velo?
  veloinf: veloinf?
 end velo

Velo: theory
 begin
  importing velo
  ≤(v1, v2: velo): bool = if velo?(v1) ∧ velo?(v2)then vel(v1) ≤ vel(v2)
    else veloinf?(v2) endif;
  ...
  c : real × v : velo: velo =if velo?(v) then velo(c × vel(v))
    else veloinf endif;
  v : velo/c : real: velo =if c ≠ 0 then velo(vel(v)/c)
    else veloinf endif
 end Velo
```

Figure 8.13: Data type and theory for velocity

Acceleration is defined by a parameterized data type called *acce*, which is a union of a finite segment of the negative real axis [*min, max*] and *noacc*; *noacc* corresponds to 0 acceleration, and the segment corresponds to the range of deceleration produced by braking i.e., [*amin, amax*]. The *Minmax* axiom stated later in the *physics* theory, ensures that the first parameter defining the *acce* data type is more negative than the second.

```
acce[min, max: real]: datatype
 begin
  acce(acc: {r: real | min ≤ r ∧ r ≤ max}): acce?
  noacce: noacce?
 end acce

Acce[min, max: real]: theory
 begin
  importing acce[min, max]
  ≤(s1, s2: acce): bool = if acce?(s1) ∧ acce?(s2) then acc(s1) ≤ acc(s2)
    elsif noacce?(s1) ∧ acce?(s2) then false else trueendif;
  ...
 end Acce
```

Figure 8.14: Data type and theory for acceleration

In the *physics* theory(Figure 8.15), we define the constants of the simple deceleration problem and formally state the assumptions made in Section 8.4.1. The first part of the theory imports the necessary theories with appropriate parameters. Next, the constants defining the problem are declared; note that the symbol *NDA* represents a constant of type *acce*, which can nondeterministically take any value in the range [*min*, *max*]. The

```
physics: theory
 begin
  min, max: real
  Minmax: axiom min ≤ max ∧ max < 0
  importing Disp, Velo, Acce[min, max], Time

  vi, vf, vfmax, vfmin:  (velo?);     si, sf: (disp?)
  a_i: acce = noacce;    NDA: (acce?);
  amax: acce =   acce(max);    amin: acce = acce(min)


  v1 : velo/a1 : acce:time =if velo?(v1) ∧ (acce?(a1)) then fintime(vel(v1)/acc(a1))
    else infinity endif;
  s : disp/v : velo:time=if disp?(s) ∧ velo?(v) ∧ (vel(v) ≠ 0) then fin-
time(dis(s)/vel(v))
    elsif dispinf?(s) ∧ veloinf?(v) then fintime(1)
    elsif disp?(s) ∧ veloinf?(v) then fintime(0) else infinity endif endif;
  v : velo × t :  time:disp=if velo?(v) ∧ fintime?(t) then disp(vel(v) × dur(t))
    else dispinf endif;
  a : acce × t : time:velo = if (noacce?(a)) then velo(0)
    elsif fintime?(t) then velo(acc(a) × dur(t))  else veloinf endif;

  Forward: axiom si < sf
  Slowdown: axiom vi > vfmax ∧ vfmax > vfmin ∧ vfmin > velo(0)
  WeakestBraking: axiom  (sf − si) ≥ (vfmax − vi) × (((vfmax − vi)/2)/amax)
  LeastTime: axiom ((vfmax − vi)/amax) ≤ ((vfmin − vi)/amin)

  A: (fintime?) = (sf − si − ((vfmax − vi) × (((vfmax − vi)/2)/amax)))/vi
  B: (fintime?) = (vfmax − vi)/amax
  C: (fintime?) = (vfmin − vi)/amin
 end physics
```

Figure 8.15: Theory *physics*

third section declares several operators involving more than one data type defined in the auxiliary theories. For instance, the second declaration defines the result of dividing *disp* by *velo*, to be *fintime* if the denominator is non-zero. The next section states the restrictions on the constants as formal axioms. The assumptions 1, 2, 4, 5, and 6 of Section 8.4.1 correspond to the axioms *Forward*, *Slowdown*, *Minmax*, *WeakestBraking* and *LeastTime* respectively. Assumption number 3 is included as an initial condition of the automaton. The final section of the *physics* theory defines the constants *A*, *B*, *C* which are used later for describing the activities of the automaton.

8.5.2 The `System_Primitive` Theory

The necessary data types and constants for defining the automaton's components have been declared in the auxiliary theories. In this section we present the instantiated parts of the template theory given in Section 7.4.1 which specifies the jumps and flows of the `System_Primitive` automaton.

The *stateVector* is a record with the variables of the `Sys_prim` automaton and the *vectors* type adds the time component *now* to *stateVector*. The *validState?* predicate does not enforce any constraining condition on the allowed valuations of the variables.

```
stateVector: type = [# ph:Phases,dis:disp,vel:velo,acc:acce,b:boolean #]
vectors: type = [# basic:stateVector, now:time #]
validState?(s: vectors): bool = true
```

The *actions* data type defines the two actions *brakeOn* and *brakeOff*. These are simple actions so their constructors have only one parameter i.e., the initial valuation *fstate*. The *enabled_specific* and *eff* functions define the preconditions and the effects of the actions exactly as in the `System_primitive` automaton. The **with** keyword is used in PVS to assign to selected components of a record. Two additional functions *velRange* and *disRange* are defined for writing the evolutions of the continuous variables. *NDA* has been defined in an auxiliary theory as a constant acceleration within the range $[amin, amax]$.

```
actions: datatype
 begin
  brakeOn(fstate: vectors): brakeOn?
  brakeOff(fstate: vectors): brakeOff?
 end actions
velRange(v: velo, t: time): velo = v + (NDA × t)
disRange(d: disp, v: velo, t: time): disp =  d + (v × t) + (1/2) × (NDA × t) × t

enabled_specific(a: actions, s: vectors): bool =
 cases a of
  brakeOn(v): ph(basic(s)) = idle ∧ now(s) ≥ 0.0 ∧ now(s) ≤ A,
  brakeOff(v): ph(basic(s)) = brake ∧ now(s) ≥ B ∧ now(s) ≤ C
```

```
 endcases

eff(a: actions, s: vectors): vectors =
 cases a of
  brakeOn(v): s with [now:=now(s),basic:=basic(s) WITH
              [(ph):=brake, (b):=true, (acc):=NDA]],
  brakeOff(v): s with [now := now(s), basic := basic(s) with
              [(ph) := done, (b) := false, (acc) := A_0]]
 endcases;
```

The *flows* data type defines the flows corresponding to the two activities *braking* and *nonbraking*. The *sameActivity?* function checks if two given flows are from the same activity. *operating?* and *evolutions* functions define the activation conditions and the progress functions of the individual activities.

```
flows: datatype
 begin
  braking(fstate: vectors, period: time): braking?
  nonbraking(fstate: vectors, period: time): nonbraking?
 end flows
sameActivity?(f, f_1: flows): bool =
 cases f of
  braking(v, preiod): if  braking?(f_1) then true else false endif,
  nonbraking(v, preiod): if nonbraking?(f_1) then true else false endif
 endcases

operating?(f: flows, s: vectors): bool = validState?(s) ∧
 cases f of
  braking(v, preiod): v = s ∧ b(basic(s)),
  nonbraking(v, period): v = s ∧ ¬ b(basic(s))
 endcases

evolve(f: flows): [interval(ftime(f), ltime(f)) → vectors] =
 cases f of
  braking(v, period): (λ (t: interval(ftime(f), ltime(f))):
  (# now := now(v) + t, basic := (# (ph) :=  ph(basic(v)),
   (dis) := disRange(dis(basic(v)), vel(basic(v)), t),
   (vel) := velRange(vel(basic(v)),t),
   (acc) := acc(basic(v)), (b) := b(basic(v)) #) #)),
  nonbraking(v, period):  (λ (t: interval(ftime(f), ltime(f))):
  (# now := now(v) + t,   basic := (# (ph) := ph(basic(v)),
   (dis) :=  dis(basic(v)) + (vel(basic(v)) × t),  (vel) := vel(basic(v)),
   (acc) := acc(basic(v)), (b) :=  b(basic(v)) #) #))
 endcases;
```

The initial valuation of the variables is defined as a *vector* called *init* and this is used to define the set of initial valuations *start*.

```
init: vectors =  (# basic := (#ph := idle, dis := s_i,vel := v_i,
            acc := 0.0, b:= false #), now := zero #)
start(s: vectors): boolean = (s = init)
```

The main components of the automaton, in the above theory, are obtained by directly substituting the definitions from the automaton `System_primitive`(Figure 8.11) into the template theory with certain secondary declarations like *velRange*, *disRange* and *init*.

8.5.3 Specifying Properties of `System_primitive` in PVS

In the previous section we have developed the *System_primitive* theory which defines a hybrid I/O automaton for the simple deceleration problem. Now we want to state the safety and the timeliness properties of the system as lemmas in PVS, so that one can derive the properties from the specification using the PVS prover. The properties of hybrid I/O automata are stated in terms of admissible executions of the automaton. In Chapter 7 we had developed the theory *execution_element*(Section 7.4.2) which defines the execution elements of an automaton in terms of its actions and activities, and we also developed the *executions* theory (Appendix C) which defines the admissible executions of an automaton from its execution elements. Using these two theories and the basic definition of the automaton given in the *System_primitive* theory, we can state the properties of the system in terms of executions.

The safety condition written in the *Safety* lemma states that in all reachable states of all the admissible executions of the automaton the *SafeState* invariant holds. This invariant asserts that, if the vehicle has reached its final destination (*sf*) then its velocity must be within the desired range [*vfmin*, *vfmax*]. The *Timeliness* lemma states the timeliness condition of Section 8.4.3, that is, there exists a time constant *tc*, such that in every admissible execution of the automaton, if there is a state with *now* = *tc*, then there exists a state in the same execution with *dis* = *sf*.

$$
\begin{array}{l}
\textit{SafeState}\texttt{:}\ \textit{invariant}\ =\ \ \lambda(v\texttt{:}\textit{vectors})\texttt{:} \\
\qquad v\,\textit{`basic`dis}\ =\ \textit{sf}\ \supset\ (v\,\textit{`basic`vel}\ \geq\ \textit{vfmin}\ \wedge\ v\,\textit{`basic`vel}\ \leq\ \textit{vfmax}) \\
\textit{Safety}\texttt{:}\ \ \mathbf{lemma}\ \forall\ (v\texttt{:}\textit{vectors}\,|\,\exists\ (c\texttt{:}\ (\textit{admissExec?})\texttt{)}\texttt{:} \\
\qquad\quad \textit{c\_reachable}(v,\ c))\texttt{:}\ \textit{SafeState}(v) \\
\\
\textit{Timeliness}\texttt{:}\ \mathbf{lemma}\ \ \exists\ (tc\texttt{:}\ (\textit{fintime?}))\texttt{:}\qquad \forall\ (c\texttt{:}\ (\textit{admissExec?}))\texttt{:} \\
\qquad\quad (\exists\ (v_1\texttt{:}\ \textit{vectors})\texttt{:}\ \textit{c\_reachable}(v_1,\ c)\ \wedge\ v_1\,\textit{`now}\ =\ tc)\ \supset \\
\qquad\qquad (\exists\ (v_2\texttt{:}\ \textit{vectors})\texttt{:}\ \textit{c\_reachable}(v_2,\ c)\ \wedge\ v_2\,\textit{`basic`dis}\ =\ \textit{sf})
\end{array}
$$

8.6 Observations from Simple Deceleration

The problem discussed in this study has been taken from [45]; it is the specification of a system consisting of a vehicle and a controller, where the objective of the controller is to retard the vehicle to a certain predefined velocity range, within a given distance, by applying the brake only once. A one shot controller for this system has been given in [45] and also its correctness has been proved. We specify this system as a composition of two hybrid automata, the `VEHICLE` and the `CONTROLLER`. The equivalent primitive form of the composed automaton is derived using the rules given in Section 5.4, and finally we develop the PVS specification of the entire system by instantiating the template theory of Figure 7.8 and state the properties of the system in terms of hybrid executions. The key observations from this study are:

- MMT-specifications of hybrid systems are translated into HIOA specifications by interpreting the tasks of the former as activities.

- With the scheme proposed in Chapter 7, the process of generating PVS theories from HIOA specifications can be partially automated. Instantiation of the template theory from components of HIOA specification can be done automatically, but the definition of the necessary data types and operators, as well as assertion of properties, still involves some manual work.

- Parameterized automata are often defined with restrictions on the values of the parameters. It would be useful to provide mechanisms in the HIOA language, similar to the **so that** clause of the IOA language[26], for enforcing constraining conditions on the formal parameters of an automaton. These conditions can then be directly translated to define suitable data types or axioms.

Chapter 9

Conclusions and Future Research

9.1 Concluding Remarks

The work presented in this dissertation is motivated by the belief that integration of tools for specification and verification of hybrid systems is essential for deriving practical benefits from formal methods. Our effort has been in developing a specification language for hybrid systems and implementing the supporting tools, which we hope would be a first step towards integration of methods.

The main contribution of this work is the design and implementation of HIOA - a specification language for hybrid systems based on the hybrid I/O automaton model. The effectiveness of a specification language is gauged by its expressive power and learnability. A specification language for hybrid systems in particular has to provide mechanisms for describing the continuous behavior of the system. In HIOA, we have introduced the notion of activities for describing the continuous evolution of the variables of the system. Activities are syntactically similar to actions, but unlike the *locations* of hybrid automata [1], more than one of them can operate simultaneously. This gives the user flexibility in developing the model of the system. The applicability of HIOA in specifying hybrid systems has been evaluated with several case studies. In this dissertation we have presented the specification of the longitudinal axis controller of a combat aircraft given in the form of a functional block diagram and a vehicle deceleration system given as a MMT-specification[11]. We believe that HIOA allows a wide range of hybrid systems to be specified intuitively without involving much learning effort.

As part of the language front end, we have written a parser which creates an Abstract Syntax Tree(AST) representation of the input specification. The AST is used for semantic checking and for generating an intermediate description of the system to be used by the other analyzing tools.

Typically complex hybrid systems are described by the interaction(composition) of several component automata, whereas most of the analytical methods are applicable to primitive automata. We have designed a composer for HIOA components, which transforms a composite automaton description into an equivalent primitive form.

We have proposed a *workbench* for analyzing tools for hybrid systems centered around HIOA. This language centric design reduces the translation efforts between different languages and tools, and as the different tools are decoupled, they can be developed independently. Establishing the basic structure of the specification language, we hope, would pave the way towards integration of tools and interfacing them with HIOA.

Deductive techniques like invariant assertions and induction over length of execution, are used to prove properties of hybrid automata and these methods can be effectively applied using mechanical provers like PVS. We have developed an interface for converting the HIOA specifications to PVS theories which will enable users to employ the powerful PVS prover, without having to manually write the PVS specification. This translation involves the instantiation of a predefined template theory, which we believe, can be done automatically. The underlying theories for inductive reasoning have also been developed.

9.2 Directions for Future Research

Just like any other language, HIOA will also have to go through a development cycle. We have presented the initial prototype of HIOA after studying the various requirements. The language specification can be improved after reviewing the requirements further and with some more case studies. The implementation of the front end which supports the language could also be improved and optimized. In this section we present, both the immediate implementation tasks and the long term development projects, which are to be undertaken for making the workbench complete.

The composer presented in Section 5.4 has to be implemented and integrated with the front end. Also, the method developed for automatic simplification of a complex set of activities(Section 3.3.3) could to be written and integrated within the front end. This procedure should be invoked during the semantic analysis of the input specification, when a particular back-end tool requires the activities of the automaton to be represented in the *simple* form.

The scheme for generating PVS theories from HIOA specifications presented in Chapter 7 has to be incorporated into a customized user interface for PVS. As a part of this interface, specialized PVS proof strategies could be developed for automatically generating the proof goals for commonly used proof techniques like induction over length of execution.

In the study of the longitudinal axis controller in Chapter 8 we found that certain blocks occur repetitively in the specification of the system. In such cases a library of automaton specification templates can perhaps help the users to develop system specifications quickly by instantiating the templates.

Apart from adding to the HIOA framework, interfaces to existing tools could be developed. For instance, a translator to HyTech[16] could be built for automatic model checking of HIOA specifications. HyTech is a tool for symbolic model checking of hybrid automata [1] and it has been shown[7] that the hybrid automata model is a subclass of the hybrid I/O automata model. So HyTech can be used for model checking a restricted class of HIOA specifications by translating them to HyTech code. This translation should be possible from the abstract syntax tree representation produced by the HIOA parser.

Appendix A

HIOA Grammar

In this appendix, we summarize the grammar and lexical conventions of HIOA. The keywords and the complete BNF grammar of HIOA are presented under annotated sections. This appendix is meant to aid understanding of the syntax and is not complete in itself since the semantic rules are not specified alongside the grammar.

Uppercase words and symbols enclosed in single quotation marks are terminal symbols in the BNF grammar. Keywords of HIOA, written in boldface, are also terminal symbols. All other words are nonterminal symbols.

A.1 Keywords

type , **enumeration of** , **tuple** , **union** — type declaration and definition

hybridautomaton — declaration of hybrid automaton

signatures — declaration of automaton action signature

input , **output** , **internal** — variable and action types

analog — variable dtype

where — constraint clause on action parameters

choose — nondeterministic assignment

transitions , **pre** , **eff** — definition of automaton transitions

trajectories , **activity** , **precon** , **postcon** , **evolve** — definition of trajectories

compose — automata composition

if , **then** , **elseif** , **else** , **fi** — conditional assignment

invariant , **of** , **simulation** , **from** , **to** , **forward** , **backward** — automaton assertions

A.2 Lexical Syntax

Apart from keywords, the lexical grammar of HIOA uses the following symbols :

- Punctuation marks: `, : ; ( ) { } [ ] __ :=`
- Comment character: %
- Identifiers: sequences of letters, digits, apostrophes, and underscores. The LaTeX identifiers for the Greek letters can also be used as identifiers.
- Operators: sequences of `- ! # $ & * + . < = > ? @ | ~ / ^` or backslash followed by one of these characters, characters `_ %`, or by an identifier(other than a name of a Greek letter). `\A` and `\E` are used as universal and existential quantifiers respectively.

A.3 BNF Grammar

Syntax of automaton definition

```
hioaSpec          ::= (typedef | hyAutomatonDef | assertion)+
hyAutomatonDef    ::= hybridautomaton automatonName automatonFormals?
                      typeInfo* (simpleBody | composition )
automatonName     ::= IDENTIFIER
automatonFormals  ::= '(' automatonFormal, + ')'
automatonFormal   ::= IDENTIFIER,+ ':'  type
typeInfo          ::= typedef | assumes traitRef,+ % see comment 1
```

Syntax of type declarations

```
type             ::= simpleType | compoundType
simpleType       ::= IDENTIFIER
compoundType     ::= IDENTIFIER '[' type,+ ']'
typeDef          ::= type simpleType '=' shorthand
shorthand        ::= enumeration of IDENTIFIER,+
                   | (tuple | union ) of (IDENTIFIER,+ ':'  type),+
```

Syntax of primitive automaton definition

```
simpleBody       ::= signatures formalActionList+ variables
                     transitions trajectories
formalActionList ::= actionType formalAction,+
actionType       ::= input | output | internal
formalAction     ::= actionName (actionFormals where?)?
actionName       ::= IDENTIFIER
actionFormals    ::= '(' actionFormal,+ ')'
actionFormal     ::= IDENTIFIER,+ ':'  type
where            ::= where predicate
```

Syntax of automaton variables

```
variables        ::= variables variable,+
variable         ::= variableType variableName ':'  type (':=' value)?
variableType     ::= input | output | internal
variableName     ::= IDENTIFIER
value            ::= term | choice
```

Syntax of nondeterministic choice

```
choice           ::= choose (openSym interval closeSym);+
```

```
openSym         ::= '[' | '('
closeSym        ::= ']' | ')'
interval        ::= term ',' term % see comment 2
```

Syntax of transitions

```
transitions     ::= transitions transition+
transition      ::= actionHead preconditions?  effect?
actionHead      ::= actionName (actionActuals where?)?
actionActuals   ::= '(' term,+ ')'
precondition    ::= pre predicate
effect          ::= eff program
```

Syntax of trajectories

```
trajectories    ::= trajectories trajectory+
trajectory      ::= activity activityName activityBody
activityName    ::= IDENTIFIER
activityBody    ::= activationCond?  evolve
activationCond  ::= (precon | postcon ) predicate
evolve          ::= evolve program
```

Syntax of hybrid programs

```
program         ::= statement;+
statement       ::= assignment | conditional
assignment      ::= lvalue ':=' value
lvalue          ::= variable
                    | lvalue '[' term,+ ']'
                    | lvalue '.'  IDENTIFIER
                    | lvalue '''
conditional     ::= if predicate then program
```

```
                    (elseif predicate then program)*
                    (else program)? fi
```

Syntax of terms and predicates

```
predicate          ::= term
term               ::= if term then term else term
                     | subtern
subterm            ::= subterm (opSym subterm)+
                     | (qunatifier | opSym)* opSym secondary
                     | (qunatifier | opSym)* quantifier primary
                     | secondary opSym*
opSym              ::= OPERATOR | '<=>' | '=>' | '/\' | '\/' | '=' | '~=' | '.'
quantifier         ::= ('\A' | '\E') variable
variable           ::= IDENTIFIER (':' type)?
secondary          ::= primary
                     | primary? bracketed ('.'? primary)?
primary            ::= primaryHead (':' type | '.' primaryHead)*
primaryHead        ::= REAL
                     | NUMERAL
                     | IDENTIFIER ('(' term,+ ')')?
                     | '(' term ')'
bracketed          ::= openSym term,* closeSym (':' type)?
```

Syntax of composite automata definitions

```
composition        ::= compose component;+
component          ::= handle ':' automatonName(automatonActuals)?
handle             ::= IDENTIFIER
automatonActuals   ::= '(' automatonActual,+ ')'
automatonActual    ::= term
```

Syntax of assertions

`asserion`	`::=`	`invariant` | `simulation`
`invariant`	`::=`	**invariant of** `automatonName ':' term`
`simulation`	`::=`	`simkind` **simulation from** `automatonName`
		to `automatonName ':' predicate`
`simkind`	`::=`	**forward** | **backward**

A.4 Comments

1. Syntax of `traitRef` is a part of LSL and can be found in [26].
2. Nondeterministic **choose** statements are defined for real variables, however the grammar allows terms of any type to be used as bound.

Appendix B

Intermediate Language Grammar

In this appendix we give the BNF grammar of the Intermediate Language(IL) code generated by the HIOA front end. The boldfaced words are keywords of IL. The keywords and the symbols enclosed in single quotation marks are terminal symbols, all other words are non terminal symbols.

B.1 BNF Grammar

Specification File

```
specfile        ::= '(' decls hybridAutoDef* ')'
```

Declarations

```
decls           ::= sortDecl* varDecl* opDecl*
sortDecl        ::= '(' sort sortId '"' name '"' sortId* lit?  ')'
varDecl         ::= '(' var varId '"' name '"' sortId ')'
opDecl          ::= '(' op opId '"' name '"' sortId+ ')'
name            ::= ( ~["\""] )*
sortId          ::= 's' (DIGIT)+
varId           ::= 'v' (DIGIT)+
```

`opId`	`::=`	`'o' (DIGIT)+`
`DIGIT`	`::=`	`[0-9]`
`REAL`	`::=`	`(DIGIT)+'.'(DIGIT)+`
`LETTER`	`::=`	`[A-za-z_]`
`ID`	`::=`	`( LETTER )+ (DIGIT` | `LETTER )*`

Automaton definition

`hybridAutoDef`	`::=`	`'('` **hybridautomaton** `nameAndDecls hyAutoBody ')'`
`nameAndDecls`	`::=`	`hyAutoId formals?`
`hyAutoId`	`::=`	`ID`
`formals`	`::=`	`'('` **formals** `varId+ ')'`
`hyAutoBody`	`::=`	**actions** `action+` **variables** `variable+` **transitions** `transition+` **trajectories** `trajectory+`

Action definition

`action`	`::=`	`'(' actionName actionFormals? where? ')'`
`actionName`	`::=`	`actionType actionId`
`actionType`	`::=`	**input** | **output** | **internal**
`actionId`	`::=`	`ID`
`actionFormals`	`::=`	`varId+`
`where`	`::=`	`'('` **where** `predicate ')'`

Variable definition

`variable`	`::=`	`'(' variableType variableDType varId value? ')'`
`variableDType`	`::=`	`ANALOG` | `DISCRETE`
`variableType`	`::=`	**input** | **output** | **internal**
`value`	`::=`	`term` | `choice`
`choice`	`::=`	`'('` **choose** `interval+ ')'`
`interval`	`::=`	`( '('` | `'[') term ',' term ( ']'` | `')' )`

Transition definition

```
transition      ::= '(' actionHead precondition?  effect?  ')'
actionHead      ::= actionName (actionActuals where?)?  NUMBER?
actionaActuals  ::= '(' actuals term+ ')'
precondition    ::= '(' pre predicate ')'
effect          ::= '(' eff program ')'
```

Trajectory definition

```
trajectory      ::= '(' activityHead (precon|postcon) evolve ')'
activityHead    ::= activityId '"' name '"'
precon          ::= '(' precon predicate ')'
postcon         ::= '(' postcon predicate ')'
evolve          ::= '(' evolve program ')'
```

Hybrid program definition

```
program         ::= '(' statement+ ')'
statement       ::= assignment | conditional
assignment      ::= '(' assign ( element | difElement) value ')' see comments
element         ::= varId
difElement      ::= 'd(' varId ')'
conditional     ::= '(' if ( '(' predicate program ')' )+
                    ( '(' else program ')' )?  ')'
```

Term definition

```
predicate       ::= term
term            ::=   '(' apply opId term+ ')'
                    | '(' quantifier term ')'
                    | sortId | varId | opId
```

```
                    | sortId NUMERAL
                    | real dortId REAL
quantifier     ::= ( '\A' | '\E') varId
```

B.2 Comments

1. The definition of hybrid program allows assignment to first derivative of variables only when the program in question defines the progress function of an activity. The syntax given here is more general and it allows derivatives to be assigned in action effects also.

Appendix C

PVS Theory for Executions

```
executions: theory
 begin

  importing execution_element

  seqs: library = "$~/PVS/lib/sequence$"

  importing seqs@sequences_type[execEle]

  k, n: var nat

  v: var vectors

  fragment: type =  {c: seq |
            ¬ empty?(c`dom) ∧
             (∀ (k: nat | c`dom(k) ∧ c`dom(k + 1)):
                lstate(c`map(k)) = fstate(c`map(k + 1)) ∧ flow?(c`map(k)) ⊃
                 jump?(c`map(k + 1)) ∧ jump?(c`map(k)) ⊃
                  flow?(c`map(k + 1)))}

  finiteSeq?(c: fragment): boolean = finite?(c)

  infiniteSeq?(c: fragment): boolean = infinite?(c)

  unique_terminal: lemma
    ∀ (c: (finiteSeq?)):
      ∀ (n1, n2: nat): largest?(c`dom)(n1) ∧ largest?(c`dom)(n2) ⊃ n1 = n2

  infinite_domain: lemma ∀ (c: (infiniteSeq?)): c`dom = nat

  execution: type =
  {c: fragment |
            (c`dom(0) ∧ flow?(c`map(0)) ∧ start(fstate(c`map(0)))) ∧
             (finiteSeq?(c) ⊃
               (∀ (x: nat | largest?(c`dom)(x)):
                  terminalFlow?(c`map(x))))}

  execution_starts: lemma ∀ (c: execution): smallest?(c`dom)(0)

  Exec_domain: lemma
    ∀ (c: execution): ∀ (n: nat): c`dom(n) ⊃ (∀ (k: nat | k < n): c`dom(k))
```

```
execution_start_state: lemma ∀ (c: execution): start(fstate(c`map(0)))

finiteExec?(c: execution): boolean =
    finiteSeq?(c) ∧
     (∀ (x: nat | largest?(c`dom)(x)): finiteEle?(c`map(x)))

finSeqInfExec?(c: execution): boolean =
    finiteSeq?(c) ∧
     (∀ (x: nat | largest?(c`dom)(x)): ¬ finiteEle?(c`map(x)))

infiniteExec?(c: execution): boolean = ¬ finiteExec?(c)

is_glb(z: time, c: (infiniteSeq?), k: nat): boolean =
    ltime(c`map(k)) ≤ z ∧ (∀ (n): n > k ⊃ z ≤ ltime(c`map(n)))

has_glb(z: time, c: (infiniteSeq?)): boolean = ∃ (k): is_glb(z, c, k)

infSeqAdmissExec?(c: execution): boolean =
    infiniteSeq?(c) ∧ (∀ (z: time): has_glb(z, c))

infSeqAdmis_domain: lemma ∀ (c: (infSeqAdmissExec?)): c`dom = nat

admissExec?(c: execution): boolean =
    finSeqInfExec?(c) ∨ infSeqAdmissExec?(c)

zeno?(c: execution): boolean = infiniteSeq?(c) ∧ ¬ admissExec?(c)

admiss_domain: lemma ∀ (c: (infSeqAdmissExec?)): c`dom = nat

initstate: lemma ∀ (c: execution): start(fstate(c`map(0)))

invariant: type = [vectors → boolean]

I: invariant

n_reachable(v: vectors, c: (infSeqAdmissExec?), n: nat):  inductive bool =
if n = 0
  then touched(c`map(0), v)
else n_reachable(v, c, n − 1) ∨ touched(c`map(n), v)
endif

f_reachable(v: vectors, c: (finSeqInfExec?), n: nat | c`dom(n)):  inductive bool =
if n = 0
  then touched(c`map(0), v)
else f_reachable(v, c, n − 1) ∨ touched(c`map(n), v)
endif

c_reachable(v: vectors, c: (admissExec?)): boolean =
    if infSeqAdmissExec?(c)
      then ∃ (n): n_reachable(v, c, n)
    else ∃ (n: nat | c`dom(n)): f_reachable(v, c, n)
    endif

satisfies(I: invariant, c: (infSeqAdmissExec?))(k): boolean =
    ∀ (v): n_reachable(v, c, k) ⊃ I(v)

f_satisfies(I: invariant, c: (finSeqInfExec?))(k: nat | c`dom(k)): boolean =
    ∀ (v): f_reachable(v, c, k) ⊃ I(v)

base(I: invariant): boolean =
    ∀ (c: (infSeqAdmissExec?)): satisfies(I, c)(0)
```

```
f_base(I: invariant): boolean =
    ∀ (c: (finSeqInfExec?)): f_satisfies(I, c)(0)

inductstep(I: invariant): boolean =
    ∀ (c: (infSeqAdmissExec?)):
      ∀ (k: nat): satisfies(I, c)(k) ⊃ satisfies(I, c)(k + 1)

f_inductstep(I: invariant): boolean =
    ∀ (c: (finSeqInfExec?)):
      ∀ (k: nat | c`dom(k) ∧ c`dom(k + 1)):
        f_satisfies(I, c)(k) ⊃ f_satisfies(I, c)(k + 1)

induction_lemma: lemma
  ∀ (I: invariant):
    base(I) ∧ inductstep(I) ⊃
     (∀ (c: (infSeqAdmissExec?)):
        ∀ (v: vectors): c_reachable(v, c) ⊃ I(v))

f_induction_lemma: lemma
  ∀ (I: invariant):
    f_base(I) ∧ f_inductstep(I) ⊃
     (∀ (c: (finSeqInfExec?)): ∀ (v: vectors): c_reachable(v, c) ⊃ I(v))

admissible_induct_theorem: theorem
  ∀ (I: invariant):
    f_base(I) ∧ base(I) ∧ f_inductstep(I) ∧ inductstep(I) ⊃
     (∀ (c: (admissExec?)): ∀ (v: vectors): c_reachable(v, c) ⊃ I(v))

start_base_lemma: lemma
  ∀ (I: invariant):
    (∀ (ee: execEle | flow?(ee) ∧ start(fstate(ee))):
      ∀ (v: vectors | touched(ee, v)): I(v)) ⊃  (base(I) ∧ f_base(I))

inf_induction_jump_lemma: lemma
  ∀ (I: invariant):
    (∀ (c: (infSeqAdmissExec?)):
       ∀ (k: nat):
         ((∀ (v): touched(c`map(k), v) ⊃ I(v)) ⊃
           (∀ (v): touched(c`map(k + 1), v) ⊃ I(v)))) ⊃ inductstep(I)
end executions
```

The complete proofs of *induction_lemma* and *f_induction_lemma* can be found in the following PVS dump file:

`http://kohinoor.csa.iisc.ernet.in/~mitras/main/research/hybrid-theory.dmp`

References

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

[2] L. E. Moser and P. M. Melliar-Smith. Formal verification of safety-critical systems. *Software, Practice and Experience*, 20(8):799–822, 1990.

[3] J. P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium (SESS'93), Brighton, UK*, pages 168–177. IEEE Computer Society Press, 30 – 3 1993.

[4] Edmund M. Clarke, Jeannette M. Wing, et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[7] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In T. Henzinger R. Alur and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, October 1995. Springer-Verlag.

[8] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata revisited.

[9] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part II: timing-based systems. In *145*, page 36. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1993.

[10] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600, pages 447–484, Mook, The Netherlands, 3–7 June 1991. Springer-Verlag.

[11] Michael Merritt, Francesmary Modugno, and Mark Tuttle. Time constrained automata. In J. C. M. Baeten andJ. F. Goote, editor, *CONCUR '91: 2nd International Conference of Concurrency Theory*, volume 527, pages 408–423, 1991.

[12] M.M. Bayoumi G. Labinaz and K. Rudie. A survey of modeling and control of hybrid systems. In *Annual Reviews in Control*, volume 21, pages 79–92, 1997.

[13] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embeded systems. *IEEE Transactions in Software Engineering*, 22(3):181–201, March 1996.

[14] T. A. Henzinger and P. -H. Ho. Algorithmic analysis of nonlinear hybrid systems. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 225–238, Liege, Belgium, 1995. Springer Verlag.

[15] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *Proceedings International Symposium on Static Analysis*, volume 818, pages 223–237. Springer-Verlag, 1994.

[16] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *Coputer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–483, 1997.

[17] A. Deshpande, A. Gollu, and L. Semenzato. The shift programming language for dynamic networks fo hybrid automata, 1998.

[18] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In Oded Maler, editor, *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201, pages 171–185, Grenoble, France, 1997. Springer-Verlag.

[19] Gunter Leeb and Nancy Lynch. Proving safety properties of the steam boiler controller. In Egon Boerger Jean Raymond Abrial and Hans Langmaack, editors, *Proceedings of Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 11654 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[20] H. B. Weinberg and Nancy Lynch. Correctness of vehicle control systems – a case study. In *17th IEEE Real-Time Systems Symposium*, pages 62–72, Washington, D. C., December 1996.

[21] Connie Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time system. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1994.

[22] M. Archer, C. Heitmeyer, and S. Sims. Tame: A pvs interface to simplify proofs for automata models, 1998.

[23] Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 192–203, Brookline, MA, 1996. IEEE Computer Society.

[24] Constance Heitmeyer. On the need for practical formal methods. Technical report, Naval Research Laboratory, Code 5546, Washington, DC 20375, USA, 1998.

[25] Stephen J. Garland. The IOA Language and Toolkit.

[26] Stephen Garland, Nancy Lynch, and Mandana Vaziri. Ioa: A language for specifying, programming and validating distributed systems. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, October 1999.

[27] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.

[28] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[29] B. Liskov, N. Mathewson, and A. Myers. Polyj: Parameterized types for java, 1998.

[30] Scott Hudson. Java CUP: LALR Parser Generator for Java, 1999.

[31] A. Chefter. A simulator for the ioa language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technolog, Cambridge, MA, 1998.

[32] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[33] Gordon, M. J. C., Melham, and Thomas F. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[34] Stephen Garland and John Guttag. A guide to LP: The larch prover, 1991.

[35] Steven P. Miller and Mandayam Sriva. Formal verification of of a commercial microprocessor. Technical Report SRI-CSL-95-04, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

[36] R. S. Boyer and J S. Moore. A theorem prover for a computational logic. In M. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449, pages 1–15. Springer-Verlag, 1990.

[37] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[38] M. Devillers and D. Griffioen. A formalization of finite and infinite sequences in pvs, 1997.

[39] Nancy Lynch. Modelling and verification of automated transit systems, using timed automata, invariants and simulations. In T. Henzinger R. Alur and E. Sontag, editors, *Hybrid Systems III: Verification and Control (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 449–463, 1995.

[40] Nancy Lynch. A three-level analysis of a simple acceleration maneuver, with uncertainties. In *Proceedings of the Third AMAST Workshop on Real-Time Systems*, pages 1–22, Salt Lake City, Utah, March 1996. World Scientific Publishing Company.

[41] H. B. Weinberg, Nancy Lynch, and Norman Delisle. Verification of automated vehicle protection systems. In T. Henzinger R. Alur and E. Sontag, editors, *Hybrid Systems III: Verification and Control (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems)*, volume 1066 of *Lecture Notes in Computer Science*, pages 101–113, New Brunswick, New Jersey, October 1995. Springer-Verlag.

[42] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In *HART'97 (International Workshop on Hybrid and Real-Time System)*, volume 1201 of *Lecture Notes in Computer Science series*, Grenoble, France, March 1997. Springer Verlag.

[43] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems, 1991.

[44] M. Devillers, D. Griffioen, and O. Mueller. Possibly infinite sequences in theorem provers: A comparative study. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104, Murray Hill, NJ, August 1997. Springer-Verlag.

[45] H. B. Weinberg. Correctness of vehicle control systems – a case study. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technolog, Cambridge, February 1996.