COMPUTER SCIENCE
AND ARTIFICIAL
INTELLIGENCE
LABORATORY

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Verification Framework for Hybrid Systems

Sayan Mitra

September 2007

32 VASSAR STREET, CAMBRIDGE, MASSACHUSETTS 02139

A Verification Framework for Hybrid Systems

by

Sayan Mitra

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 31, 2007

Certified by . . . .
Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Verification Framework for Hybrid Systems
by
Sayan Mitra

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Combining discrete state transitions with differential equations, *Hybrid system* models provide an expressive formalism for describing software systems that interact with a physical environment. Automatically checking properties, such as invariance and stability, is extremely hard for general hybrid models, and therefore current research focuses on models with restricted expressive power. In this thesis we take a complementary approach by developing proof techniques that are not necessarily automatic, but are applicable to a general class of hybrid systems. Three components of this thesis, namely, (i) semantics for ordinary and probabilistic hybrid models, (ii) methods for proving invariance, stability, and abstraction, and (iii) software tools supporting (i) and (ii), are integrated within a common mathematical framework.

(i) For specifying nonprobabilistic hybrid models, we present *Structured Hybrid I/O Automata (SHIOAs)* which adds control theory-inspired structures, namely state models, to the existing Hybrid I/O Automata, thereby facilitating description of continuous behavior. We introduce a generalization of SHIOAs which allows both nondeterministic and stochastic transitions and develop the trace-based semantics for this framework.

(ii) We present two techniques for establishing lower-bounds on *average dwell time (ADT)* for SHIOA models. This provides a sufficient condition of establishing stability for SHIOAs with stable state models. A new simulation-based technique which is sound for proving ADT-equivalence of SHIOAs is proposed.

We develop notions of approximate implementation and corresponding proof techniques for Probabilistic I/O Automata. Specifically, a PIOA $\mathcal{A}$ is an ϵ-approximate implementation of $\mathcal{B}$, if every trace distribution of $\mathcal{A}$ is ϵ-close to some trace distribution of $\mathcal{B}$—closeness being measured by a metric on the space of trace distributions. We present a new class of real-valued simulation functions for proving ϵ-approximate implementations, and demonstrate their utility in quantitatively reasoning about probabilistic safety and termination.

(iii) We introduce a specification language for SHIOAs and a theorem prover interface for this language. The latter consists of a translator to typed high order logic and a set of PVS-strategies that partially automate the above verification techniques within the PVS theorem prover.

Thesis Supervisor: Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I am grateful to have had Nancy Lynch as my advisor. She shared her wisdom with me and gave me freedom to explore my interests. Her enthusiasm about my ideas, blended with her perfectionism made for a stimulating research environment. Nancy has profoundly influenced the problems I choose to work on, how I go about doing research, and the aesthetics of how I present my solutions.

It has been a pleasure to discuss my work with Sanjoy Mitter. Sanjoy shaped my thinking on probabilistic hybrid systems and encouraged me to take courses that proved to be extremely useful. I have gained from sketching my ideas to him on the blackboard and also from discussing general research directions. A significant portion of this work has benefited from bringing ideas from control theory into formal verification. This has been possible because of Daniel Liberzon's help and guidance. I am also grateful to Daniel for being a delightfully supportive colleague and for providing precise, timely and reasoned feedback.

A special thanks to Daniel Jackson for reading the thesis; your probing questions helped me improve the presentation. I would like to acknowledge Piotr Indyk, Madhu Sudan, and Ronitt Rubinfeld for being available for both academic and personal counsel, *always*; Piotr, even at unearthly hours. The theorem prover connection is based on my collaboration with Myla Archer. I thank her for sharing her knowledge, experience, and insights. I thank the members of Tempo research group: Alex Shvartsman, Radu Grosu, Scott Smolka, Laurent Michel. I would like to thank Joanne Hanley and Be Blackburn for everything that you have done to make things smooth for us—the proposals, the letters, the corn, the cookies, and yes, the moral support.

A defining feature of my MIT experience is the interaction with truly exceptional post-docs and fellow students. The company of Vineet Sinha, Tina Nolte, Seth Gilbert, Calvin Newport, Xavier Koegler, Ben Leong, Han-Pang Chiu, Dah-Yoh Lim, Vinod Vaikuntanathan, Victor Chen, Roger Khazan, Carl Livadas, and Yong Wang, has been stimulating and enjoyable. I am grateful for the support of Dilsun Kaynar in my early days, when she patiently listened to and commented on many of my half-baked ideas. My work with junior colleagues, Shinya Umeno and Hongping Lim, has been uniquely rewarding and has contributed to the thesis. Ling Cheung deserves special mention as a formidable colleague and for quietly suffering *numerous* practice talks. A special thanks to Rui Fan for many stimulating technical discussions which have influenced my thinking as a researcher, and also for introducing me to the music of Jascha Heifetz. Thanks to my dear friend David Huynh for the endless hours of fun we had together.

I have the deepest gratitude to my family for their love and kindness. The support, encouragement, and the occasional nudge (to finish) from my parents, Tapan and Mamata, has been crucial for this thesis. Your wisdom and compassion continue to astonish and inspire me. I am grateful to my sister, Shreya, for being such an unwavering champion; taking on this journey with you has been singularly rewarding. Finally, to my wife Shinjinee, for support, for keeping it colorful, for giving me time and space to think, and for being a constant source of happiness—thank you.

Sayan Mitra
31st August 2007, Cambridge, MA.

Contents

1 Introduction **10**
- 1.1 Modeling and Verification of Embedded Software . . . 10
- 1.2 Hybrid Systems . . . 11
- 1.3 Thesis Overview . . . 13
  - 1.3.1 Modeling . . . 13
  - 1.3.2 Verification . . . 15
  - 1.3.3 Software Tools . . . 16
  - 1.3.4 Reading the Thesis . . . 17
- 1.4 Related Work . . . 18

I Non-probabilistic Hybrid Systems **21**

2 Interactive State Machines **22**
- 2.1 Preliminaries . . . 22
- 2.2 Hybrid Automata . . . 25
  - 2.2.1 Definition of Hybrid Automata . . . 25
  - 2.2.2 Executions and Traces . . . 26
  - 2.2.3 Composition of HA . . . 27
- 2.3 Hybrid Input/Output Automata . . . 28
  - 2.3.1 Composition of HIOA . . . 29
- 2.4 Structured Hybrid I/O Automata . . . 30
  - 2.4.1 State models . . . 30
  - 2.4.2 Definition of Structured HIOA . . . 33
  - 2.4.3 Some Special Classes of SHIOAs . . . 35
  - 2.4.4 Composition of SHIOA . . . 35
  - 2.4.5 Summary . . . 37

3 The HIOA Language **38**
- 3.1 An Overview . . . 38
- 3.2 Variables . . . 39
  - 3.2.1 Built-in Types . . . 40
  - 3.2.2 Vocabularies . . . 40
  - 3.2.3 Dynamic Types . . . 41
  - 3.2.4 Initial Values . . . 42
- 3.3 Functions . . . 43
- 3.4 Signature . . . 43

3.5 Transitions . . . 44
3.5.1 Preconditions . . . 44
3.5.2 Effects . . . 45
3.6 Trajectories . . . 45
3.6.1 Invariant Condition . . . 46
3.6.2 Stopping condition . . . 46
3.6.3 DAIs . . . 47
3.7 Operations and Properties . . . 49
3.7.1 Composition . . . 49
3.7.2 Property Assertions . . . 50
3.8 Summary . . . 52

4 Verifying Safety Properties **53**
4.1 An Overview . . . 53
4.2 Proving Invariants . . . 54
4.3 Proving Implementation Relations . . . 55
4.4 Case Study: Safety Verification of Helicopter Testbed . . . 56
4.4.1 System Specification . . . 57
4.4.2 Safety Verification . . . 63
4.4.3 Preliminary Properties . . . 65
4.4.4 User Mode . . . 66
4.4.5 Supervisor Mode . . . 68
4.5 Summary . . . 74

5 Verifying Stability Properties **75**
5.1 Assumptions . . . 75
5.2 Stability and Average Dwell Time . . . 76
5.2.1 Stability Definitions . . . 76
5.2.2 ADT Theorem of Heshpanha and Morse . . . 77
5.3 An Overview . . . 79
5.4 ADT Equivalence . . . 80
5.5 Verifying ADT: Invariant approach . . . 82
5.5.1 Transformations for ADT verification . . . 82
5.5.2 Case Study: Leaking Gas-burner . . . 85
5.5.3 Case Study: Scale-independent Hysteresis Switch . . . 86
5.6 Verifying ADT: Optimization-based Approach . . . 89
5.6.1 One-clock Initialized SHIOA . . . 90
5.6.2 Case Study: Linear Hysteresis Switch . . . 91
5.6.3 Initialized SHIOA . . . 96
5.6.4 MILP formulation of OPT(τ_a) . . . 98
5.6.5 Case Study: Thermostat . . . 100
5.7 Summary . . . 102

6 Mechanizing Proofs **104**
6.1 An Overview . . . 104
6.2 Translation . . . 107
6.2.1 Assumptions . . . 107
6.2.2 Types and Vocabularies . . . 108

6.2.3 Variables and Initial States . . . 109
6.2.4 Trajectory Types . . . 109
6.2.5 Actions, State Models, and Moves . . . 110
6.2.6 Transitions . . . 111
6.2.7 Trajectories . . . 113
6.2.8 Invariants . . . 117
6.2.9 Simulation Relations . . . 119
6.3 Strategies . . . 122
6.3.1 Strategies for Proving Invariants . . . 123
6.3.2 Strategies for Proving Forward Simulation . . . 125
6.4 Discussion of Case Studies . . . 129
6.4.1 Failure Detector . . . 129
6.4.2 Two-Task Race . . . 130
6.5 Summary . . . 131

II Probabilistic Hybrid Systems **133**

7 Probabilistic State Machines **134**
7.1 An Overview . . . 134
7.2 Preliminaries . . . 136
7.3 Task-Deterministic Probabilistic Timed I/O Automata . . . 137
7.3.1 Definition of Task DPTIOAs . . . 137
7.3.2 Executions and Traces . . . 140
7.3.3 Composition of Task-DPTIOAs . . . 141
7.4 Probabilistic Semantics for Task-DPTIOAs . . . 142
7.4.1 Semi-ring on Executions and Traces . . . 143
7.4.2 Probability Measure Over Executions . . . 145
7.4.3 Probability Measure Over Traces . . . 147
7.5 Implementation and Compositionality . . . 151
7.6 PTIOAs and Local Schedulers . . . 152
7.7 A Language for Specifying PTIOAs . . . 153
7.8 Summary . . . 157

8 Verifying Approximate Implementation Relations **158**
8.1 An Overview . . . 158
8.2 Task-structured PIOA . . . 160
8.2.1 Definition of Task Structured PIOA . . . 160
8.2.2 Executions and Traces . . . 161
8.2.3 Composition of Task-PIOAs . . . 161
8.2.4 Probabilistic Executions and Trace Distributions . . . 162
8.2.5 Exact implementations and Simulations . . . 163
8.3 Uniform Approximate Implementation . . . 165
8.3.1 Uniform Metric on Traces . . . 165
8.3.2 Expanded Approximate Simulations . . . 166
8.3.3 Soundness of Expanded Approximate Simulations . . . 169
8.3.4 Need for Expansion . . . 173
8.3.5 Probabilistic Safety . . . 176

8.4 Discounted Uniform Approximate Implementation 176
8.4.1 Discounted Uniform Metric on Traces 177
8.4.2 Discounted Approximate Simulation 179
8.4.3 Soundness of Discounted Approximate Simulation 179
8.5 Approximations for Task-PIOAs 181
8.6 Related Work 181
8.7 Summary 182
8.8 Appendix: Limits of Chains of Distributions 183

9 Conclusions **184**
9.1 Evaluation 184
9.2 Future Directions 185
9.2.1 Modeling Probabilistic Hybrid Systems 185
9.2.2 Stability 186
9.2.3 Approximate Implementations 187
9.2.4 Software Tools 187

List of Symbols and Functions **188**

Index **190**

Bibliography **193**

List of Figures

2-1 Example of trajectories of a real-valued continuous variable. 24
2-2 Hybrid automaton model of a vehicle and a typical execution. 26
2-3 Composition of Vehicle and Controller. 28

3-1 Bouncing ball: HIOA specification and an execution. 39
3-2 Vocabulary for graphs. 41
3-3 Periodically sending process. 46
3-4 Thermostat. 48
3-5 Time-bounded channel. 48
3-6 Process participating in a clock synchronization algorithm. 49
3-7 Failure detector specification. 50
3-8 Periodic sender and simple failure detector. 51
3-9 Composed automaton and property assertions. 51

4-1 Helicopter testbed manufactured by Quanser Inc. 57
4-2 Interconnection of SHIOA components in Quanser helicopter system. 58
4-3 Quanser helicopter pitch dynamics. 58
4-4 Periodic noisy sensor. 59
4-5 An arbitrary user-defined controller. 60
4-6 Switching regions of supervisory controller. 61
4-7 Switched supervisory controller for helicopter testbed. 62
4-8 Actuator with delay. 62
4-9 *Left:*A_t regions between U and R, *Right:*B_t regions between R and C. . . . 67

5-1 Leaking gas burner. 85
5-2 Scale-independent hysteresis switch. 87
5-3 Transformed hysteresis switch. 87
5-4 One-clock initialized SHIOA Aut(G) defined by directed graph G. 90
5-5 Linear hysteresis switch. 92
5-6 ADT-equivalent graph ($m = 3$) for LinHSwitch. 93
5-7 Generic rectangular initialized SHIOA. 98
5-8 Objective function and constraints for MOPT(K, τ_a) 99
5-9 Thermostat2 SHIOA and its rectangular initialized abstraction ThermAbs. . 101

6-1 directedGraphs vocabulary in HIOA translated to *directedGraphs* theory in PVS. 109
6-2 Variable declarations in HIOA translated to type declarations in PVS 110
6-3 Actions and State models in HIOA translated to Moves in PVS 111
6-4 Discrete transitions in HIOA and their PVS translation. 113
6-5 Bounce automaton translated to *Bounce* theory in PVS. 114

6-6 Spec of Figure 3-7 translated to *Spec_decls* theory. 116
6-7 Timeout automaton in HIOA 117
6-8 Timeout of Figure 6-7 translated to *Timeout_decls* theory. 118
6-9 Translation of invariant assertions for Timeout. 119
6-10 *SHIOA* PVS theory. 120
6-11 *Forward_simulation* theory. 122
6-12 Translation of forward simulation relation of failure detector. 123
6-13 Base case sequent of simulation proof. 125
6-14 Inductive step sequent for internal actions. 126
6-15 Inductive step sequent for external actions. 127
6-16 Structure of Prove_Fwd_Sim strategy. 128
6-17 Inductive step sequent for trajectories. 128
6-18 TwoTaskRace automaton and its abstract specification. 131
6-19 Forward simulation relation for TwoTaskRace. 132

7-1 Noisy sensor. 155
7-2 Randomized consensus with exponential message delays. 156

8-1 Marginal distributions of the optimal joint distribution ψ for $\hat{\phi}(x_1, y_1) = \epsilon$. . 168
8-2 Discrepancy and *lstate* distributions for $\mathcal{A}_1$ and $\mathcal{A}_2$. 173
8-3 Rand and Trapdoor automata. 174
8-4 Automata representing Ben-Or consensus protocol. 177

Chapter 1

Introduction

1.1 Modeling and Verification of Embedded Software

Software in modern engineering systems is designed to perform sophisticated and safety-critical tasks by interacting with physical environments. Examples of such "embedded software" abound in automotive, avionics, robotics, medical, and process control systems. Testing and simulation based methods alone cannot guarantee the absence of defects nor can they guarantee that the software system has the required properties. In the context of safety-critical embedded software, the mantra of formal modeling and verification—to build a mathematical model of the software system and to *prove* that the model satisfies the required properties—is appealing. If the actual implementation of the software respects its mathematically proved model, then it is also guaranteed to satisfy the properties.

There are two components to model-based software verification, namely, *specification* and *verification*. First, one has to describe the behavior of the software system as a mathematical object which can be reasoned about. Such a mathematical representation is called a *specification*. There are various levels of detail at which one could specify the behavior of a piece of software. At one end, the model could closely mimic the behavior of code (and hardware), and at the other end of the spectrum, the model could only describe what the software should do, but not how it should do it. At all these different levels of abstraction, behavior of software is typically described by discrete state transitions. For complete mathematical description of an embedded software system one has to also capture the evolution of the physical environment of the software—motion, flow of matter, action of forces. Mathematical models that combine differential equations and state transitions are called hybrid models or *hybrid systems*.

The second component is to prove or *verify* that the specification of the software satisfies the properties that are required for correctness. From the early days of software development it has been recognized that proving correctness of even purely sequential state transition systems is a difficult, and often computationally intractable [Dav83]. Although significant progress has been made in verifying relatively shallow properties for programs, software verification in general remains one of the grand challenges in computer science [MMS05]. Software implemented using concurrent threads improve efficiency by exploiting the support of multiprocessing in modern microprocessors, but concurrency gives rise to subtle defects such as race conditions, starving, and deadlocks that are notoriously difficult to find. Embedded software typically employ multiple threads for managing several tasks simultaneously (e.g., sensing and actuation) and hence, verification of hybrid systems inherit

all the difficulties that plague ordinary concurrent systems. In addition, now we also have to reason about continuously evolving states described by differential equations. Analysis of general continuous systems is a daunting task and the tools employed for this purpose, typically from the realm of systems and control theory, are quite different and hence difficult to combine with those employed in analyzing concurrent systems.

1.2 Hybrid Systems

It has been known for some time now that purely algorithmic analysis is impossible for even fairly restricted classes of hybrid systems. Consider, for example, the class of *rectangular, initialized hybrid systems* [HKPV95]. These are hybrid systems with a finite number of clocks, all evolving at *constant* (possibly different) rates as time elapses. When the variables satisfy certain predicates, then they are *initialized* to zero, and a new set of constant rates guide their evolution from then onwards. Further, the predicates which trigger the initializations have to be such that they *do not compare* two clocks. For rectangular, initialized hybrid systems it is possible to to compute the set of reachable states (that is, the attainable clock values) in PSPACE. Computing the reachable set is the key toward algorithmic verification of several types of properties; it can be used to answer questions such as: "Does the system every hit any undesirable states?" However, even if one of the above restrictions—constant rates, initialization, and independent constraints—is relaxed, reachable set computation becomes undecidable.

In practice, therefore, a compromise has to be struck between the expressive power of the class of hybrid models that we use for specifying, and the degree of automation we get in verification. Traditionally, concurrency theorists have focused on subclasses with relatively simple continuous dynamics (described, by linear or rectangular differential equations) but interesting discrete behavior, and researchers in systems and control theory, on the other hand, have placed a greater emphasis on models with more general continuous behavior with isolated switching events.

Instead of focusing on models that are amenable to fully automatic verification, in this thesis, we explore a general class of hybrid models for the purpose of developing specification and verification techniques for embedded software systems. Of course, verification is not going to be fully automatic for these general hybrid models, but we do expect that by bringing in tools from control and optimization theory, we will be able to develop effective techniques. Our strategy for addressing the challenge of general hybrid systems with complex discrete and continuous behavior is to (i) decompose large systems into simpler components, (ii) abstract complex components with simpler ones, and (iii) develop new verification techniques based on deduction and optimization. For certain restricted classes of hybrid systems (iii) also yields brand new, fully automatic, verification procedures. Our results on composition and abstraction can be used in conjunction with the existing algorithmic approaches to reason about composite hybrid systems. Thus, our approach can be viewed to complement the algorithmic approaches.

Before we go any further into discussing the contributions of the thesis, we describe some key aspects of hybrid system specification and verification. Since our developments are based on the *Hybrid Input/Output Automata (HIOA)* model of Lynch, Segala, and Vaandrager [LSV03], we also take this opportunity to introduce the basic terminologies in this framework.

Hybrid behavior. HIOA is a automaton framework for describing discrete and continuous

behavior. The simplest hybrid I/O automaton consists of sets of *internal variables*, *internal actions*, *transitions* and *trajectories*. Valuations of the internal variables define the state of the automaton. The valuations can change discretely through transitions (which are labeled by the internal actions) and continuously over a period of time following a trajectory. No structural or computational restrictions are imposed on the dynamics of the variables over the trajectories and the transitions—this makes HIOAs very expressive. A particular run or a behavior of a HIOA is modeled as an alternating sequence of actions and trajectories, which is called an *execution*. Typical properties of a HIOA $\mathcal{A}$ that one is interested in verifying include invariant properties (all executions of $\mathcal{A}$ remain within a certain set of states), stability properties (e.g., all executions of $\mathcal{A}$ converge to some target state), and timing-related properties (e.g., in every execution, every occurrence of action a is followed by an an occurrence of b within a certain time bound).

Uncertainties and underspecification. Any modeling framework has to provide mechanisms for capturing uncertainties in the system. In HIOA, *nondeterministic* transitions and trajectories provide such a mechanism. Nondeterminism is necessary for constructing (a) models for arbitrarily interleaving concurrent processes, and (b) models that are implementation-free and underspecified. Nondeterminism cannot, however, capture the probabilistic information about uncertainties, such as the probability that a random bit turns out to be 1 or the probability that a processor fails. Probabilistic models make it possible to verify quantitative properties of hybrid systems such as expected time of convergence to an equilibrium state and probability of hitting a set of bad states.

Implementation or abstraction. In reasoning about complex systems it is essential that we are able *abstract* a complicated model with a simpler one, so that the visible behavior of the former is subsumed by that of the latter. The notion of abstraction finds an application in the process of *hierarchical refinement*. Here, one starts with an abstract specification that easily checked to be correct, ands adds more and more implementation details to the point where a detailed enough specification is obtained that can actually be built. If the subsumption relation is preserved in all the intermediate steps, then the final implementation is provably correct.

Interfaces. The external interface of a HIOA is defined by adding sets of *input* and *output variables* and *input* and *output actions* to the simple HIOA model described earlier. Valuations of input/output variables also change over transitions (which are now labeled by internal as well as input/output actions) and trajectories. The externally visible behavior corresponding to an execution, called a *trace*, is obtained by removing all the internal variables and actions from the execution and keeping the input/output variables and actions. A HIOA $\mathcal{A}$ *implements* another HIOA $\mathcal{B}$, written as $\mathcal{A} \leq \mathcal{B}$, if the set of traces of $\mathcal{A}$ is contained in the set of traces of $\mathcal{B}$. This is equivalent to saying that $\mathcal{B}$ is an abstraction for $\mathcal{A}$. The key mechanism for constructing abstract HIOA models is through the use of nondeterminism.

Composition. For analyzing complex systems it is essential that we are able to reason *compositionally*. Informally, this means that we should be decompose the system into a set of interacting components, verify correctness of the individual components, and deduce the correctness of the whole system from the correctness of the components

without much extra work. HIOA models can communicate through shared interfaces, that is, through shared input/output actions and input/output variables. Such communicating HIOAs can be *composed* to get more complex HIOAs. The composition operation respects the notion of implementation. That is, if component $\mathcal{A}$ is an implementation of $\mathcal{B}$, then for every automaton C, the composition of $\mathcal{A}$ and $\mathcal{C}$, $A||C$, implements $\mathcal{B}||\mathcal{C}$. Formally, this property of any class of automaton is called *substitutivity*.

1.3 Thesis Overview

The thesis has two parts. Part I is about specifying and verifying nonprobabilistic hybrid system models. It includes improvements on existing mathematical models, new theoretical results that yield verification techniques, and also design of software tools that embody these techniques. Part II presents a set of foundational results on specifying and analyzing hybrid models with probabilities. It includes the development of semantics for a general class of hybrid systems that supports both nondeterministic and probabilistic choices, and a set of proof techniques for verifying quantitative properties of discretely evolving probabilistic systems. Composition, implementation, substitutivity, and inductive proof techniques for invariance and implementation are the recurrent themes in both parts. In this section, we present an overview of the thesis staring with the different models for hybrid systems that are used and developed, then the verification techniques, and concluding with the supporting software tools.

1.3.1 Modeling

The starting point of the thesis is the *Hybrid Input/Output Automaton (HIOA)* of [LSV03]. A HIOA is a nondeterministic automata which can evolve discretely and continuously, and which can communicate both discretely (through shared actions) and continuously (through shared variables) with other HIOAs. The HIOA model is described in Section 2.3.

Continuous evolution or the trajectories of a HIOA are specified as a set of functions that satisfy certain closure properties. For developing verification techniques that rely on analysis of the trajectories, we would like to have a structured way of specifying them. In Section 2.4, we introduce the *Structured Hybrid I/O Automaton (SHIOA)* model where the trajectories are specified by a collection of *state models*. The state model description of trajectories is similar in spirit to the standard state space representation used for describing continuous time systems in control theory (see, e.g., [Oga97, Lue79]). Each state model consists of Differential and Algebraic Inequalities (DAIs), an invariant condition, and a stopping condition, which define a set of valid trajectories for the SHIOA. We define the composition operation of SHIOA and show that SHIOAs are semantically equivalent to HIOAs. All our developments in Part I are based on SHIOAs.

In HIOAs and SHIOAs, uncertainties are captured as nondeterministic choices. Nondeterminism can describe uncertainty as a set of possible choices, but cannot capture the probability of individual choices. Incorporating probabilities in hybrid system framework gives us a richer language to construct models with. In Chapter 7 of Part II, we introduce a new model for probabilistic hybrid systems called *Probabilistic Timed I/O Automata (PTIOA)*. The continuous evolution of a PTIOA is non-probabilistic, but the discrete transitions can be both probabilistic and non-deterministic. Thus, PTIOAs can be be used to model hybrid systems where failures and message delays are defined by a discrete time stochastic process,

for example, randomized, real-time algorithms, timing based security protocols, control systems with noisy sampling, and randomly switched hybrid systems. It is worth remarking that the *Probabilistic I/O Automaton (PIOA)* of Segala et al. [Seg95b, CCK+06a] that we use in Chapter 8 for developing notions of approximate implementation, is essentially a "discrete" PTIOAs; PIOAs do not have continuous state spaces, trajectories or continuous probability distributions.

The key challenges in developing the semantics for PTIOAs are (i) reconciling the interaction between probabilistic and nondeterministic choices, and (ii) ensuring *measurability* of the various quantities. If we restrict our attention to discrete probability distributions over countable sets alone, then it suffices to assign probabilities to individual elements in the countable set; we can determine the probability of a set S by simply adding up the probability of the individual elements in S. This approach does not work when we have to deal with continuous distributions over uncountable sets. For example, consider an infinite sequence of coin tosses. We cannot determine the probability of getting infinitely many $0's$—which in this case should be 1—by simply adding the probabilities of a set of outcomes. We have to carefully assign probabilities to certain sets of outcomes, and these sets are precisely going to be the *measurable sets* of outcomes, or in the case of PTIOAs, these are going to be the measurable sets of executions. Thus, we are led to use measure theoretic constructions, and in the process we have to solve several technical problems for preserving measurability properties.

Nondeterminism has to be resolved in order to assign probabilities to measurable sets of executions. Nondeterminism in PTIOAs (and in SHIOAs) comes from two sources: (a) *external nondeterminism* : choice of one automaton which makes the next move from a set of interacting automata, and (b) *internal nondeterminism*: choice of one move (transition or trajectory) of an automaton from a set of possible moves. We proceed by first working with *task-structured Deterministic PTIOAs (task-DPTIOAs)*—a subclass of PTIOAs that have limited internal nondeterminism. In order to ensure that all reasonable sets of executions are measurable we impose the following measurability condition on task-DPTIOAs: (1) for any action, the set of states in which the action is enabled a measurable, and (2) for a measurable subset R of $\mathbb{R}_{\geq 0}$ and a measurable set Y of states, the set of states from which there exists a trajectory of length in R and final state in Y, is measurable. For resolving external nondeterminism we rely on the *task schedules* which uniquely determine an automaton which gets to make the next move. Combining a task schedule with a PTIOA $\mathcal{A}$ gives rise to a *probabilistic execution*—a probability measure over the set of executions of $\mathcal{A}$. In Section 7.4, we provide an explicit inductive construction for this measure. We show that the trace function is measurable for PTIOAs, and therefore, each probabilistic execution in turn gives rise to a *trace distribution*—a probability measure over the set of traces of $\mathcal{A}$.

We use a simple, but intuitive notion of *external behavior* for task-DPTIOAs and show that they are substitutive with respect to it. We define the composition operation for PTIOAs and show that the class of PTIOAs (and task-DPTIOAs) are closed under composition, provided the composite automaton satisfies an additional measurability requirement. Finally, based on the trace semantics of task-DPTIOAs, we define the semantics for PTIOAs simply by interpreting a PTIOA as a collection of task-DPTIOA, each of which resolves internal nondeterminism in a different way.

1.3.2 Verification

Verifying Invariant Properties and Implementation Relations

In Chapter 4 we present techniques for establishing invariant properties and implementation relations for SHIOAs. An invariant property $\mathcal{I}$ of an SHIOA $\mathcal{A}$ is deduced by first finding a stronger *inductive property* $\mathcal{I}' \subseteq \mathcal{I}$, and then checking, through case analysis, that all the actions and state models of $\mathcal{A}$ preserve $\mathcal{I}'$. An implementation relation $\mathcal{A} \leq \mathcal{B}$, is deduced by first finding a suitable *simulation relation* $\mathcal{R}$ on the states of $\mathcal{A}$ and $\mathcal{B}$, and then checking that, starting from related states, each transition and trajectory of $\mathcal{A}$ can be "simulated" by a sequence of transitions and trajectories of $\mathcal{B}$, with the same trace, while preserving $\mathcal{R}$.

These proof techniques enable us to construct stylized hand-proofs by systematic analysis of SHIOA specifications. This is a first step toward automation in deductive verification. Indeed, the theorem prover strategies of Chapter 6 which partially automate construction of such stylized proofs, are based on these techniques. Furthermore, because these techniques decouple the reasoning about the transitions and the trajectories, it is possible to apply methods from computer science and control theory within the same proof. For example, to prove that a certain closed set S is an invariant for a given SHIOA, we check that (a) the transitions do not leave S by symbolically computing the post state of the transitions, and (b) the trajectories do not leave S by invoking a well known theorem from control theory on subtangential relationships between the boundary of S and the vector field of the state models.

The proof techniques are applied to several case studies throughout the thesis. In particular, Section 4.4-4.4.2 present an application in verifying the safety of a supervisory controller for a model helicopter system.

Verifying Stability Properties

The inductive proof techniques that are available for proving invariance cannot be directly applied to prove stability. We consider the case of SHIOAs that have stable state models. A theorem by Hespanha and Morse [HM99] tell us that for such SHIOAs, it suffices to verify that $\mathcal{A}$ switches among the different state models "slowly enough", in order to establish stability. This notion of slow switching is formalized by the *Average Dwell Time (ADT)* property of $\mathcal{A}$. In Chapter 5, we present a set of techniques for verifying ADT properties of SHIOAs.

Specifically, we define what it means for a given SHIOA to be equivalent to another SHIOA with respect to ADT, and introduce *switching simulation relations* for proving such relationships between a pair of SHIOAs. Next, we present two complementary techniques for proving ADT properties. The first technique relies on transforming the given automaton $\mathcal{A}$ to a new automaton $\mathcal{A}'$, such that $\mathcal{A}$ satisfies the ADT property in question if and only if $\mathcal{A}'$ has a certain invariant property. Then the techniques developed above for proving invariant properties can be used on $\mathcal{A}'$ to verify the ADT of $\mathcal{A}$. The second technique relies on solving an optimization problem over the set of executions of an automaton, to search for a counterexample execution that violates the candidate ADT property. We show that for *initialized* SHIOAs it is necessary and sufficient to optimize over a small set of execution fragments. In addition, if the SHIOA is rectangular then it is possible to solve the optimization problem efficiently using mixed-integer linear programming.

These two methods for verifying ADT properties complement each other as they can be used in combination to find the average dwell time of a hybrid system. We apply both

the abstraction and verification techniques to verify the ADT, and hence the stability, of several hybrid systems, including a scale-independent hysteresis switch. It is worth noting that ADT properties have proved to be helpful in analyzing different forms of stability in various contexts other than those we study in Chapter 5. For example, in analyzing stability of SHIOAs with stable and unstable state models [ZHYM00], input-to-state stability in the presence of inputs [VCL06], and stochastic stability of randomly switched systems [CL06]. Our ADT verification techniques are likely to be valuable in these other contexts as well.

Verifying Approximate Implementations

Task-structured PIOAs of [CCK+06a] can be viewed as discrete version of the task-DPTIOAs of Chapter 7, in the sense that they do not have continuous evolution and are restricted to probabilistic transitions with discrete probability distributions. Like taks-DPTIOAs, a task schedule resolves the nondeterministic choices in a task-PIOA and gives rise to a probability distribution over its space of executions, which in turn gives a unique trace distribution. A PIOA $\mathcal{A}$ is said to implement PIOA $\mathcal{B}$, if each trace distribution of $\mathcal{A}$ is also a trace distribution of $\mathcal{B}$. But small perturbations to the parameters of $\mathcal{A}$ produces traces distributions with slightly different probabilities and this breaks this implementation relation. In Chapter 8 we define *approximate implementation relations* that not only capture the binary fact of whether or not $\mathcal{A}$ implements $\mathcal{B}$, but also the degree to which $\mathcal{A}$ implements $\mathcal{B}$. Specifically, a PTIOA $\mathcal{A}$ is an ϵ-approximate implementation of $\mathcal{B}$, if every trace distribution of $\mathcal{A}$ is ϵ-close to some trace distribution of $\mathcal{B}$, where closeness is measured by the uniform metric on the space of trace distributions. We present a new class of real-valued *simulation functions*, the existence of which is sound for proving ϵ-approximate implementation relationships between PIOAs. A second notion of *discounted approximate implementation* is introduced based on the discounted uniform metric on trace distributions which allows finer-grained comparison of task-PIOAs. And we develop a simulation based inductive proof technique for discounted approximate implementations. We show how approximate implementations can be used to quantitatively reason about probabilistic safety and robustness of termination probability.

1.3.3 Software Tools

The software tools that we have designed as a part of this thesis facilitate and partially automate the specification and verification techniques for SHIOAs. The software components that have been implemented based on the designs proposed in this thesis include: 1. a specification language, called `HIOA`, for describing SHIOAs. 2. a tool for translating `HIOA` specifications to the language of the PVS theorem prover [ORR+96], 3. a set of strategies (PVS programs) which partially automate invariant and simulation proofs for SHIOAs. In addition, in Chapter 5 we discuss how our results in stability verification can be used in conjunction with model checking tools (such as, PHAVer [Fre05] and HyTech [HHWT97]) and linear program-solvers, for automatic verification of ADT properties.

`HIOA` language

Chapter 3 presents the syntax and the semantics of `HIOA` and illustrates its usage with examples. `HIOA` provides a rich set of language constructs for specifying user-defined types, transitions, state models, compositions and forms the basis for developing SHIOA-verification framework. `HIOA` is an extension of the IOA language [GLTV03]. This language has been

used throughout this thesis and elsewhere for describing vehicle and air-traffic control systems [UL07, MWLF03], cardiac cell models used in systems biology [GMY+07], algorithms for mobile robotics [LMN], real-time and distributed algorithms [FDGL07, ALL+06, CLMT05]. The subset of the HIOA language without input/output variables is called the TIOA [KLMG05]. A front end for TIOA including a GUI-based editor has been implemented and it serves as the backbone of the Tempo Toolset [TEM07].

HIOA to PVS translator

Theorem provers are used to mechanically construct proofs from a given set of definitions and axioms, and hence they provide the highest level of assurance in system verification. With a theorem prover, one can quickly re-check the validity of an existing proof by rerunning the saved proof scripts, after changes have been made to the specification. But, in order to use a theorem prover one has to specify the system, in our case SHIOAs, in the language of the theorem prover. In Section 6.2, we present the design of a scheme for translating HIOA specifications to the language of the PVS—the theorem prover of our choice—hence eliminating the intermediate manual step of specifying SHIOAs in PVS. The key challenge here has been to translate the trajectories of an SHIOA to corresponding *moves* in the PVS representation, so that they can be reasoned about effectively. Based on the design presented in this thesis, a prototype tool for translating TIOA specifications to PVS has been implemented by Lim [Lim01].

Proof Strategies

In order to verify a property of an HIOA specification that has been translated to PVS, the user invokes the prover and supplies a sequence of proof commands to interactively manipulate and resolve all proof obligations. Typically this process is lengthy, tedious to construct, and it always requires careful attention to specification details. In Section 6.3, we present several *PVS strategies* for partially automating this process for proving implementation relations for SHIOAs. A *strategy* is a Lisp program that accesses the state of an ongoing proof, constructs a sequence of proof steps on-the-fly, and applies it to the proof. Our strategies exploit the known structure of SHIOAs and that of inductive simulation proofs to construct sequences of proof steps. Prior to this work, the Timed Automaton Modeling Environment (TAME) of Archer [Arc01] provided several PVS strategies for proving invariant properties of timed I/O automata. These strategies are also compatible with our translation of HIOA to PVS. In Section 6.4 we discuss our experiences in using the HIOA to PVS translator and our proof strategies.

1.3.4 Reading the Thesis

The nonprobabilistic and probabilistic parts of the thesis are independent, but both rely on the basic definitions of Section 2.1. All of Part I relies on the mathematical models of Chapter 2. In order to read that examples throughout Part I, the reader should also look at the semantics of the HIOA language presented in Chapter 3. The verification techniques of Chapter 5 depend only tangentially on those of Chapter 4, and therefore these two chapters can be read independently. The design of the software tools in Chapter 6 rely on the HIOA language and the proof techniques described in Sections 4.2-4.3. Chapter 8 of Part II can be read after Sections 2.1 and 7.2.

1.4 Related Work

In this section we provide an overview of existing work on modeling hybrid systems. Discussion of existing research that is related to our work on verification and supporting software design are presented in the subsequent chapters. We follow the conceptual development of hybrid system models rather than the chronological order in which the models appeared.

Untimed Automata. Models for qualitative reasoning about concurrent systems have been studied in great detail. Examples include ω-automata [CES86], modal logics [MP81, Pnu77], and I/O automata [Lyn96b, GLTV03]. These formalisms either abstract away time completely, retaining only the sequence of actions, or assume the time sequence to be a monotonically increasing sequence of integers. The *I/O automaton* model, for example, has no continuous variables and only a trivial set of trajectories; it is used to model discrete time systems that communicate synchronously. These frameworks are not entirely satisfactory for reasoning about systems that must interact with physical processes and therefore critically depend upon *real-time* constraints.

Timed and Hybrid Automata. Merritt, Modugno, and Tuttle [MMT91] addressed the above problem by proposing a timed version of the I/O automata model, which associated lower and upper real-time bounds with the actions (or sets of actions) of the automaton. The semantics of the resulting *MMT-automaton* model is as follows: from any point in an execution where an action gets enabled, it must actually occurs within the corresponding time bounds. Alur and Dill introduced the *Timed Automaton* model [AD94], in which it is possible to express continuously evolving clock and stop-watch variables, and not just bounds on the time between the successive transitions. The Alur, Henzinger, et al. *Hybrid Automaton (AH)* model [ACH+95, Hen96] generalizes the timed automaton model so that the continuous variables do not necessarily evolve at constant rates. The Alur-Dill timed automata, the Alur-Henzinger hybrid automata and their variants have been the basis for a large body of research on formal-language theoretic study of hybrid systems and for the development of algorithms for automatic analysis (selected references: [Alu91, HKPV95, HNSY94, ACH+95]).

The timed and hybrid I/O automaton models [LV96, DLL97, MMT91, KLSV04, KLSV03, LSV03, KLSV05] have been developed as a framework for describing general hybrid systems with well-defined notions of external behavior, parallel composition, abstraction, and with the aim of creating deductive proof techniques. The only difference between the hybrid and the timed I/O automaton (TIOA) model is that the latter does not allow external (input/output) continuously changing variables. That is, TIOAs can communicate only through shared actions, but not shared variables. TIOAs, HIOAs, and SHIOAs have been used to specify and verify hybrid systems from a variety of domains including vehicle and air-traffic control systems [UL07, MWLF03, LLL99, WLD95, WL96, HL94], systems biology [GMY+07], mobile robotics [LMN], real-time and distributed algorithms [FDGL07, ALL+06, CLMT05].

Unlike the timed automaton model of Alur-Dill, the continuous state variables of a TIOA can have unrestricted dynamics. In this sense, the expressive power of TIOAs is closely related to that of Alur-Henzinger's hybrid automata. However, for the purpose of algorithmic verification, several simplifying assumptions are built-in within the Alur-Henzinger model. For example, it is assumed that the discrete state of any automaton is a finite set of *locations*; the locations determine the continuous dynamics but are not suitable for describing data-structures such as stacks, counters, and trees, which are useful

for modeling computation. In contrast, the discrete transitions of a TIOA (or a HIOA) may modify its discrete state to perform computations. These computations may or may not alter the continuous dynamics.

Switched and Dynamical Systems. The *General Hybrid Dynamical System (GHDS)* model introduced by Branicky, Borkar, and Mitter [BBM98, Bra95] subsumes most other hybrid models including those proposed in [ASL93, BGM93, Bro94, NK93]. A GHDS is an interacting collection of dynamical systems, each evolving on continuous-variable state spaces. The set of continuous variables of each constituent dynamical system may be different. In each of the constituent dynamical system, the dynamics may be continuous time, discrete time, or mixed, and are given by difference or differential equations. The *switched system* model [Lib03, HM99, vdSS00] is a special case of the GHDS model where all the constituent dynamical systems have the same state space and the right hand side of the differential equations defining the dynamical systems are globally Lipschitz continuous. The switched system model has been widely used to obtain stability and controllability related results for hybrid systems. A switched system can be viewed as higher-level abstraction of a HIOA where details of the discrete behavior are abstracted in terms of an exogenous switching signal that brings about the switching between the different dynamical systems.

Continuous Probabilistic Models without Nondeterminism. Probabilistic or stochastic models are used to capture uncertainties about the system model. Uncertainties may affect the behavior of a hybrid system model, in many different ways. For example, there might be uncertainties about the outcome of the discrete transitions, the parameters of the differential equations might be uncertain, or there may be some white-noise-like disturbance affecting the continuous evolution. Consequently, various models for probabilistic hybrid systems are possible.

In the *Stochastic Hybrid System (SHS)* model of [HLS00], the transitions between the modes of the system are guided by a discrete time Markov chain, and within each mode the evolution of the continuous variables is described by stochastic differential equations. In the SHS model of [Hes04], on the other hand, transitions between discrete modes are triggered by transitions between states of a continuous-time Markov chain and the rate at which transitions occur is allowed to depend both on the continuous and the discrete states. Both these models develop the theory for finding invariant distributions over the state space. These models do not define external behavior or composition of model components and they do not permit nondeterminism in the models.

Discrete State-space Probabilistic Models. Probabilistic extensions of I/O automata were presented by Segala in [Seg95b, Seg96, Seg95a]. The notion of traces is generalized to *trace distributions* that define the external behavior of a probabilistic automaton. Each trace distribution is induced by a probabilistic scheduler which resolves all nondeterministic choices. These extensions are natural, but the resulting abstraction relations are not compositional. Difficulties arise from the interaction between probabilistic choice and the resolution of nondeterminism of the model. This is because nondeterministic choices are resolved by a powerful global scheduler, which can use arbitrary information about the execution so far in resolving nondeterministic choices. For example, such a scheduler may resolve nondeterministic choices in one automaton component in a composed system based on internal state information of the other component. A special case of PIOAs, *switched PIOA*s, has been proposed by Cheung et al. [CLSV04] in which these difficulties have been overcome by carefully defining a local scheduler for each automaton, which resolves local

nondeterministic choices using local information only.

Continuous Probabilistic Models with Nondeterminism. In order to verify hybrid systems, such as sensor networks and mobile robots, that have traits of both hybrid and distributed systems we need a framework supporting continuous dynamics, probabilistic transitions and nondeterminism. The interplay between probability and nondeterminism makes the development of semantics such frameworks challenging [Seg95b, MOW04, Che06, CCK+06b]. Introduction of continuous state spaces and distributions adds another layer of complexity to the problem [CSKN05, vBMOW05, DDLP05].

Recently, several continuous state probabilistic automaton models have been proposed. In *Labelled Markov Processes* (see e.g., [DDLP05, vBMOW05]) state transitions can give rise to continuous probability distributions. In *Piecewise Deterministic Markov Processes (PDP)* [Dav93] discrete transitions are probabilistic and the continuous evolution of state in between those transitions is deterministic. In the *Communicating Piecewise Deterministic Markov Processes (CPDP)* model of [SvdS05], component PDPs communicate discretely through shared events. Existing models do not permit *internal nondeterminism*. That is, choice of an action uniquely determines a transition, which in turn gives a probability distribution over the states. Modeling frameworks that support composition of automata have to resolve *external nondeterminism*, that is, the choice of which automaton gets to make the next move. This nondeterminism can be replaced by a race between the automata [ES03, Sta03], else it can be explicitly resolved by a *scheduler* [CSKN05, Che06, CCK+06b]. Nondeterminism can also be allowed by treating the probabilistic and nondeterministic transitions as separate kinds of objects [Her02]. In CPDPs [SvdS05], on the other hand, nondeterminism is resolved using the maximal progress strategy and a randomized scheduler.

Our *Probabilistic Timed Input/Output Automata (PTIOA)* framework shares certain features with the *Stochastic Transition Systems (STS)* of [CSKN05]. Both frameworks allow continuous state spaces, general probability distributions, and nondeterminism. An STS, however, does not have notions of time or trajectories. This leads to very different semantics for the two frameworks and also important technical differences in the underlying construction of probability spaces. We discuss these issues in Section 7.4.

Part I

Non-probabilistic Hybrid Systems

Chapter 2

Interactive State Machines

Throughout Part I we shall specify (nonprobabilistic) hybrid systems as *Structured Hybrid I/O Automata (SHIOA)*. The state of an SHIOA may change instantaneously as a result of the occurrence of some *discrete transition*, or it may evolve continuously over a period of time according to some *trajectory*. From a particular state, if multiple transitions and trajectories are possible, then the system evolves by nondeterministically choosing one. SHIOA specializes the Hybrid I/O Automaton (HIOA) model of Lynch, Segala, and Vaandrager [LSV03] with additional structures which facilitate description and manipulation of trajectories. In Part II, we will consider a generalization of the SHIOA model which allows probabilistic discrete transitions.

SHIOAs are suitable for modeling physical and computing processes at different levels of abstraction. For example, at one level, the delayed change in the output of a logic gate can be modeled as a nondeterministically chosen time delay followed by a discrete transition. In a more detailed model, the transition may be triggered when the gate current trajectory stabilizes within a certain range. In the SHIOA framework, we can then formally state (and prove) that the latter model is an *implementation* of the former. A complex system is described as a *composition* of a set of SHIOAs that interact through shared variables and transition labels. This chapter presents the basic definitions and semantics, taken mostly from [LSV03], for HIOAs, SHIOAs, their compositions, and implementation relations.

2.1 Preliminaries

Sets and Functions. The complement of a set A is denoted by A^c. The union of a collection $\{A_i\}_{i\in I}$ of pairwise disjoint sets indexed by a set I is written as $\biguplus_{i\in I} A_i$. For any function f we denote the domain and the range of f by $dom(f)$ and $range(f)$. For a set S, we write $f \lceil S$ for the restriction of f to S, that is, the function g with $dom(g) = dom(f) \cap S$, such that $g(c) = f(c)$ for each $c \in dom(g)$. If f is a function whose range is a set of functions, then we write $f \downarrow S$ for the function g with $dom(g) = dom(f)$ such that $g(c) = f(c) \lceil S$ for each $c \in dom(g)$. For an indexed tuple or an array b with n elements, we use the special notation $b[i]$ for referring to its ith element.

Time. We measure time by numbers in the set $\mathbb{T} \triangleq \mathbb{R}_{\geq 0} \cup \{\infty\}$. A *time interval* is a nonempty, convex subset of $\mathbb{T}$. For an interval $K \subseteq \mathbb{T}$ and any $t \in \mathbb{T}$, we define the *t-shifted* interval as $K + t \triangleq \{t' + t \mid t' \in K\}$. For a function $f : K \to \mathbb{R}$ and $t \in \mathbb{T}$, the *t-shifted* function, $(f + t) : (K + t) \to \mathbb{R}$, is defined as $(f + t)(t') = f(t' - t)$, for each

$t' \in K + t$. The pasting of two functions f_1 and f_2, where the domain of f_1 is right closed and $\max(dom(f_1)) = \min(dom(f_2)) = t'$, is defined to be the function $f_1 \diamond f_2(t) \triangleq f_1(t)$ for each $t \leq t'$, and $f_2(t)$ for $t > t'$. Similarly, a finite sequence of functions can be pasted if all the non-final functions have right closed domains.

Variables. A *variable* is a name for either a component of the system's state or a channel through which information flows from one part of the system to another. Each variable v is associated with a *static type* (or simply type) and a *dynamic type*. The static type of v, $type(v)$, is the set of values that v can take. A *valuation* $\mathbf{v}$ for a set of variables V is a function that associates each variable $v \in V$ to a value in $type(v)$. The set of all valuations of V is denoted by $val(V)$.

The dynamic type of v imposes certain wellformedness criterion on how the value of v can change over time intervals. This allows us to deduce basic properties of variables (e.g., input variables) that are otherwise unconstrained.

Definition 2.1. For any variable v its *dynamic type*, $dtype(v)$, is a set of functions from left-closed time intervals to $type(v)$ that satisfies the following properties:

DT1 *(Shift)* For any $f \in dtype(v)$, and $t \in \mathbb{T}$, $f + t \in dtype(v)$.

DT2 *(Subinterval)* For any $f \in dtype(v)$, left-closed interval $J \subseteq dom(f)$, $f \lceil J \in dtype(v)$.

DT3 *(Pasting)* If $f_0, f_1, \ldots f_n \in dtype(v)$, such that for all $i < n$, $dom(f_i)$ is right-closed and $\max(dom(f_i)) = \min(dom(f_{i+1}))$. Then $f_1 \diamond f_2 \diamond \ldots f_n \in dtype(v)$.

The third requirement **DT3** is necessary for modeling jumps in the values of the variable resulting from discrete transitions. Dynamic types are constructed by taking the pasting closure of sets of functions, such as continuous functions, continuously differentiable functions, k-times differentiable functions, Lipschitz functions, and smooth functions. We define two special kinds of variables based on two dynamic types that frequently appear in hybrid system specifications.

Definition 2.2. A variable v is said to be *continuous* if (1) $type(v) = \mathbb{R}^n$, for some natural number n, and (2) $dtype(v)$ is the pasting closure of the class of functions where each function f maps a left-closed time-interval J to $type(v)$ and f is continuous with respect to the Euclidean topologies on J and $type(v)$. A variable v is said to be *discrete* if $dtype(v)$ is the pasting closure of the class of constant functions from left-closed time-intervals to $type(v)$.

Any function in the dynamic type of a continuous variable is piece-wise continuous and is continuous from the left at each point. By the above definition, a variable v whose *type* is $\mathbb{R}^n$ and dynamic type is the pasting closure of constant functions from left-closed time-intervals to $type(v)$, is a discrete variable. Thus, real-valued variables can be either discrete or continuous or neither.

Example 2.1. Real-valued continuous variables are useful for describing the evolution of physical quantities such as temperature, velocity, etc., and also resettable timers, and piece-wise continuous signals. Real-valued discrete variables are useful for modeling sampled data, for example, positional coordinates from a periodically broadcasting GPS device. Implementation of algorithms typically involve data structures such as counters, stacks, trees, graphs, etc. These are captured by discrete variables with the appropriate types.

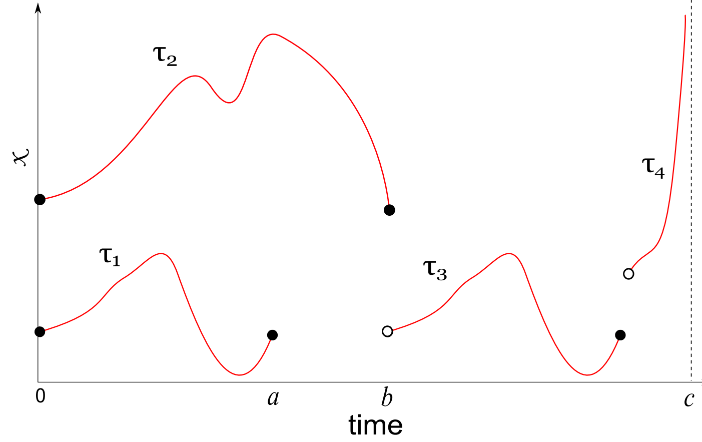

Figure 2-1: Example of trajectories of a real-valued continuous variable.

Trajectories. A trajectory for a set of variables V describes the evolution of the values of the variables over a certain time interval. A *trajectory* τ of V is a function $\tau : J \rightarrow val(V)$, where J is a left-closed interval of time with left endpoint equal to 0, such that for each $v \in V$, $\tau \downarrow v \in dtype(v)$. That is, the restriction of τ to v is a function that conforms to the dynamic type of v. The set of all trajectories for the set of variables V is denoted by $trajs(V)$.

A trajectory τ with domain $[0, 0]$ is called a *point trajectory*. We say that a trajectory τ is *finite* if $dom(\tau)$ is of finite length, *closed* if $dom(\tau)$ is (finite) right closed, and *open* if $dom(\tau)$ is a right open interval. If τ is closed its *limit time* is the supremum of $dom(\tau)$, also written as $\tau.ltime$. Also, we define $\tau.fval$, the *first valuation* of τ, to be $\tau(0)$, and if τ is closed, we define $\tau.lval$, the *last valuation* of τ, to be $\tau(\tau.ltime)$.

Trajectory τ is a *prefix* of trajectory τ', denoted by $\tau \leq \tau'$, if τ can be obtained by restricting τ' to a subinterval $[0, t]$ of its domain, for some $t \in dom(\tau')$. Trajectory τ is a *suffix* of τ', if there exists $t \in dom(\tau')$ such that $(\tau' \lceil [t, \infty)) - t = \tau$. The *concatenation* of a closed trajectory τ and another trajectory τ' is the function $\tau \frown \tau' : (\tau.dom \cup \tau'.dom + \tau.ltime) \rightarrow val(V)$ defined as $(\tau \frown \tau')(t) \triangleq \tau(t)$ for all $t \leq \tau.ltime$ and $\tau'(t)$ for $t > \tau.ltime$. Alternatively, the concatenated function can be defined as $\tau \frown \tau' \triangleq \tau \cup (\tau' \lceil (0, \infty) + \tau.ltime)$. Notice that the domain of τ' is restricted by a left-open interval, and therefore $\tau \frown \tau'(\tau.ltime)$ is $\tau.lval$ and not $\tau'.fval$. As the dynamic types of the variables in V are closed under shift and pasting, the concatenated trajectory $\tau \frown \tau'$ is a valid trajectory for V.

Example 2.2. Let x be a real-valued continuous variable. Several trajectories of x are shown in Figure 2-1. The domains of the closed trajectories τ_1 and τ_2 are the intervals $[0, a]$ and $[0, b]$. The function τ_3 is obtained by b-shifting τ_1, and therefore is contained in the dynamic type of x. The trajectory $\tau_2 \frown \tau_1$ is defined as the point-wise union of τ_2 and τ_3. Note that $(\tau_2 \frown \tau_1)(b)$ is defined to be $(\tau_2 \frown \tau_3)(b) = \tau_2(b)$ and not $\tau_3(b)$.

2.2 Hybrid Automata

In this section, we introduce the *hybrid automaton* model of [LSV03].

2.2.1 Definition of Hybrid Automata

Definition 2.3. A *hybrid automaton* (HA) $\mathcal{H} = (X, W, Q, \Theta, H, E, \mathcal{D}, \mathcal{T})$ consists of:

(a) Disjoint sets W and X of *external* and *internal variables*. The internal variables are also called *state* variables. The set of *variables* V is defined as $W \cup X$.

(b) A set $Q \subseteq val(X)$ of *states* and a non-empty subset $\Theta \subseteq Q$ of *start states*.

(c) Disjoint sets E and H of *external* and *internal actions*. The set of *actions* $A \triangleq E \cup H$.

(d) A set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions*.

(e) A set $\mathcal{T}$ of *trajectories* for V, such that for every trajectory τ in $\mathcal{T}$, and for every $t \in dom(\tau)$, $(\tau \downarrow X)(t) \in Q$. The set of trajectories $\mathcal{T}$ satisfies the following closure properties:

T1 *(Prefix closure)* For every $\tau \in \mathcal{T}$ and every prefix τ' of τ, $\tau' \in \mathcal{T}$.

T2 *(Suffix closure)* For every $\tau \in \mathcal{T}$ and every suffix τ' of τ, $\tau' \in \mathcal{T}$.

T3 *(Concatenation closure)* If $\tau_0, \tau_1, \ldots \in \mathcal{T}$ is a sequence of trajectories such that $\tau_i.lval \lceil X = \tau_{i+1}.fval \lceil X$ for each non-final index i, then $\tau_0 \frown \tau_1 \ldots \in \mathcal{T}$.

Notations. A transition $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$ is written in short as $\mathbf{x} \xrightarrow{a}_{\mathcal{H}} \mathbf{x}'$ or as $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ when $\mathcal{H}$ is clear from the context. If $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, we say that action a is enabled at $\mathbf{x}$. The set of states at which a is enabled is denoted by $enabled(a)$. The *first state* of a trajectory τ, $\tau.fstate$, is $\tau.fval \lceil X$, and *last state* of a closed τ, $\tau.lstate$, is $\tau.lval \lceil X$. We often denote the components of a HA $\mathcal{H}$ by $X_{\mathcal{H}}, W_{\mathcal{H}}, Q_{\mathcal{H}}, \Theta_{\mathcal{H}}$, etc., and the components of a HA $\mathcal{H}_i$ by X_i, W_i, Q_i, Θ_i, etc.

Example 2.3. Consider a HA model of a vehicle whose brakes are controlled externally. Figure 2-2 shows the variables and actions of this automaton. As a convention, throughout this thesis we indicate external actions by dashed arrows and external variables by solid arrows. The Vehicle automaton has real-valued continuous state variables x_1, x_2, and x_3, corresponding to the position, the velocity, and the acceleration, and a discrete boolean variable b, which indicates whether or not the brakes are engaged. Here we describe the components of Vehicle in English. In Section 2.4.1 we shall describe how the trajectories can be succinctly described mathematically, and in Chapter 3 we will present a formal language for specifying HAs.

The set of states or the *state space* Q of Vehicle is $\mathbb{R}^3 \times \{0, 1\}$. The external actions brakeOn and brakeOff are enabled at each state in Q. If the action brakeOn (or brakeOff) occurs at a state $\mathbf{x}$, the value of the variable b is set to $true$ ($false$ respectively) and the other variables remain unchanged. This defines the discrete transitions of Vehicle. Over any trajectory τ of Vehicle, the variable b remains constant, and y equals x_1. If τ starts with $b = true$, the deceleration x_3 remains in a range $[-a_{max}, -a_{min}]$, and otherwise x_3 is zero. The velocity x_2 and the position x_1 variables follow the usual kinematic equations $\dot{x_2} = x_3$ and $\dot{x_1} = x_2$. These conditions specify the set of trajectories $\mathcal{T}$ of the Vehicle automaton. It can be checked easily that the set $\mathcal{T}$ is closed under prefix, suffix, and concatenation.

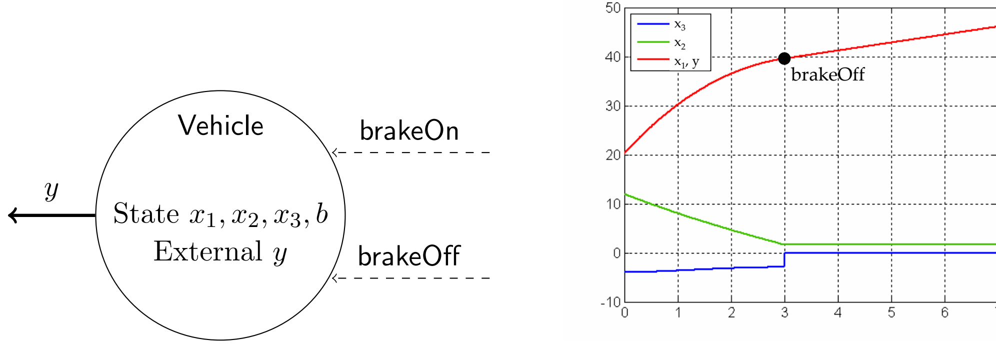

Figure 2-2: Hybrid automaton model of a vehicle and a typical execution.

2.2.2 Executions and Traces

An execution fragment of a hybrid automaton $\mathcal{H}$ describes a particular behavior or run of $\mathcal{H}$. Formally, an *execution fragment* is an alternating sequence of actions and trajectories $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$, where (1) each $\tau_i \in \mathcal{T}$, and (2) if τ_i is not the last trajectory then $\tau_i.lstate \overset{a_{i+1}}{\rightarrow} \tau_{i+1}.fstate$. The *first state* of an execution fragment α, $\alpha.fstate$, is $\tau_0.fstate$. The graph of an execution fragment of the Vehicle HA starting from the state $x_1 = 20, x_2 = 12, x_3 = -4$, and $b = true$ is shown in Figure 2-2.

An execution fragment is *closed* if it is a finite sequence and the last trajectory is closed. The *last state* of a closed execution α, $\alpha.lstate$, is $\tau_n.lstate$, where τ_n is the last trajectory of α. The *limit time* of such an execution fragment, $\alpha.ltime$, is defined as $\sum_{i=0}^{n} \tau_i.ltime$. An execution fragment α is an *execution* if $\alpha.fstate \in \Theta$. The (A_1, V_1)-restriction of an execution keeps information about occurrence of actions in A_1 and evolution of the variables in V_1 and filters out everything else.

Definition 2.4. Suppose $\mathcal{A}$ is a HA with set of variables V and set of actions A and let α be an execution of $\mathcal{A}$. The (A_1, V_1)-*restriction* of an execution α is defined as:

$$\begin{aligned} \alpha \lceil (A_1, V_1) &= \tau \downarrow V_1 \text{ if } \alpha \text{ is a single trajectory,} \\ \alpha a \tau \lceil (A_1, V_1) &= \begin{cases} (\alpha \lceil (A_1, V_1)) \; a \; (\tau \downarrow V_1) & \text{if } a \in \mathcal{A}_1, \\ (\alpha \lceil (A_1, V_1)) \frown (\tau \downarrow V_1) & \text{otherwise.} \end{cases} \end{aligned}$$

A state $\mathbf{x} \in Q$ is said to be *reachable* if it is the last state of some execution of $\mathcal{A}$. An execution fragment is reachable if its first state is reachable. That is, any suffix of an execution is a *reachable execution fragment*. The set of all reachable states of $\mathcal{A}$ is denoted by $\mathsf{Reach}_{\mathcal{A}}$.

The *length* of a finite execution is the number of trajectories in the sequence. We define the following shorthand notation for the valuation of the state variables of $\mathcal{H}$ in an execution α at time $t \in [0, \alpha.ltime)$, $\alpha(t) \triangleq \alpha'.lstate$, where α' is the longest prefix of α with $\alpha'.ltime = t$. For a closed execution α, the notation extends to all $t \in [0, \alpha.ltime]$. The Euclidean norm of $\alpha(t)$ restricted to the set of real-valued continuous variables is denoted by $|\alpha(t)|$. We extend this notation to $|\alpha.lstate|$ and $|\alpha.fstate|$. The set of all executions and execution fragments of $\mathcal{A}$ are denoted by $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Frags}_{\mathcal{A}}$, respectively.

Many interesting properties of hybrid automata can be stated in terms of its executions and reachable states. For example, an *invariant property* or simply an *invariant* is a predicate on the state variables that is true in all reachable states. Given an invariant $\mathcal{I}$, we

often identify the name of the invariant $\mathcal{I}$ with the set of states that satisfy it. Hence, for any invariant $\mathcal{I}$ of $\mathcal{A}$, $\mathsf{Reach}_{\mathcal{A}}$ is contained in $\mathcal{I}$ and $\mathcal{I}$ can serve as an over approximation of $\mathsf{Reach}_{\mathcal{A}}$. Invariants are also useful for specifying *safety properties* such as the property that the velocity of a vehicle *always* remains within some range.

Stability properties of hybrid automata, introduced in [MLL06], can also be stated in terms of executions. For example, a hybrid automaton is said to be *globally uniformly asymptotically stable*, if for any $\epsilon > 0$ and any state q_0, there exists a $T_{\epsilon,q_0} \in \mathbb{T}$, such that for any execution fragment α with $\alpha.fstate = q_0$, for all $t \geq T_{\epsilon,q_0}$, $|\alpha(t)| \leq \epsilon$. Such properties capture requirements such as the velocity of the vehicle should *eventually* (or within a certain time) *converge* to a target range, even though in the interim the velocity may exceed the range. In Chapters 4 and 5 we will present techniques for proving safety and stability properties and their applications.

Often, we are interested in the externally visible part of an execution, which is called a trace. The *trace* of α, denoted by $trace(\alpha)$, is the (E, W)-restriction of α. For example, the traces of the HA $\mathsf{Vehicle}$ are alternating sequences of trajectories of y and ($\mathsf{brakeOn}$ or $\mathsf{brakeOff}$) actions. The set of all traces of $\mathcal{H}$ is denoted by $\mathsf{Traces}_{\mathcal{H}}$.

Sometimes we specify the desired observable properties of an automaton $\mathcal{H}_1$ as another, perhaps more abstract, automaton $\mathcal{H}_2$. Then, to show that $\mathcal{H}_1$ indeed satisfies the desired properties we show that $\mathsf{Traces}_{\mathcal{H}_1} \subseteq \mathsf{Traces}_{\mathcal{H}_2}$. This motivates our next definition of implementation relation.

Definition 2.5. Hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ are *comparable* if they have the same external interface, that is, if $W_1 = W_2$ and $E_1 = E_2$. If $\mathcal{H}_1$ and $\mathcal{H}_2$ are comparable then we say that $\mathcal{H}_1$ *implements* $\mathcal{H}_2$, denoted by $\mathcal{H}_1 \leq \mathcal{H}_2$, if $\mathsf{Traces}_{\mathcal{H}_1} \subseteq \mathsf{Traces}_{\mathcal{H}_2}$.

In Chapters 4 and 5 we shall present techniques for verifying implementation relations and their applications. In Chapter 6 we will discuss how such proofs can be mechanized using theorem proving tools.

2.2.3 Composition of HA

The composition operation enables us to construct a new hybrid automaton from a pair of interacting automata by identifying external actions and variables with the same name.

Definition 2.6. Hybrid automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if $H_1 \cap A_2 = H_2 \cap A_1 = \emptyset$ and $X_1 \cap V_2 = X_2 \cap V_1 = \emptyset$. If $\mathcal{H}_1$ and $\mathcal{H}_2$ are compatible then their composition $H_1||H_2$ is defined to be $\mathcal{H} \triangleq (X, W, Q, \Theta, H, E, \mathcal{D}, \mathcal{T})$, where

(a) $X = X_1 \cup X_2$ and $W = W_1 \cup W_2$.

(b) $Q = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in Q_1 \wedge \mathbf{x} \lceil X_2 \in Q_2\}$ and
$\Theta = \{\mathbf{x} \in Q \mid \mathbf{x} \lceil X_1 \in \Theta_1 \wedge \mathbf{x} \lceil X_2 \in \Theta_2\}$.

(c) $H = H_1 \cup H_2$ and $E = E_1 \cup E_2$.

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

(e) $\mathcal{T} \subseteq trajs(V)$ is given by $\tau \in \mathcal{T}$ iff $\tau \downarrow V_i \in \mathcal{T}_i$, $i \in \{1, 2\}$.

The next theorem from [LSV03] states that the class of HAs is closed under composition.

Theorem 2.1. *If $\mathcal{H}_1$ and $\mathcal{H}_2$ are hybrid automata then $\mathcal{H}_1||\mathcal{H}_2$ is a hybrid automaton.*

The following Lemma from [KLSV05] states that execution fragments of a composition of HAs project to give execution fragments of the component automata. Moreover, the fragments of the composition have certain properties if and only if the component fragments also have similar properties.

Lemma 2.2. *Let $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ and let α be an execution fragment of $\mathcal{A}$. Then $\alpha \lceil (A_1, V_1)$ and $\alpha \lceil (A_2, V_2)$ are execution fragments of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Furthermore,*

1. *α is time bounded iff both $\alpha \lceil (A_1, V_1)$ and $\alpha \lceil (A_2, V_2)$ are time bounded.*
2. *α is closed iff both $\alpha \lceil (A_1, V_1)$ and $\alpha \lceil (A_2, V_2)$ are closed.*
3. *α is an execution iff both $\alpha \lceil (A_1, V_1)$ and $\alpha \lceil (A_2, V_2)$ are executions.*

Example 2.4. Consider a controller that reads the external variable y of the Vehicle automaton of Example 2.3 and triggers the brakeOn and brakeOff actions. Figure 2-3 shows the composition of the Vehicle with such a Controller. The composed automaton Vehicle||Controller has external variable y and external actions brakeOn and brakeOff actions. Theorem 2.1 ensures that this composed automaton is also a HA. Lemma 2.2 tells us that for any execution α of Vehicle||Controller the "projection" of α on brakeOn, brakeOff, and $\{y\}$, that is, $\alpha \lceil (\{\text{brakeOn}, \text{brakeOff}\}, \{y, \ldots\})$ is an execution of Controller. Similarly, $\alpha \lceil (\{\text{brakeOn}, \text{brakeOff}\}, \{x_1, x_2, x_3, y\})$ is an execution of Vehicle.

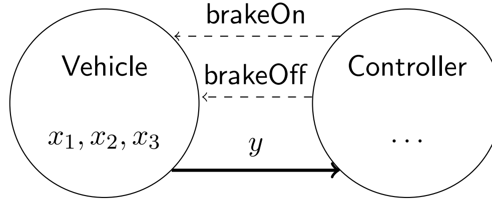

Figure 2-3: Composition of Vehicle and Controller.

We conclude the section on HAs with another theorem from [LSV03], which provides a method for proving implementation relations for composed systems by first proving implementation of its components.

Theorem 2.3. *Suppose $\mathcal{H}_1$ and $\mathcal{H}_2$ are comparable hybrid automata and $\mathcal{H}_1 \leq \mathcal{H}_2$. If $\mathcal{B}$ is a HA that is compatible with each of $\mathcal{H}_1$ and $\mathcal{H}_2$, then $\mathcal{H}_1||\mathcal{B}$ and $\mathcal{H}_2||\mathcal{B}$ are comparable and $\mathcal{H}_1||\mathcal{B} \leq \mathcal{H}_2||\mathcal{B}$.*

This theorem states what is called the *substitutivity* property of HAs: If $\mathcal{H}_1$ implements H_2 then in the context of of any automaton $\mathcal{B}$, $\mathcal{H}_2$ can be substituted by $\mathcal{H}_1$.

2.3 Hybrid Input/Output Automata

The *Hybrid Input/Output Automaton (HIOA)* model is a refinement of the HA model in which certain external actions and variables are distinguished as inputs while the rest are outputs.

Definition 2.7. A *hybrid Input/Output automaton (HIOA)* $\mathcal{A}$ is a tuple $(X, Y, U, Q, \Theta, H, O, I, \mathcal{D}, \mathcal{T})$ where:

(a) $\mathcal{H} = (X, Y \cup U, Q, \Theta, H, O \cup I, \mathcal{D}, \mathcal{T})$ is a hybrid automaton,

(b) U and Y are disjoint sets of *input* and *output* variables. The set $Z \triangleq X \cup Y$ is called the set of *local variables*.

(c) I and O are disjoint sets of *input* and *output* actions. The set $L \triangleq H \cup O$ is called the set of *locally controlled* actions. The set $A \triangleq E \cup H$ is called the set of *actions*.

In addition, $\mathcal{A}$ satisfies the following axioms:

E1 *(Input action enabled)* For every $\mathbf{x} \in Q$ and $a \in I$, there exists $\mathbf{x}' \in Q$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

E2 *(Input trajectory enabled)* For every $\mathbf{x} \in Q$ and every $\upsilon \in trajs(U)$, there exists $\tau \in \mathcal{T}$, such that $\tau.fstate = \mathbf{x}, \tau \downarrow U \leq \upsilon$, and either (a) $\tau \downarrow U = \upsilon$, or (b) τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

The **E1** and **E2** axioms capture the non-blocking assumptions of the model: an automaton cannot choose to ignore actions and variables over which it has no control, namely the input actions and variables. It will sometimes be convenient to consider automata in which the external variables and actions are partitioned into input and output, but the axioms **E1** and **E2** are not necessarily satisfied. We call such an automaton a *pre-HIOA*. Executions and traces of a pre-HIOA $\mathcal{A}$ are defined to to be the executions and traces of the underlying HA. We extend the notations for HAs introduced in 2.2.2 to pre-HIOAs.

Example 2.5. For the Vehicle automaton of Example 2.3, if we classify brakeOn and brakeOff as input actions and the variable y as an output variable, then the resulting automaton is a HIOA. Axiom **E1** is satisfied because the input actions are enabled in all states. **E2** is satisfied vacuously as there are no input variables.

A pair of HIOAs are *comparable* if they have the same external interface. Implementation of comparable HIOAs is defined in a manner that is analogous to the corresponding definition for HAs.

Definition 2.8. Two pre-HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* 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 A_1 *implements* $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \leq A_2$, if $\mathsf{Traces}_{\mathcal{A}_1} \subseteq \mathsf{Traces}_{\mathcal{A}_2}$.

2.3.1 Composition of HIOA

Definition 2.9. Pre-HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible if $\mathcal{H}_1$ and $\mathcal{H}_2$ are compatible, and $Y_1 \cap Y_2 = O_2 \cap O_1 = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-HIOAs then their composition $A_1 || A_2$ is defined to be $\mathcal{A} \triangleq (X, Y, U, Q, \Theta, H, O, I, \mathcal{D}, \mathcal{T})$, where

(a) $X, Q, \Theta, H, \mathcal{D}$ and $\mathcal{T}$ are defined in the same way as in Definition 2.6.

(b) $Y = Y_1 \cup Y_2$ and $U = (U_1 \cup U_2) - Y$.

(c) $O = O_1 \cup O_2$ and $I = (I_1 \cup I_2) - O$.

Like hybrid automata, we would like to have a theorem that states that HIOAs are closed under composition. However, in general the composition of two HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is not a HIOA. The problem arises when the the input variables of $\mathcal{A}_1$ are output variables of $\mathcal{A}_2$ and vice-versa, such that the trajectories of these variables defined by the automata are inconsistent (see Example 6.11 in [LSV03]). As a result, the composed entity $\mathcal{A}_1||\mathcal{A}_2$ does not have trajectories that allow time to elapse, and therefore it does not satisfy **E2**. Thus, we have the following weaker theorem (from [LSV03]) which states that pre-HIOAs are closed under composition.

Theorem 2.4. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible hybrid I/O automata then $\mathcal{A}_1||\mathcal{A}_2$ is a pre-HIOA.*

The next theorem is the HIOA analogue of the substitutivity Theorem 2.5.

Theorem 2.5. *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable pre-HIOAs and $\mathcal{A}_1 \leq \mathcal{A}_2$. If $\mathcal{B}$ is a pre-HIOA that is compatible with each of $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{A}_1||\mathcal{B}$ and $\mathcal{A}_2||\mathcal{B}$ are comparable and $\mathcal{A}_1||\mathcal{B} \leq \mathcal{A}_2||\mathcal{B}$.*

2.4 Structured Hybrid I/O Automata

The set of trajectories of a HA or a HIOA is described by a set of functions that satisfy the closure properties **T1-3**. For developing verification techniques that rely on analysis of the trajectories, we would like to have a structured way of specifying them. *State models* consisting of differential and algebraic equations have served as a standard language for describing the trajectories of continuous time systems (see any standard textbook in control theory, e.g., [Oga97, Lue79]). A hybrid system may have several state models, each describing the dynamics of the continuous variables under a different set of conditions. In this section, we describe *Structured Hybrid I/O Automata (SHIOA)* which was first introduced in [MLL06]. An SHIOA is similar to a HIOA except that it uses a collection of state models for partitioning the state space into several pieces and for describing the trajectories within each piece by *Differential and Algebraic Inequalities (DAIs)*.

2.4.1 State models

First, we define state models independent of any automaton, then, in Section 2.4.2, we use state models for defining trajectories of SHIOAs. We assume that all variables are either discrete or continuous. Suppose V is a set of variables. For a real-valued continuous variable $x \in V$, a *V-algebraic inequality* is an expression of the form $x \lesseqgtr f(\mathbf{v})$, where f is a function that maps $val(V)$ to $\mathbb{R}$ and $\lesseqgtr$ is an element of the set $\{=, \leq, \geq, <, >\}$. A trajectory $\tau \in trajs(V)$ *satisfies* the inequality $x \lesseqgtr f(\mathbf{v})$, if

$$\forall\ t \in dom(\tau), (\tau \downarrow x)(t) \lesseqgtr f(\tau(t)).$$

A *V-differential inequality for x* is an expression of the form $d(x) \lesseqgtr f(\mathbf{v})$, where f is a function that maps $val(V)$ to $\mathbb{R}$. A trajectory $\tau \in trajs(V)$ *satisfies* the differential inequality $d(x) \lesseqgtr f(\mathbf{v})$, if $f \circ \tau : dom(\tau) \to \mathbb{R}$ is an integrable function and

$$\forall\ t_1, t_2 \in dom(\tau), t_1 \leq t_2, (\tau \downarrow x)(t_2) - (\tau \downarrow x)(t_1) \lesseqgtr \int_{t_1}^{t_2} f(\tau(s))ds.$$

A *V-DAI* for a set of variables $Z \subseteq V$ is a collection of inequalities each of which is either a V-algebraic inequality or a V-differential inequality for some variable in Z. A trajectory $\tau \in trajs(V)$ *satisfies* a collection F of V-DAIs for Z, if it satisfies each of the inequalities in F. The set of trajectories satisfying a given set of DAIs for Z may be empty.

The right hand sides of differential and algebraic inequalities are functions of the variables and do not contain time explicitly. Thus, the DAIs impose time invariant constraints on how the values of the variables can change.

Lemma 2.6. *Let V be a set of variables, F be a collection of V-DAIs, and τ be a trajectory for V that satisfies F. For any $c > 0$, $\tau + c$ also satisfies F.*

Proof. Let x be a variable in V and $x \lesseqgtr f(\mathbf{x})$ be any V-algebraic equation in F. Since τ satisfies F, for all $t \in dom(\tau)$, $(\tau \downarrow x)(t) \lesseqgtr f(\tau(t))$. As for all $t \in dom(\tau)$, $\tau(t) = (\tau{+}c)(t{+}c)$, it follows that $((\tau + c) \downarrow x)(t + c) \lesseqgtr f((\tau + c)(t + c))$. That is, $\tau + c$ satisfies all V-algebraic equations in F.

Next, suppose $d(x) \lesseqgtr f(\mathbf{x})$ is any V-differential equation in F. Let us fix $t_1, t_2 \in dom(\tau{+}c)$. We know that $((\tau{+}c) \downarrow x)(t_2) - ((\tau{+}c) \downarrow x)(t_1) = (\tau \downarrow x)(t_2{-}c) - (\tau \downarrow x)(t_1{-}c)$. Since τ satisfies F, we obtain $((\tau + c) \downarrow x)(t_2) - ((\tau + c) \downarrow x)(t_1) \lesseqgtr \int_{t_1-c}^{t_2-c} f(\tau(s))ds$,

$$
\begin{aligned}
&\lesseqgtr \int_{t_1-c}^{t_2-c} f((\tau + c)(s + c))ds \quad [\text{since } \tau(s) = (\tau + c)(s + c) \text{ for all } s]. \\
&\lesseqgtr \int_{t_1}^{t_2} f((\tau + c)(u))du \quad [\text{by change of variable, } u = s + c].
\end{aligned}
$$

∎

A state model combines a collection of DAIs with an invariant condition and a stopping condition. The invariant condition is used to define a subset of the state space over which the DAIs are active in governing continuous evolution. Each state model can be thought of as a mode of the hybrid system. The stopping condition, on the other hand, defines the states at which the trajectories defined by the DAIs must stop, and this in turn forces transitions to occur.

Definition 2.10. A *state model* $\mathcal{S}$ is a 6-tuple $(X, Y, U, F, Inv, Stop)$, where X, Y, U are disjoint sets of variables called *internal, output,* and *input* variables, respectively; $V \triangleq X \cup Y \cup U$ and $Z \triangleq X \cup Y$; F is a collection of V-DAIs for the continuous variables in Z; Inv and $Stop$ are subsets of $val(X)$. If Inv and $Stop$ are specified as predicates on X then they are called the *invariant condition* and the *stopping condition* of $\mathcal{S}$.

Notations. We denote the components of a state model $\mathcal{S}$ by $X_{\mathcal{S}}, Y_{\mathcal{S}}, U_{\mathcal{S}}, F_{\mathcal{S}}, Inv_{\mathcal{S}}$, and $Stop_{\mathcal{S}}$, and the components of a state model $\mathcal{S}_i$ by $X_i, Y_i, U_i, F_i, Inv_i$, and $Stop_i$.

Definition 2.11. Given a state model $\mathcal{S} = (X, Y, U, F, Inv, Stop)$ $trajs(\mathcal{S})$ is the set of trajectories for $X \cup Y \cup U$ defined as follows: a trajectory τ is in $trajs(\mathcal{S})$ if and only if:

1. The discrete variables in X remain constant over τ.

2. τ satisfies all the DAIs in F.

3. At every point in time $t \in dom(\tau)$, $(\tau \downarrow X)(t) \in Inv$.

4. If $(\tau \downarrow X)(t) \in \mathit{Stop}$ for some $t \in \mathit{dom}(\tau)$, then τ is closed and $t = \tau.\mathit{ltime}$.

Example 2.6. The trajectories for the Vehicle HA of Example 2.3 can be described by the two state models shown in Table 2.6. For example, a trajectory $\tau \in \mathit{trajs}(\text{braking})$ if:

	braking	not_braking
X	x_1, x_2, x_3, b	x_1, x_2, x_3, b
Y	y	y
U	–	–
F	$-a_{max} \leq x_3 \leq -a_{min}$ $d(x_2) = x_3, d(x_1) = x_2$ $y = x_1$	$x_3 = 0$ $d(x_2) = x_3, d(x_2) = x_1$ $y = x_1$
Inv	b	$\neg b$
Stop	$false$	$false$

Table 2.1: State models specifying trajectories of Vehicle.

(i) For all $t \in \mathit{dom}(\tau)$

1. $(\tau \downarrow b)(t) = \mathit{true}$,
2. $(\tau \downarrow x_1)(t) = (\tau \downarrow y)(t)$, and
3. $-a_{max} \leq (\tau \downarrow x_3)(t) \leq -a_{min}$.

(ii) $\tau \downarrow x_3 : \mathit{dom}(\tau) \to \mathbb{R}$ is an integrable function.

(iii) For all $t_1, t_2 \in \mathit{dom}(\tau)$ and $i \in \{1, 2\}, (\tau \downarrow x_i)(t_2) - (\tau \downarrow x_i)(t_1) = \int_{t_1}^{t_2} (\tau \downarrow x_{i+1})(s) ds$.

Associating an invariant set with each set F of DAIs allow us to avoid parts of the state space, for example singularities, where the equations in F may not be well formed. The stopping conditions provide a mechanism to define trajectories which satisfy certain conditions at all points in time, except possibly at the right end-point. Suppose we wish to model the positions of two vehicles represented by variables x and y. The vehicles move along a single track according to the DAIs F_1 *until* they collide. After a collision occurs, their dynamics is specified by DAIs F_2. Now, if we associate $x \neq y$ as an invariant for the first state model $\mathcal{S}_1$ then $\mathit{trajs}(\mathcal{S}_1)$ will not contain the trajectories which lead to collisions. However, if we associate $x = y$ as a stopping condition for $\mathcal{S}_1$ then $\mathit{trajs}(\mathcal{S}_1)$ may contain trajectories which satisfy $x = y$ at their right endpoint.

In order to capture that F_2 (and not F_1) describes the dynamics after a collision, following the modeling style of [DL97], we add an action that is enabled when $x = y$ and that it sets a boolean variable *collided* to *true*. Then we add *collided* and $\neg$*collided* as invariant conditions for $\mathcal{S}_2$ and $\mathcal{S}_1$, respectively.

Next, we show that the sets of trajectories specified by state models satisfy **T1**, **T2**, and **T3**.

Proposition 2.7. Consider a state model $\mathcal{S}$.

(i) If $\tau \in trajs(\mathcal{S})$ then any prefix and any suffix of τ is also in $trajs(\mathcal{S})$.

(ii) If $\tau_0, \tau_1 \in trajs(\mathcal{S})$, τ_0 is closed and $\tau_0.lval \lceil X = \tau_1.fval \lceil X$, then $\tau_0 \frown \tau_1 \in trajs(\mathcal{S})$.

Proof. Let $\mathcal{S} = (X, Y, U, F, Inv, Stop)$ and τ_1 be a trajectory in $trajs(\mathcal{S})$. For Part (i), it is routine to check that any prefix (and suffix) of τ_1 satisfies all the four conditions in Definition 2.11, and therefore is also in the set $trajs(\mathcal{S})$.

For Part(ii), we show that $\tau_2 \triangleq \tau_0 \frown \tau_1$ is in $traj(\mathcal{S})$ by checking that it satisfies the four conditions in Definition 2.11.

1. Let x be a discrete variable in X. From Part 1 of Definition 2.11 we know that x remains constant over τ_0 and τ_1. Since $\tau_0.lval \lceil X = \tau_1.fval \lceil X$, it follows that x remains constant over τ_2.

2. Let x be a continuous variable in $X \cup Y$ and f be a function over the values of the variables in $X \cup Y \cup U$. Since τ_0 and τ_1 satisfy every algebraic inequality in F, of the form $x \lesseqgtr f(\mathbf{v})$, so does τ_2. To show that τ_2 satisfies any differential inequality of the form $d(x) \lesseqgtr f(\mathbf{v})$ in F, consider $t_0, t_1 \in dom(\tau_2)$, $t_1 \geq t_0$. The interesting case arises when $t_0 < \tau_0.ltime$ and $t_1 > \tau_0.ltime$. Let $t_2 = \tau_0.ltime$. We can write $(\tau_2 \downarrow x)(t_1) - (\tau_2 \downarrow x)(t_0)$ as

$$\begin{aligned} & (\tau_2 \downarrow x)(t_1) - (\tau_2 \downarrow x)(t_2) + (\tau_2 \downarrow x)(t_2) - (\tau_2 \downarrow x)(t_0) \\ = \; & ((\tau_1 + t_2) \downarrow x)(t_1) - ((\tau_1 + t_2) \downarrow x)(t_2) + (\tau_0 \downarrow x)(t_2) - (\tau_0 \downarrow x)(t_0) \\ \lesseqgtr \; & \int_{t_2}^{t_1} f((\tau_1 + t_2)(s))ds + \int_{t_0}^{t_2} f(\tau_0(s))ds \\ \lesseqgtr \; & \int_{t_2}^{t_1} f(\tau_2(s))ds + \int_{t_0}^{t_2} f(\tau_2(s))ds = \int_{t_0}^{t_1} f(\tau_2(s))ds. \end{aligned}$$

In rewriting the second expression we have used the fact that τ_0 and $\tau_1 + t_2$ satisfy the differential inequality $d(x) \lesseqgtr f(\mathbf{v})$—the latter follows from Lemma 2.6. The last step follows from the fact that, for all $s \in [t_2, t_1]$, $(\tau_1 + t_2)(s) = \tau_2(s)$.

3. Immediate from definition of τ_2.

4. We consider two cases here. If $\tau_0.lval \lceil X \in Stop$, then $\tau_1.fval \lceil X \in Stop$ and τ_1 is a point trajectory, that is, $\tau_2 = \tau_0$. Otherwise, from Part 4 of Definition 2.11 it follows that for every $t \in dom(\tau_2)$ except possibly $\tau_1.ltime$, $\tau_2(t) \lceil X \notin Stop$.

∎

2.4.2 Definition of Structured HIOA

A *Structured Hybrid I/O Automaton (SHIOA)* is a state machine model that uses a collection of state models for specifying its trajectories.

Definition 2.12. A *Structured Hybrid I/O Automaton (SHIOA)* $\mathcal{A}$ is a tuple $(X, Y, U, Q, \Theta, H, O, I, \mathcal{D}, \mathscr{S})$ where

(a) X, Y and U are disjoint sets of *internal, output* and *input* variables. The set $V \triangleq X \cup Y \cup U$ is the set of *variables*, and $Z \triangleq X \cup Y$ is the set of *local variables*.

(b) A set $Q \subseteq val(X)$ of *states* and a non-empty subset $\Theta \subseteq Q$ of *start states*.

(c) H, O and I are disjoint sets of *internal, output* and *input* actions. The set $A \triangleq H \cup I \cup O$ is the set of *actions*, and $L \triangleq H \cup O$ is the set of local actions.

(d) A set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions*.

(e) A collection $\mathscr{S}$ of *state models* such that for each state model $\mathcal{S} \in \mathscr{S}$, $X_{\mathcal{S}} = X$, $Y_{\mathcal{S}} = Y$, and $U_{\mathcal{S}} = U$, and for every pair $\mathcal{S}, \mathcal{S}' \in \mathscr{S}$, $Inv_{\mathcal{S}} \cap Inv_{\mathcal{S}'} = \emptyset$.

In addition, $\mathcal{A}$ satisfies the following axioms:

E1 *(Input action enabled)* For every input action $a \in I$ and for every state $\mathbf{x} \in Q$, there exists a state $\mathbf{x}' \in Q$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

S1 *(Complete)* $Q \subseteq \bigcup_{\mathcal{S} \in \mathscr{S}} Inv_{\mathcal{S}}$.

S2 *(Input trajectory enabled)* For every trajectory $\upsilon \in trajs(U)$ of the input variables, for every state model $\mathcal{S} \in \mathscr{S}$ and every state $\mathbf{x} \in Inv_{\mathcal{S}}$, there exists a trajectory $\tau \in trajs(\mathcal{S})$ with $\tau.fstate = \mathbf{x}$, $\tau \downarrow U \leq \upsilon$, and either (a) $\tau \downarrow U = \upsilon$, or (b) τ is closed and some local action in L is enabled at $\tau.lstate$.

E1 is the same action nonblocking axiom as in the the case of HIOAs. The **S1** axiom ensures that the disjoint invariant sets of the state models together cover the state space Q. Axiom **S2** is a non-blocking axiom for individual state models: given any trajectory υ of the input variables and any state model, either time can elapse for the entire duration of υ, or time elapses to a point at which some local action of $\mathcal{A}$ is enabled. An SHIOA which does not necessarily satisfy **S2** is called a *pre-SHIOA*.

Notations. We denote the components of a pre-SHIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Y_{\mathcal{A}}, U_{\mathcal{A}}, Q_{\mathcal{A}}, \mathscr{S}_{\mathcal{A}}$ etc., and the components of an pre-SHIOA $\mathcal{A}_i$ by $X_i, Y_i, U_i, Q_i, \mathscr{S}_i$, etc. Given a pre-SHIOA $\mathcal{A}$ we define $\texttt{HIOA}(\mathcal{A})$ to be the structure $(X_{\mathcal{A}}, Y_{\mathcal{A}}, U_{\mathcal{A}}, Q_{\mathcal{A}}, \Theta_{\mathcal{A}}, H_{\mathcal{A}}, O_{\mathcal{A}}, I_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \mathcal{T})$, where $\mathcal{T} = \bigcup_{\mathcal{S} \in \mathscr{S}_{\mathcal{A}}} trajs(\mathcal{S})$. Note that $\texttt{HIOA}(\mathcal{A})$ may not necessarily be a HIOA. The next two results state that by augmenting the variables, actions, and discrete transitions with a set of state models, what we obtain is either a HIOA or a pre-HIOA depending on whether on not the state models satisfy axioms **S1** and **S2**.

Lemma 2.8. *If $\mathcal{A}$ is a pre-SHIOA then* $\texttt{HIOA}(\mathcal{A})$ *is a pre-HIOA.*

Proof. From Proposition 2.7 we know that for each state model $\mathcal{S}$ of $\mathcal{A}$, $trajs(\mathcal{S})$ satisfies axioms **T1**, **T2**, and **T3** of Definition 2.3. Therefore, the set $\mathcal{T}$ of trajectories of $\texttt{HIOA}(\mathcal{A})$, defined as $\bigcup_{\mathcal{S} \in \mathscr{S}_{\mathcal{A}}} trajs(\mathcal{S})$ satisfies **T1** and **T2**. Consider $\tau_1, \tau_2 \in \mathcal{T}$, such that τ_1 is closed and $\tau_1.lval \lceil X = \tau_2.fval \lceil X$. Suppose $\tau_1 \in trajs(\mathcal{S})$ for some state model $\mathcal{S}$ of $\mathcal{A}$. Then $\tau_1.lstate \in Inv_{\mathcal{S}}$ and as $\tau_1.lstate = \tau_2.fstate$, $\tau_2.fstate \in Inv_{\mathcal{S}}$. Since the invariant conditions for the state models of $\mathcal{A}$ are disjoint, it follows that $\tau_2 \in trajs(\mathcal{S})$, and therefore $\tau_1 \frown \tau_2 \in trajs(\mathcal{S}) \subseteq \mathcal{T}$. Thus, $\mathcal{T}$ also satisfies **T3**. Since $\mathcal{A}$ satisfies axiom **E1** and the sets of discrete transitions of $\mathcal{A}$ and $\texttt{HIOA}(\mathcal{A})$ are identical, $\texttt{HIOA}(\mathcal{A})$ also satisfies **E1**. ■

Theorem 2.9. *If $\mathcal{A}$ is an SHIOA then* $\texttt{HIOA}(\mathcal{A})$ *is a HIOA.*

Proof. From Lemma 2.8 we know that $\texttt{HIOA}(\mathcal{A})$ is a pre-HIOA. It remains to check that the set of trajectories $\mathcal{T}$ of $\texttt{HIOA}(\mathcal{A})$ satisfies **E2**. Consider any $\mathbf{x} \in Q$ and any $\upsilon \in trajs(U)$. By **S1**, $\mathbf{x} \in Inv_{\mathcal{S}}$ for some state model $\mathcal{S}$ of $\mathcal{A}$. Therefore, according to **S2**, there exists a trajectory $\tau \in trajs(\mathcal{S})$ with $\tau.fstate = \mathbf{x}$, $\tau \downarrow U \leq \upsilon$, and either $\tau \downarrow U = \upsilon$ or τ is closed and a local action is enabled at $\tau.lstate$ as needed for **E2**. ∎

We conclude this section by defining implementation of pre-SHIOAs. A pair of pre-SHIOAs are comparable if they have the same external interface; that is, if the corresponding pre-HIOAs are comparable. Implementation of pre-SHIOAs is also defined in terms of the implementation of the corresponding pre-HIOAs.

Definition 2.13. Two pre-SHIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if $\texttt{HIOA}(\mathcal{A}_1)$ and $\texttt{HIOA}(\mathcal{A}_2)$ are comparable. If pre-SHIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable then A_1 *implements* $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \leq A_2$, if $\texttt{HIOA}(\mathcal{A}_1)$ implements $\texttt{HIOA}(\mathcal{A}_2)$.

2.4.3 Some Special Classes of SHIOAs

Definition 2.14. Let $\mathcal{A}$ be an SHIOA with collection of state models $\mathscr{S}$. A discrete transition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ of $\mathcal{A}$ is said to be a *mode switch* if for some $i, j \in \mathscr{S}, i \neq j$, $\mathbf{x} \in Inv_i$ and $\mathbf{x}' \in Inv_j$. The set of mode switching transitions of $\mathcal{A}$ is denoted by $\mathcal{M}$.

An initialized SHIOA is one in which every mode switching transition resets the values of the variables, nondeterministically, by choosing the values from a set that is independent of the pre-state.

Definition 2.15. An SHIOA $\mathcal{A}$ is said to be *initialized* if every action $a \in A$ is associated with two sets $R_a, Pre_a \subseteq Q$, such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ is a mode switch if and only if $\mathbf{x} \in Pre_a$ and $\mathbf{x}' \in R_a$. The set R_a is called the *initialization predicate* of action a.

Definition 2.16. An SHIOA is *linear* if the V-DAIs of all its state models are linear and the precondition and the initialization predicates (restricted to the set of continuous variables) are described by linear inequalities. A linear SHIOA is *rectangular*[1] if the differential equations in all the state models have constant right hand sides.

Initialized SHIOAs are suitable for modeling periodic systems and systems with reset timers. Rectangular dynamics is suitable for modeling drifting clocks, motion under constant velocity, fluid-level under constant flow. Initialized-rectangular SHIOAs have received a attention in the verification literature [HKPV95, HKPV98], because computing $\mathsf{Reach}_{\mathcal{A}}$ is decidable for this class when the automaton is represented in some appropriate language.

2.4.4 Composition of SHIOA

Two pre-SHIOAs $\mathcal{A}_i = (X_i, Y_i, U_i, Q_i, \Theta_i, H_i, O_i, I_i, \mathcal{D}_i, \mathscr{S}_i)$, $i \in \{1, 2\}$, are *compatible* if $H_1 \cap A_2 = H_2 \cap A_1 = \emptyset$, $X_1 \cap V_2 = X_2 \cap V_1 = \emptyset$, and $Y_1 \cap Y_2 = O_1 \cap O_2 = \emptyset$. The variables, actions, and transitions of the composed pre-SHIOA $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$ are defined in the same way as the corresponding components of a composed pre-HIOA. The following definition states how the state models of the component automata are combined to obtain state models of $\mathcal{A}$.

[1]This definition of rectangular SHIOA is more general than the commonly used one, as for example in [HK96]. In the latter, for each action a, the precondition Pre_a and the reset map R_a are required to be rectangles.

Definition 2.17. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-SHIOAs then their composition $\mathcal{A}_1||\mathcal{A}_2$ is $(X, Y, U, Q, \Theta, H, O, I, \mathcal{D}, \mathscr{S})$ where

1. The sets $X, Y, U, Q, \Theta, H, O, I$, and $\mathcal{D}$ are defined as in Definition 2.9.
2. Suppose the state models in $\mathscr{S}_1$ and $\mathscr{S}_2$ are indexed by sets I_1 and I_2. For each $i \in I_1$ and $j \in I_2$, the combination of $(X_1, Y_1, U_1, F_i, Inv_i, Stop_i) \in \mathscr{S}_1$ and $(X_2, Y_2, U_2, F_j, Inv_j, Stop_j) \in \mathscr{S}_2$ is the state model $\mathcal{S}_{ij} = (X, Y, U, F, Inv, Stop)$, where
   - F is the collection of all DAIs in F_i and F_j,
   - $Inv = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in Inv_i \wedge \mathbf{x} \lceil X_2 \in Inv_j\}$, and
   - $Stop = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in Stop_i \vee \mathbf{x} \lceil X_2 \in Stop_j\}$.

   The set of state models $\mathscr{S}$ for $\mathcal{A}_1||\mathcal{A}_2$ is the collection $\{\mathcal{S}_{ij}\}$.

For the reasons described in Section 2.3.1, the class of SHIOAs is not closed under composition, but the class of pre-SHIOAs is:

Theorem 2.10. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-SHIOAs then $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ is a pre-SHIOA.*

Proof. Let $\mathcal{A}_i = (X_i, Y_i, U_i, Q_i, \Theta_i, H_i, O_i, I_i, \mathcal{D}_i, \mathscr{S}_i)$, for $i \in \{1, 2\}$. Since the variables, actions, and transitions of $\mathcal{A}$ are defined according to Definition 2.9, $\mathcal{A}$ satisfies **E1**. Thus, it suffices to check that $\mathscr{S}_\mathcal{A}$ is a collection of state models for $\mathcal{A}$, with disjoint invariant sets and that $\mathscr{S}_\mathcal{A}$ satisfies the completeness condition **S1**.

Let us fix a state model $\mathcal{S}_i = (X_1, Y_1, U_1, F_i, Inv_i, Stop_i)$ from $\mathscr{S}_1$ and another state model $\mathcal{S}_j = (X_2, Y_2, U_2, F_j, Inv_j, Stop_j)$ from $\mathscr{S}_2$, and investigate their combination $\mathcal{S}_{ij} = (X, Y, U, F', Inv', Stop')$. F_i and F_j are collections of DAIs for the continuous variables in $X_1 \cup Y_1$ and $X_2 \cup Y_2$, respectively. Therefore, F is a collection of DAIs for the set of continuous variables in $X_1 \cup Y_1 \cup X_2 \cup Y_2$, which by Definition 2.17 is the same as the set of continuous variables in $Z_\mathcal{A}$.

Consider a different state model $\mathcal{S}_{kl}$ of $\mathcal{A}$ obtained by combining $\mathcal{S}_k \in \mathscr{S}_1$ with $\mathcal{S}_l \in \mathscr{S}_2$. Since $\mathcal{S}_{ij}$ is different from $\mathcal{S}_{kl}$, we assume without loss of generality that $j \neq l$. For any $\mathbf{x} \in Inv_{\mathcal{S}_{ij}}$, by Definition 2.17, $\mathbf{x} \lceil X_2 \in Inv_j$. Since $Inv_j \cap Inv_l = \emptyset$, $\mathbf{x} \lceil X_2 \notin Inv_l$, and therefore $\mathbf{x} \notin Inv_{\mathcal{S}_{kl}}$. Thus, any two state models of $\mathcal{A}$ have disjoint invariant sets.

For the completeness condition, consider a state $\mathbf{x} \in Q_\mathcal{A}$. Since $\mathbf{x} \lceil X_1 \in Q_1$, from the completeness of $\mathscr{S}_1$ we know that there exists $\mathcal{S}_i \in \mathscr{S}_1$, such that $\mathbf{x} \lceil X_1 \in Inv(\mathcal{S}_i)$. Likewise, from completeness of $\mathscr{S}_2$, there exists $\mathcal{S}_j \in \mathscr{S}_2$, such that $\mathbf{x} \lceil X_2 \in Inv(\mathcal{S}_j)$. It follows that for any $\mathbf{x} \in Q_\mathcal{A}$, $\mathbf{x} \in Inv(\mathcal{S}_{ij}) \subseteq \bigcup_{\mathcal{S} \in \mathscr{S}_\mathcal{A}} Inv(\mathcal{S})$. ■

Although we shall use pre-SHIOAs for systematically describing the trajectories of hybrid systems, the underlying objects are really HIOAs and pre-HIOAs. We show that the operator `HIOA` and the composition operation interact in the expected way. Formally, `HIOA` is a homomorphism from the set of pre-SHIOAs to the set of pre-HIOAs, which preserves the composition operation within each of these sets.

Theorem 2.11. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-SHIOAs then* $\mathtt{HIOA}(\mathcal{A}_1)||\mathtt{HIOA}(\mathcal{A}_2) = \mathtt{HIOA}(\mathcal{A}_1||\mathcal{A}_2)$.

Proof. Let $\mathcal{T}$ be the set of trajectories of $\texttt{HIOA}(\mathcal{A}_1)||\texttt{HIOA}(\mathcal{A}_2)$ and $\mathcal{T}'$ be the set of trajectories for $\texttt{HIOA}(\mathcal{A}_1||\mathcal{A}_2)$. It suffices to show that $\mathcal{T} = \mathcal{T}'$. Let $\mathscr{S}$ be the collection of state models for the composed pre-SHIOA $\mathcal{A}_1||\mathcal{A}_2$; that is, $\mathcal{T}' \triangleq \bigcup_{\mathcal{S}\in\mathscr{S}} trajs(\mathcal{S})$.

First, we show that $\mathcal{T} \subseteq \mathcal{T}'$. Consider a trajectory $\tau \in \mathcal{T}$. From Definition 2.9 we know that $\tau \downarrow V_1$ is in $\mathcal{T}_1$, where $\mathcal{T}_1 = \bigcup_{\mathcal{S}\in\mathscr{S}_1} trajs(\mathcal{S})$ is the set of trajectories of $\texttt{HIOA}(\mathcal{A}_1)$. That is, there exists $\mathcal{S}_1 \in \mathscr{S}_1$, such that $\mathcal{T}_1 \downarrow V_1 \in trajs(\mathcal{S}_1)$. Likewise, $\tau \downarrow V_2 \in \mathcal{T}_2 = \bigcup_{\mathcal{S}\in\mathscr{S}_2} trajs(\mathcal{S})$ and there exists $\mathcal{S}_2 \in \mathscr{S}_2$, such that $\mathcal{T}_2 \downarrow V_2 \in trajs(\mathcal{S}_2)$. Then, $\tau \in trajs(\mathcal{S}_{12})$, where $\mathcal{S}_{12}$ is the state model for the composed pre-SHIOA $\mathcal{A}_1||\mathcal{A}_2$ obtained by combining $\mathcal{S}_1$ and $\mathcal{S}_2$. Thus, $\tau \in trajs(\mathcal{S}_{12}) \subseteq \mathcal{T}'$.

Next, we show that $\mathcal{T}' \subseteq \mathcal{T}$. Consider a trajectory $\tau \in \mathcal{T}' = \bigcup_{\mathcal{S}\in\mathscr{S}} trajs(\mathcal{S})$. There exists $\mathcal{S}_{ij} \in \mathscr{S}$, such that $\tau \in trajs(\mathcal{S}_{ij})$, where $\mathcal{S}_{ij} \in \mathscr{S}$ obtained by combining some $\mathcal{S}_i \in \mathscr{S}_1$ and $\mathcal{S}_j \in \mathscr{S}_2$. Thus, $\tau \downarrow V_1 \in trajs(\mathcal{S}_i) \subseteq \mathcal{T}_1$ and $\tau \downarrow V_2 \in trajs(\mathcal{S}_j) \subseteq \mathcal{T}_2$, and it follows that τ is in $\mathcal{T}$. ∎

Corollary 2.12. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible SHIOAs and their composition $\mathcal{A}_1||\mathcal{A}_2$ is an SHIOA, then $\texttt{HIOA}(\mathcal{A}_1)||\texttt{HIOA}(\mathcal{A}_2)$ is a HIOA.*

Proof. From Theorem 2.11 we know $\texttt{HIOA}(\mathcal{A}_1||\mathcal{A}_2) = \texttt{HIOA}(\mathcal{A}_1)||\texttt{HIOA}(\mathcal{A}_2)$. Since $\mathcal{A}_1||\mathcal{A}_2$ is an SHIOA, from Theorem 2.9 it follows that $\texttt{HIOA}(\mathcal{A}_1||\mathcal{A}_2)$ and therefore $\texttt{HIOA}(\mathcal{A}_1)||\texttt{HIOA}(\mathcal{A}_2)$, is also a HIOA. ∎

2.4.5 Summary

We presented the basic definitions and results that underlie the Structured Hybrid I/O Automaton (SHIOA) framework. Much the the foundational concepts came from the Hybrid I/O Automaton (HIOA) framework of [LSV03]. The SHIOA incorporate state models in the HIOA framework for the purpose of systematically describing trajectories. The idea of state models, as in SHIOAs, is related to the similar notion of state-space description of continuous time systems. However, in addition to the differential and algebraic equations that describe the evolution of the variables in such state-space models, the state models may also contain invariants and stopping conditions. These additional features capture the essence of hybrid systems where discrete transitions bring changes in the state variables that in turn change the state model that govern further evolution. In the next chapter we develop the HIOA language for specifying SHIOAs; the design of this language relies heavily on the state model structure. In Chapters 4, 5, and 6 we develop tools and techniques for verifying SHIOAs; these are also aided by the state model structure.

Chapter 3

The HIOA Language

HIOA is a language for succinctly and precisely specifying hybrid systems. The semantics of the language is based on the Structured Hybrid I/O Automata (SHIOA) model introduced in Chapter 2. This language is used throughout Part I for describing hybrid systems. The HIOA language evolved in part from the IOA language [GLTV03] which has been widely used for specifying untimed distributed systems. The new features of the HIOA language, namely the constructs for continuous variables and trajectories, are based on the work presented in [MWLF03, Mit01, KLMG05]. A somewhat restricted version of HIOA, which does not allow external variables, is implemented as part of the Tempo Toolkit [TEM07]. In Chapter 6 we will show how HIOA specifications can be translated to the language of the PVS Theorem prover [ORR+96]. In Section 7.7 of Part II, HIOA is extended to allow specification of probabilistic transitions. In this Chapter, we describe the key features of the language through a sequence of small examples.

3.1 An Overview

Consider a SHIOA Bounce describing the position and the velocity of a ball bouncing on a surface. The HIOA language specification for Bounce is shown in Figure 3-1. The first line consists of the keyword **automaton** followed by the automaton's name Bounce, and a list of *formal parameters*. Formal parameters are used for succinctly specifying sets of objects such as automata, actions, trajectories, etc. In this example, the coefficient of restitution ρ of the surface is a real-valued formal parameter of Bounce. The value of ρ is restricted to be strictly between 0 and 1 using a *where-clause*. Where-clauses are predicates that constrain values of formal parameters and therefore are useful for avoiding ill-formed specifications, such as those resulting from division-by-zero. Usage of formal parameters and where-clauses are further described in Section 3.2.4.

The body of the specification of Bounce has four sections: (a) signature, (b) variables, (c) transitions, and (d) trajectories. The **signature** section declares the names and *kinds* of actions of the automaton. Bounce has a single output action called bounce. The other kinds of actions that an automaton can have are input and internal. Signature definitions are discussed in further detail in Section 3.4.

The variables of an automaton are declared by the keyword **variables** followed by list of variables, their kinds, types, dynamic types, and possibly their initial values. Bounce has a real-valued internal variable v with initial value 0, and a real-valued output variable x with initial value 100. In HIOA all real valued variables are implicitly continuous. In

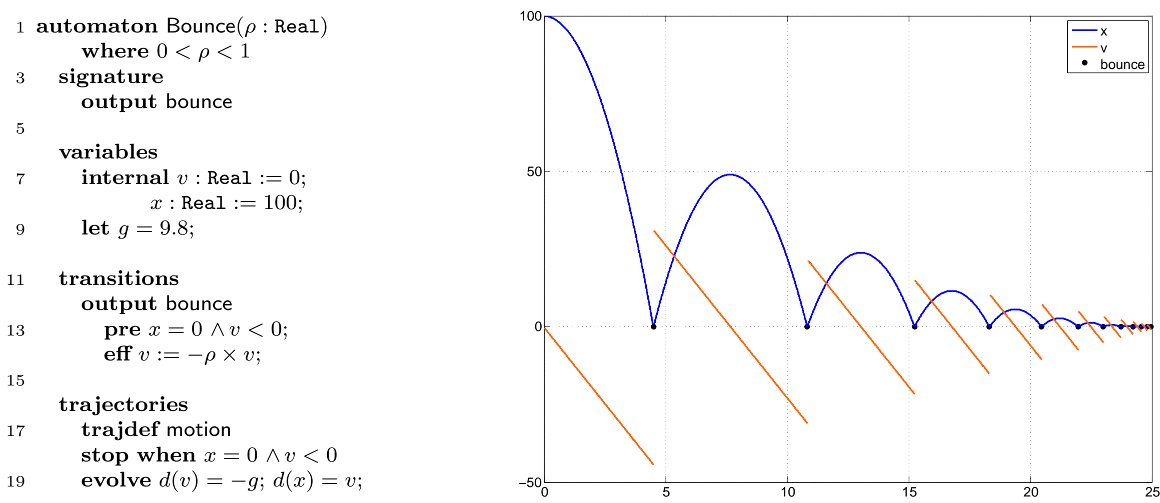

Figure 3-1: Bouncing ball: HIOA specification and an execution.

Section 3.2, specification of variables, types, and dynamic types are discussed in detail. Constant functions such as g can be defined using the **let** keyword. Other types of function declarations and their scopes are discussed in Section 3.3.

The **transitions** section defines the discrete transitions corresponding to each action of the automaton. Transitions are defined in the guarded-program style. The predicate following the **pre** keyword defines the set of states where the action is enabled, and the *program* following **eff** defines how the state changes when the action occurs. For example, the bounce action *can* occur in a state where $x = 0$ and $v < 0$, and when it does occur v is assigned a new value which is $-\rho$ times its previous value. Constructs for specifying preconditions and programs are discussed in Section 3.5.

Finally, the **trajectories** section of the specification defines a set of state models. Each state model definition begins with the keyword **trajdef** followed by the name of the state model, an invariant condition, a stopping condition, and a set of Differential and Algebraic Inequalities (DAIs). Bounce has a single state model called motion. The list of expressions following the keyword **evolve** define the relationship between x, v, and their derivatives with respect to real-time. In Section 2.4.1 we defined how a state model defines a set of trajectories for a SHIOA. We discuss the **trajdef** construct in Section 3.6.

A few observations about the specification of Bounce: The combination of the stopping condition $x = 0 \wedge v < 0$ of the state model motion and the matching precondition for bounce, forces the action to occur following a trajectory whose last state satisfies this condition. This usage pattern for stopping conditions and action preconditions is common in HIOA specifications where actions have to be triggered as soon as certain deadlines are met.

3.2 Variables

In HIOA, variables are used in two different ways. First, they are used as variables of automata (always declared following the **variables** keyword). Secondly, variables are used as formal parameters in defining automata, actions, functions, trajectories, etc. In order to avoid ambiguity, we will refer to variables from the first category as *automaton variables* and those from the second as *formal parameters*.

Each automaton variable is of one of three kinds, namely, input, output, or internal. The kind of a variable is specified by the corresponding keyword preceding its name.

3.2.1 Built-in Types

All variables are associated with types. There are several built-in types, such as, `Bool`, `Char`, `Int`, `Nat`, `Real`, `AugmentedReal`, and `String` that do not require explicit definitions. The `AugmentedReal` type is $\texttt{Real} \cup \{\infty, -\infty\}$. In modeling embedded systems as SHIOAs, often we have to define a real-time upper-bound within which an action has to occur. As the automaton executes, this upper-bound may change. A common idiom in HIOA is to model such bounds by the value of a *deadline variable* (see, Example 3.4). `AugmentedReal` is a useful type for such a deadline variable because setting the variable to ∞ implies that it does not impose any deadline.

Additional types can be defined using the following type constructors:

- `Enumeration` $[e_1, e_2, e_3]$: finite set with elements e_1, e_2, and e_3.
- `Null` $[E]$: `E` extended by a single element `nil`.
- `Tuple` $[I_1 : E_1, \ldots I_n : E_n]$: n-tuple with fields $I_1, \ldots, I_n$ of types, $E_1, \ldots E_n$.
- `Union` $[I_1 : E_1, \ldots I_n : E_n]$: union of types $E_1, \ldots E_n$ with accessors $I_1, \ldots, I_n$.
- `Seq` $[E]$: finite sequence of elements of type E.
- `Set` $[E]$: finite set of elements of type E.
- `Mset` $[E]$: finite multiset of elements of type E.
- `Map` $[D_1, \ldots, D_n, E]$: finite (possibly partial) mapping of $D_1 \times \ldots \times D_n$ to E.
- `Array` $[I_1, \ldots, I_n, E]$: n-dimensional array of type E indexed by element of types $I_1, \ldots, I_n$ where the indices are of finite types.

Type constructors can be nested and can be used in-place to declare types of formal and automaton variables. For example, the code fragment

```
variables
  internal buffer: Seq [Bool],
  output position: Tuple [x : Real, y : Real, z : Real]
```

defines an internal variable *buffer* which takes values in $\{0, 1\}^*$, and an output variable *position* which takes values in $\mathbb{R}^3$. A restricted version of the HIOA language, which does not allow input/output variables, is called the TIOA language. Current implementation of the TIOA language is available as a part of the Tempo toolset [TEM07] In this implementation, type equivalences are decided based on types names, and hence, two types with identical definitions but with different names are judged to be distinct.

3.2.2 Vocabularies

Named types are defined using the **vocabularies** construct. Vocabularies can define concrete as well as uninterpreted types, and also operators on these types. The HIOA language allows polymorphic types and operators, using which one can specify automata that manipulate objects of polymorphic types.

In Figure 3-2 a vocabulary for directed graphs is defined and used in the updateGraph automaton. Vocabulary definitions consist of the keyword **vocabulary**, followed by the name of the vocabulary, followed by an optional list of formal parameters. The vocabulary *directedGraphs* has a single formal *type* parameter T. Type parameters are special kinds of formal parameters which denote a type. They are declared using the keyword **type** and they provide a mechanism for polymorphic HIOA specifications. In this example, the

type parameter T specifies the type of the vertices of the directed graphs defined by the *directedGraphs* vocabulary. The main body of any vocabulary definition has two sections: (a) types and (b) operators. The **types** section introduces names and definitions for types. The *directedGraphs* vocabulary introduces the types *Edge*, *Digraph*, and *Path*. The type *Edge* is defined as an ordered pair of elements of type T. *Digraph* is defined as an ordered pair of sets—the first component *vset* is a set of type T and the second component *eset* is a set of *Edge*s. The type *Path* is defined as a finite sequence of elements of type T. The **operators** section introduces a set of operator names and their signatures. For example, *directedGraphs* has two operators *connected* and *addEdge*. The *connected* operator takes a pair of elements of type T and returns a boolean. The *addEdge* operator takes a *Digraph* and an *Edge* and returns a *Digraph*.

If *directedGraphs*(`Nat`) is imported into an automaton, then all the type and operator definitions in *directedGraphs* are interpreted with T replaced by `Nat`. For example, the initial graph G would be an arbitrary graph with a set of natural numbers as vertices. The automaton updateGraph of Figure 3-2 imports *directedGraphs*(V), where V is a formal type parameter of the automaton. This illustrates how we can specify an automaton that performs operations on generic graphs without ever instantiating the type of the graph vertices.

Of course, for proving or model-checking properties of the automaton, for simulating it, or for generating executable code from the HIOA specification, it may become necessary to assert additional properties or to provide implementations of the types and operations. For example, these may take the form of axioms stating key properties of the *connected* function for a theorem prover, or a Java implementation of the *Digraph* data-type for a code generator. These are provided as appropriate auxiliary files to the back-end tools independent of the HIOA language. In Chapter 6 we discuss this in more detail in the context of the PVS theorem prover.

```
1   vocabulary directedGraphs(T :type)
      types
3       Edge = Tuple [src : T, dst : T];
        Digraph =
5         Tuple [vset :Set[T], eset :Set[Edge]];
        Path = Seq [T];
7
      operators
9       connected : T, T → Bool;
        addEdge : Digraph, Edge → Digraph;
11  end

                                                   12
    automaton updateGraph(V : type)
    imports directedGraphs(V)                      14

      signature                                    16
        input add(e : Edge)
                                                   18
      variables
        internal G : Digraph;                      20

      transitions                                  22
        input add(e)
          eff G := addEdge(G, e);                  24
```

Figure 3-2: Vocabulary for graphs.

3.2.3 Dynamic Types

All automaton variables are associated with dynamic types. In HIOA, dynamic types can be either discrete or continuous (see, Definition 2.2). The dynamic type of a variable is declared implicitly; it is inferred automatically from the variable's static type. For variables of all built-in simple types except `Real` the dynamic type is discrete. The dynamic type of `Real` variables is continuous. For defining a `Real` variable with discrete dynamic type, its type name is qualified with the keyword `Discrete`.

If the type of a variable v is defined by one of the type constructors `Tuple` or `Array`, then $dtype(v)$ is defined as follows. The variable v is viewed as a ordered tuple of variables $\{v_1, \ldots, v_k\}$, for some finite k. The type of v, $type(v) = type(v_1) \times \ldots \times type(v_k)$. The dynamic type of v is the set of functions f from left-closed intervals of time to $type(v)$ such that $f.v_i \in dtype(v_i)$ for each $i \in \{1, \ldots, k\}$.

If the type of v is defined by nesting the type constructors `Tuple` and `Array`, then the $dtype(v)$ is defined recursively using the above rule. The dynamic type of all other compound types and user-defined types is discrete. Consider the following vocabulary definition:

```
vocabulary matrix
  types
    T: Tuple [a : Real, b : Nat, c : Discrete Real],
    Row: Enumeration [p1, p2, p3],
    Col: Enumeration [q1, q2, q3],
    matrix: Array [Row, Col, Real],
    intMatrix: Array [Row, Col, Int]
end
```

Suppose that variable v is declared to be of type T. Then $dtype(v)$ is the set of functions f from left-closed intervals of time to T such that $f.a$ is piecewise continuous with real values, $f.b$ is piecewise constant with natural values, and $f.c$ is piecewise constant with real values. The type *matrix* represents a 3×3 array of real numbers. A variable x declared to be of type *matrix* can be viewed as an ordered 9-tuple of reals. The dynamic type of x is the set of functions f from left-closed intervals of time to $\mathbb{R}^9$ such that the restriction of f on each of the coordinates is a piecewise continuous function from left-closed intervals of time to $\mathbb{R}$. A variable y of type *intMatrix* can be viewed as an ordered 9-tuple of integers. And $dtype(y)$ is the set of functions f from left-closed intervals of time to $\mathbb{Z}^9$ such that the restriction of f on each coordinate is piece-wise constant.

3.2.4 Initial Values

Initial valuations of the internal variables of an automaton may be specified in the HIOA language in two ways. First, each variable can be individually initialized using the assignment operator **:=** followed by an expression or a nondeterministic choice over a set. The expression should not refer to any other automaton variable. This method is used in lines 7–8 of the Bounce automaton of Figure 3-1. A nondeterministic choice is specified by the keyword **choose** followed by a set, written as a where-clause. The second method is to use the **initially** keyword followed by a predicate on the internal variables defining the set of allowed initial values. This selects an arbitrary value for the variables from the set of values that satisfy the predicate. It is possible to mix these methods.

```
variables
  output u : Discrete Real
  internal x : Real := 5,
      y : Real := choose k where −5 ≤ k ≤ 5,
      z : Real
      initially z × z + y × y ≤ 10
  ...
```

In the above code fragment, the initial value of x is set to 5, y gets some value in $[-5, 5]$, and z is chosen such that $z^2 + y^2 \leq 10$.

3.3 Functions

Function definitions can occur as stand-alone specification units, following the definition of automaton variables, and within a transition (or a trajectory) definition. When a function definition appears within a transition (or trajectory) definition, it follows the name and the formal parameters of the transition (trajectory, respectively). A function declaration begins with the keyword **let** and consists of the following parts: (a) a name, (b) an optional list of formal parameters, and (c) an expression defining the body of the function. The domain of the function consists of the sequence of types associated with the formal parameters, and the range of the function is the type of the expression defining the body. A function with an empty domain, such as g in line 9 of Figure 3-1, is a *constant function* or simply a constant. Each constant, operator, or variable appearing in the expression defining the body of the function must be defined in the current context, that is, in terms of the vocabulary of the automaton, the set of variables, the list of formal parameters of the function, etc.

```
let dist(x1, x2, y1, y2 : Real) = sqrt((x2 − x1) × (x2 − x1) + (y2 − y1) × (y2 − y1))
automaton Track
signature
  input TargetUpdate(xt, yt : Real)

variables
  internal pos : Tuple[x : Real, y : Real],
      togo : AugmentedReal := ∞,
      let dist_to_target(x, y : Real) = dist(x, pos.x, y, pos.y)

transitions
  input TargetUpdate(xt, yt)
    eff togo := dist_to_target(xt, yt)
```

In the above code fragment, the *pos* represents the position of a vehicle on a plane. The evolution of *pos* is not shown. The vehicle receives information about the location of targets through the input action TargetUpdate(xt, yt) and updates the variable *togo* with the value of its current distance to the most recently detected target (xt, yt). The function *dist* returns the Euclidean distance between any two points $(x1, y1)$ and $(x2, y2)$. It is a global function and does not use any of the automaton variables. The function *dist_to_target* uses *dist* and the state variable *pos* and returns the distance of any point (x, y) to *pos*. This is legal because its definition appears *after* the **variables** section of the code and so the state variable *pos* is in its scope. The action TargetUpdate(xt, yt) updates the variable *togo* with the distance of *pos* to (xt, yt) as computed by *dist_to_target*. Each formal parameter of a function must be distinct from all other formal parameters and all other variables in the scope. That is, they must differ either in their names or their types.

3.4 Signature

The signature section of an automaton specification declares the names and kinds of actions of the automaton. Each action is of one of the three kinds, namely, input, output, or internal. The kind of an action is specified by the corresponding keyword preceding its name. Each action name is followed by an optional list of formal parameters. Each formal parameter of an action must be distinct from all the others. If a formal parameter of an action is the same as an automaton parameter then the former shadows the latter. Automaton parameters can be used as formal action parameters by using the **const** keyword; this is discussed shortly. Each action name denotes a set of actions, one for each value of its parameters. It

is possible to constrain the values of the parameters of an action in the signature using a where-clause. For example, consider the signature of automaton Proc.

```
automaton Proc(I : type, i : I)
  signature
    input add(k, j : Int) where k > 0 ∧ j > 0
    output result(k : Int) where k > 1
    output send(const i)
  ...
```

The automaton is parameterized by a type parameter I and a parameter i of type I. This type of parameterization is often used for specifying a collection of processes indexed by the set I. The signature restricts the values of the input action add's parameters to positive integers and the value of the output action result's parameter to integers greater than 1. Often, we find it necessary to name actions based on automaton parameters. In the above code fragment, for example, we would like to say that for each $i \in I$, $\mathsf{Proc}(I, i)$, has a single action $\mathsf{send}(i)$. The **const** keyword provides a mechanism for achieving this. The **const** preceding the parameter i in the signature of send indicates that the value of this parameter is a constant fixed by the value of the automaton parameter i.

3.5 Transitions

In the transitions section of an automaton specification the discrete transitions corresponding to the actions are specified using the guarded-program style. A transition definition consists of (a) an action kind, (b) an action name, (c) optional formal parameters and an optional where-clause constraining the values of the parameters, (d) optional function definitions, (e) an optional precondition, (f) an optional effect. More than one transition definition can be given for a parameterized action. In such cases, where-clauses can be used to split-up the set of transition definitions according to predicates on formal parameters. The predicates associated with the where clauses are not required to be disjoint although in most common usages they are.

```
signature
  input read(i : Nat)

variables
  internal high : Bool

transitions
  input read(i) where i > 50
    eff high := true

  input read(i) where i ≤ 50
    eff high := false
```

3.5.1 Preconditions

A precondition can be defined for a transition of an output or an internal action using the keyword **pre** followed by a predicate on the automaton parameters, action parameters, and the internal variables. Preconditions cannot be defined for transitions of input actions. This corresponds to the input action enabling condition **E1** of Definition 2.12. If no precondition is given, it is assumed to be *true*. Each variable appearing in the precondition must be one of the following: automaton parameter, internal variable, a parameter in the transition definition, or a variable that is quantified explicitly in the precondition.

3.5.2 Effects

The effect of a transition is defined in terms of a (possibly nondeterministic) program following the keyword **eff**. The program assigns values to the internal variables of the automaton, thus defining the *post-state* (i.e., the state after the transition occurs) in terms of the values of the variables in the *pre-state* (i.e., the state before the transition occurs). If the effect of a transition is omitted, then the state remains unchanged. The program used to specify the relation between the pre-state and the post-state of a transition is a list of statements, separated by semicolons. The statements are executed sequentially and the computation of the whole program is completed in a single atomic step. There are three types of statements: (i) assignments, (ii) conditionals, and (iii) loops, which we describe next.

Assignments

An assignment statement consists of an internal variable of the automaton followed by the assignment operator := and either an expression (see, e.g., line 24 of Figure 3-2) or a nondeterministic choice (as in the case of nondeterministic choice of initial values). The expression or nondeterministic choice in an assignment statement must have the same type as the internal variable. The execution of an assignment does not have side-effects, that is, it does not change the value of any variable other than the internal variable appearing on the left side of the assignment operator.

Conditionals

A conditional statement is used to control the flow of program execution within the effects part of a transition. It starts with the **if** keyword followed by a predicate, the keyword **then**, and a program segment; it ends with a keyword **fi**. In between there can be any number of **elseif** clauses. For example, the conditional statement

if *clock* $\geq$ *Delta* **then** *timeout* := *true* **fi**

assigns a value to the *timeout* variable based on the value of *clock*.

Loops

A loop is used to repeatedly execute a program segment a finite number of times—once for each value of a variable that satisfies a given condition. A loop starts with the keyword **for** followed by a variable, a predicate defining a finite set of values for this variable, and a program segment enclosed within the keywords **do** and **od**. For example, the for-loop

for i : `Nat` **in** S **do** *time_stamp*[i] := *clock* **od**

sets the value of *time_stamp*[i] to *clock* for each natural number i in the set S.

3.6 Trajectories

The trajectories section of a HIOA specification defines a set of state models for the automaton. These state models in turn define a set of trajectories according to Definition 2.11. Each trajectory definition starts with the keyword **trajdef** and has the following components: (a) a name for the state model, (b) optional list of formal parameters and a where-clause constraining the values of the parameters, (c) optional function definitions, (d) an optional

invariant condition, (e) an optional stopping condition, and (f) a collection of Differential and Algebraic Inequalities (DAIs). As in the case of transition definitions, names of trajectory definitions can be parameterized by formal variables and the values of these parameters can be constrained by where-clauses. In the following example, the parameter m is used to specify the motion of a body of mass m.

```
variables
  input P : Real,
  internal f : Real, v : Real, s : Real
...
trajectories
  trajdef motion(m : Real) where m > 0
    evolve f = P/m; d(v) = f; d(s) = v
```

3.6.1 Invariant Condition

An invariant condition for a trajectory definition can be defined using the keyword **invariant** followed by a predicate on the formal parameters and the internal automaton variables. The invariant condition for a trajectory definition, say mode1, defines the set of states over which the DAIs for mode1 govern the evolution of the continuous variables. If no invariant condition is given, it is assumed to be $true$. Each variable appearing in the invariant condition must be one of the following: automaton parameter, internal variable, a parameter in the trajectory definition, or quantified explicitly in the predicate. If an automaton specification contains multiple trajectory definitions, then the corresponding invariant conditions must be disjoint. In order for the automaton specification to satisfy the completeness condition **S1** of Definition 2.12, the union of the sets corresponding to the invariant conditions for all the trajectory definitions should contain the state space of the automaton.

3.6.2 Stopping condition

A stopping condition for a trajectory definition can be defined using the keywords **stop when** followed by a predicate on the automaton parameters, trajdef parameters, and the internal variables. The semantics and motivation for stopping conditions were discussed in Section 2.4.1. If no stopping condition is given, it is assumed to be $false$. Each variable appearing in the stopping condition must be one of the following: automaton parameter, internal variable, a parameter in the trajectory definition, or quantified explicitly in the precondition.

If the stopping condition for a trajectory definition is satisfied at a given point in time t of a trajectory τ, then t is the right end-point of the trajectory. This provides a mechanism for forcing enabled actions to occur.

```
1   automaton PSend(d : Real) where d > 0
      signature
3       output send

5     variables
        internal clock : Real := 0;
7           next_send : DiscreteReal := d;

9     transitions
        output send
11        pre clock = next_send;
          eff next_send := next_send + d;
13
      trajectories
15      trajdef normal
          stop when clock = next_send
17        evolve d(clock) = 1;
```

Figure 3-3: Periodically sending process.

In the example automaton PSend of Figure 3-3, the *next_send* variable is used as a deadline variable to trigger a send action every d time. The variable *clock* increases at the same rate as that of real-time. A send action occurs whenever *clock* equals *next_send*. When the action occurs, the *next_send* variable is incremented by d and the clock continues to increase.

3.6.3 DAIs

The behavior of real-valued, continuous, local (internal and output) variables in a particular trajectory definition is specified in terms of a collection of DAIs following the keyword **evolve**. Discrete local variables remain constant over all trajectories. The DAIs constrain the set of trajectories, and thereby specify how the continuous variables are allowed to evolve over time. The semantics of how DAIs together with invariant and stopping conditions constrain the set of trajectories has been defined in Section 2.10. If no DAI is specified, then the local continuous variables are constrained only by their dynamic types.

HIOA does not impose any specific conditions for well-formedness of DAIs. The set of differential and algebraic equations may have unique, multiple, or no solutions. The DAIs are specified as a list of expressions separated by semicolons. Each expression is of the form `lval` $\lesseqgtr$ `term`, where $\lesseqgtr$ is an element of the set $\{=, \leq, \geq, <, >\}$, and `term` is a real-valued algebraic expression involving the continuous automaton variables and parameters. The `lval` must either be a locally controlled (internal or output) continuous variable or of the form $d(v)$, where v is such a variable. In the latter case the `term` must be integrable.

Note that the rules described above do not guarantee that a semantically correct entity satisfies the input trajectory enabled condition **S2** of 2.12. Therefore, the specified entity is guaranteed to be a pre-SHIOA but not necessarily a SHIOA. Checking existence and uniqueness of solutions of DAIs is difficult and we do not know of any automatic methods for checking these properties in the general setting. Therefore, given a semantically correct specification written in the HIOA language we will have to reason about it separately to show that it satisfies **S2**.

Example 3.1. The automaton Thermostat(u, l, K, h : `Real`) of Figure 3-4 specifies the SHIOA model of a thermostat. A hybrid automaton model of a similar thermostat appeared earlier in [A$^+$95]. Both the variables x and *loc* are internal variables, and both actions switchOn and switchOff are internal actions. The temperature x is governed by standard linear differential equations $d(x) = K(h - x)$ and $d(x) = -Kx$, depending on whether or not the heater is on. These two conditions correspond to the disjoint invariant conditions of the two state models defined by the heaterOn and heaterOff state models. The switchOn action is enabled and triggered exactly when the temperature x drops to l. Note that once the switchOn action occurs, repeated occurrences of the action at the same point in time are prevented by the $loc = \mathit{off}$ conjunct in the precondition.

Example 3.2. The automaton TimedChannel(M : **type**, b : `Real`, I : **type**, i, j : I) of Figure 3-5 specifies a reliable FIFO channel [KLSV05], that delivers messages of type M between processes that are indexed by the type I. The channel delivers its messages from the process i to process j within a time bound of b. The automaton has two internal variables. The variable *now* corresponds to real-time, and *queue* is a finite sequence of *Packet*s. Each *Packet* is a pair—the first element is the message of type M and the second element is a real number corresponding to the delivery deadline for the message. The input action send(m) adds a packet to the *queue* with message m and the upper time bound for the delivery

```
1  automaton Thermostat(u, l, K, h : Real) where u > l
     type Status enumeration [on, off]
3    signature
       internal switchOn; switchOff
5
     variables
7      internal x : Real := u; loc : Status := on;

9    transitions
       internal switchOn
11       pre x = l ∧ loc = off;
         eff loc := on;
13
       internal switchOff
15       pre x = u ∧ loc = on;
         eff loc := off;

     trajectories                                  18
       trajdef heaterOn
         invariant x ≤ u ∧ loc = on;               20
         stop when x = u;
         evolve d(x) = K × (h − x);                22

       trajdef heaterOff                           24
         invariant x ≥ l ∧ loc = off;
         stop when x = l;                          26
         evolve d(x) = −Kx;
```

Figure 3-4: Thermostat.

of the message. The output action receive(m) *can* occur whenever there is a packet with message m at the head of the queue and it *must* occur before *now* exceeds the deadline for the message.

```
1  vocabulary Packet(T :type)
     types
3      Packet = Tuple [message : T,
              deadline : AugmentedReal]
5  end

7  automaton TimedChannel(M :type, b : Real
          I :type, i, j : I) where b ≥ 0
9    imports Packet(M)
     signature
11     input send(m : M, const i, j)
       output receive(m : M, const i, j)
13
     variables
15     internal queue : Seq[Packet] := {};
              now : Real := 0                      16

     transitions                                   18
     input send(m, i, j)
       eff queue := queue ⊢ [m, now + b];          20

     output receive(m, i, j)                       22
       pre head(queue).message = m;
       eff queue :=  tail(queue);                  24

     trajectories                                  26
       trajdef timePassage
         stop when head(queue).deadline = now      28
         evolve d(now) = 1;
```

Figure 3-5: Time-bounded channel.

Example 3.3. The automaton ClockSync$(u, \rho : \texttt{Real}, I : \textbf{type}, i : I)$ of Figure 3-6 specifies processes participating in a distributed clock synchronization algorithm. The physical clock (the variable *clock*) of each process drifts at a bounded rate of ρ, and the processes communicate to ensure that their logical clocks (the variable *logclock*) are close to one another and also close to their physical clocks. This example is taken from [KLSV05], where the authors prove through invariant assertions that the participating processes indeed satisfy the above requirements.

The processes are indexed by elements of type I. The synchronization algorithm is based on the exchange of physical clock values between different processes in the system through TimedChannels. A process uses its real-valued, discrete variable *maxother* to keep track of the largest physical clock value of other processes in the system. For ClockSync(i), the parameter i of the send action is a constant while the parameter j is a variable. Thus ClockSync(i) can send messages to, and receive messages from ClockSync(j), for all j in I, $j \neq i$. The trajectory specification consists of a single trajectory definition timeElapse. This specifies the evolution of the continuous variable *clock* using two differential inclusions that

bound the drift. The stopping condition ensures that the deadline imposed by $nextsend$ is not crossed.

```
1  automaton ClockSync(u, ρ : Real, I :type, i : I)
              where 0 ≤ ρ ≤ 1 ∧ u > 0
3  signature
     output send(m : Real, const i, j : I) where i ≠ j
5    input receive(m : Real, j : I, const i) where i ≠ j

7  variables
     internal clock : Real := 0;
9      nextsend : Discrete Real := 0;
       maxother : Discrete Real := 0;
11   let logclock = max(maxother, clock);

   transitions                                          13
     output send(m, i, j)
       pre m = clock ∧ clock = nextsend;                15
       eff nextsend = nextsend + u;
                                                        17
     input receive(m, j, i)
       eff maxother = max(maxother, m);                 19

   trajectories                                         21
   trajdef timeElapse
     stop when clock = nextsend                         23
     evolve d(clock) ≤ (1 + ρ);
            d(clock) ≥ (1 − ρ);                         25
```

Figure 3-6: Process participating in a clock synchronization algorithm.

3.7 Operations and Properties

In specifying a complex system with many logical pieces, it is convenient to model each piece as a separate SHIOA. Then the whole system is specified as a composition of the component SHIOAs. In section 2.4.4 we defined the composition operation for SHIOAs. Here we describe the the HIOA language constructs for specifying compositions and for asserting properties of automata.

3.7.1 Composition

A composed automaton is specified by the automaton name followed by the keyword **components** and a semicolon-separated list of component names and automaton types. For example, using the TimedChannel and ClockSync automata from the previous section we can define the composed automaton:

```
automaton Sync(u, ρ, b : Real, I :type)
  components
    P[i : I] : ClockSync(u, ρ, I, i);
    C[i, j : I] : TimedChannel(b, I, i, j) where i ≠ j
```

The keyword **components** introduces a list of named components: one ClockSync(u, ρ, I, i) automaton, $P[i]$, for each i of type I, and one TimedChannel(b, I, i, j) automaton $C[i, j]$, for each i, j of type I, such that $i \neq j$. If the input action of one component has the same name as the output action of another component, then they are identified and the composition has the resulting output action. Likewise, the names of input and output variables are identified and the composed automaton has the corresponding output variables.

A parameterized automaton specification defines a set of automata rather than a single automaton. The composition operation can be used to instantiate a particular automaton. For example, the Sync automaton above defines a set of composed automata for different parameter types I. We can fix the type parameter to a particular index set $\{a, b, c, d\}$ as follows:

```
vocabulary types abcd = Enumeration [a, b, c, d] end
automaton Sync4(u, ρ, b : Real)
  components theOnly : Sync(u, ρ, b, abcd)
```

3.7.2 Property Assertions

Along with specifications, properties of automata can also be asserted in HIOA. Such assertions must appear after the automaton specification. An invariant property of an automaton is stated by the keywords **invariant of**, followed by the name of the automaton and a predicate on the variables of the automaton.

invariant of Sync : $\forall\ i, j : I, |P[i].logclock - P[j].logclock| \leq u + b \times (1 + \rho)$

A simulation relation is a relation over the states of a pair of HIOAs and it provides a sound way of proving implementation relations (defined in Section 2.2.2). We shall encounter different kinds of simulation relations and study their properties in Chapters 4 and 5. For now, we note that simulation relations can also be specified in the HIOA language using the **simulation from** keywords. We conclude the presentation of the HIOA language with another example which uses many of its features.

Example 3.4. This failure detector example is taken from [KLSV05]; mechanized proofs of correctness appeared in [ALL+06]. The automaton Spec(d) of Figure 3-7 abstractly specifies the timing related requirements of a simple failure detector. It specifies that a timeout occurs within d time of a fail. Furthermore, a timeout occurs only if it is preceded by a fail. Specifically, the value of $failed$ is initially $false$. Thus, the value of the $clock$ variable increases indefinitely at the same rate as that of real-time. The fail action can occur at any point in time. When it does occur, the variable $failed$ is set to $true$ and the deadline variable $last_timeout$ is set to $clock + d$. This enables the timeout action. The trajectory definition failed becomes active and the $clock$ continues to increase at the same rate, but now it can increase only to the point where it equals $last_timeout$. When the clock reaches this value the timeout action *must* occur. This action sets $suspected$ to $true$, which in turn once again enables time to elapse indefinitely.

```
automaton Spec(d : Real) where d > 0
  signature
    input fail
    output timeout

  variables
    internal last_timeout : AugmentedReal := ∞,
      clock : Real := 0;
      suspected : Bool := false;
      failed : Bool := false;

  transitions
    input fail
      eff if ¬failed
        then last_timeout := clock + d;
          failed := true; fi

  output timeout
        pre failed ∧ ¬suspected;
        eff suspected := true;
          last_timeout := ∞;

  trajectories
    trajdef normal
      invariant ¬failed ∨ (failed ∧ suspected)
      evolve d(clock) = 1;

    trajdef failed
      invariant failed ∧ ¬suspected
      stop when clock = last_t imeout
      evolve d(clock) = 1;
```

Figure 3-7: Failure detector specification.

The automata in Figure 3-8 implement a failure detector using the TimedChannel of Figure 3-5. PeriodicSend($M, u1$) sends a message once every $u1$ time units, until a fail action occurs. This periodic timing is maintained by the $next_send$ deadline variable. Notice how the AugmentedReal type of this variable becomes useful when the fail action occurs.

The messages from PeriodicSend($M, u1$) are transmitted over a channel that connects this process with the Detector process and delivers every sent message within b time units. In the composite specification Timeout of Figure 3-9, a single TimedChannel is instantiated

with the index set $\{1, 2\}$. The Detector($u2$) automaton receives messages from the channel. If it does not receive anything over an $u2$ time-interval, then it performs a timeout and declares PeriodicSend to be *suspected* to have failed. The *next_recv* variable is used to postpone and then if necessary trigger, the timeout action.

```
automaton PeriodicSend(M : type, u1 : Real)
    where u1 ≥ 0
  signature
    output send(m : M)
    input fail

  variables
    internal next_send : AugmentedReal := 0;
      failed : Bool := false;
      clock : Real := 0;

  transitions
    output send(m, 1, 2)
      pre clock = next_send ∧ ¬failed;
      eff next_send := clock + u1;

    input fail
      eff if ¬failed then
        failed := true; next_send := ∞; fi

  trajectories
    trajdef normal
      stop when clock ≥ next_send
      evolve d(clock) = 1
```

```
automaton Detector(M : type, u2 : Real)
    where u2 > 0
  signature
    input receive(m : M, 1, 2)
    output timedout

  variables
    internal next_recv : AugmentedReal := u2;
      suspected : Bool := false;
      clock : Real := 0

  transitions
    input receive(m, 1, 2)
      eff next_recv := clock + u2

    output timedout
      pre ¬suspected ∧ clock = next_recv
      eff suspected := true;
        next_recv := ∞

  trajectories
    trajdef normal
      stop when ≠ suspected ∧ clock ≥ next_recv
      evolve d(clock) = 1
```

Figure 3-8: Periodic sender and simple failure detector.

Figure 3-9 also presents two key invariants and a partial statement of the simulation relation from Timeout to Spec, which are used for proving that Timeout implements Spec. In Chapter 6 we show how HIOA specifications can be translated to the language of the

```
automaton Timeout(M : type, u1, u2, b : Real) where u1 ≥ 0 ∧ u2 ≥ 0 ∧ b ≥ 0 ∧ u2 ≥ (u1 + b)
  components
    Sender:PeriodicSend(M, u1);
    Detector:Detector(M, u2);
    Channel:TimedChannel(M, b, {1, 2}, 1, 2);

invariant of Timeout: ¬Detector.suspected ⇒ Detector.next_recv ≤ Detector.clock + u2
invariant of Timeout: Detector.suspected ⇒ Sender.failed
...
forward simulation from Timeout(u1, u2, b) to Spec(d):
    (Timeout. Sender.failed = Spec.failed ∧
      Timeout. Detector.suspected = Spec.suspected ∧
      ...)
end
```

Figure 3-9: Composed automaton and property assertions.

PVS Theorem prover [ORR+96] and how invariant properties and simulation relations can be proved partially automatically using specially designed proof strategies.

3.8 Summary

We described the HIOA language for specifying a general class hybrid systems modeled as SHIOAs. In Section 7.7 a slightly extended version of HIOA is described which allows specification of probabilistic transitions. In addition to all examples in this thesis HIOA has been used to specify mobile robot coordination algorithms [LMN], cardiac cell models [GMY+07], an air-traffic control system [UL07], and several real-time distributed algorithms [FDGL07].

Chapter 4

Verifying Safety Properties

In this chapter we present techniques for verifying safety properties of SHIOAs. A safety property S is a predicate on state variables and an SHIOA $\mathcal{A}$ is said to be safe with respect to S, if $\mathsf{Reach}_{\mathcal{A}} \subseteq S$, that is, if S is an invariant for $\mathcal{A}$. In other words, $\mathcal{A}$ is unsafe if there are executions of $\mathcal{A}$ that reach some "bad state" in S^c. We present an overview of existing work in automatic invariance verification in Section 4.1. In Sections 4.2 and 4.3, we present our deductive approach for verifying invariance and implementation relations. Sections 4.4 and 4.4.2 are dedicated to a case study—a supervisory controller for a helicopter testbed—which illustrates modeling and safety verification in the SHIOA framework.

4.1 An Overview

Definition 4.1. Let $\mathcal{A}$ be a pre-SHIOA with set of internal variables X and set of states $Q \subseteq val(X)$, and let $\mathcal{I}$ be a predicate on X. The set of states satisfying $\mathcal{I}$ is also denoted by $\mathcal{I}$. The predicate $\mathcal{I}$ is said to be an *invariant* of $\mathcal{A}$ if $\mathsf{Reach}_{\mathcal{A}} \subseteq \mathcal{I}$.

An SHIOA $\mathcal{A}$ is safe with respect to a particular safety property $\mathcal{I}$, if $\mathcal{I}$ is an invariant for $\mathcal{A}$. Given $\mathcal{A}$ and a predicate $\mathcal{I}$ on X, how do we verify that $\mathcal{I}$ is an invariant of $\mathcal{A}$? If the set of reachable states of $\mathcal{A}$, $\mathsf{Reach}_{\mathcal{A}}$, is computable then we could check if $\mathsf{Reach}_{\mathcal{A}} \subseteq \mathcal{I}$. It has been known since [HKPV95] that computing $\mathsf{Reach}_{\mathcal{A}}$, is decidable only if $\mathcal{A}$ belongs to a fairly restricted class, such as the class of rectangular, initialized SHIOAs. Other decidable classes of hybrid automata with linear, polynomial, and exponential state models have been identified (see, reachability related papers in [TG02, MP03, AP04, MT05, HT06]). These decidable classes are generally called *order minimal* or *o-minimal* hybrid automata [LPY99]. The notion of o-minimality comes from mathematical logic. Informally, A theory of real numbers is said to be o-minimal if every definable subset of $\mathbb{R}$ is a finite union of points and intervals. Then, a hybrid system is o-minimal, if its initial states, preconditions, reset maps, invariant sets, stopping conditions, and DAIs are all definable in the same o-minimal theory.

Another alternative is to compute an overapproximation of $\mathsf{Reach}_{\mathcal{A}}$, say $\overline{\mathsf{Reach}_{\mathcal{A}}}$, and then check if $\overline{\mathsf{Reach}_{\mathcal{A}}} \subseteq \mathcal{I}$. If true then $\mathcal{I}$ is verified, otherwise one has to check if the states in $\overline{\mathsf{Reach}_{\mathcal{A}}} \cap \mathcal{I}$ are really reachable or if they are false-positives generated by the overapproximation. In the latter case $\overline{\mathsf{Reach}_{\mathcal{A}}}$ is iteratively refined. Algorithms for overapproximating $\mathsf{Reach}_{\mathcal{A}}$ for general hybrid systems have been developed (see, for example, [BCT02]), but making these techniques scale remains a challenge.

In practice, therefore, a compromise has to be made between the expressive power of the class of SHIOA that we use for specifying the system and the degree of automation we can hope to achieve in verifying safety. Compared to above mentioned approaches, we opt for a different trade-off between automation and expressive power. We develop invariant proof techniques that are not always automatic but are applicable to general SHIOAs. Moreover, because a set pattern of steps are always executed in constructing these invariant proofs, the process can be at least partially automated.

4.2 Proving Invariants

We deduce the desired invariant property $\mathcal{I}$ from the specification of $\mathcal{A}$ by (a) finding an inductive property $\mathcal{I}' \subseteq \mathcal{I}$, and (b) checking that the transitions and trajectories of $\mathcal{A}$ preserve $\mathcal{I}'$. This technique has been applied earlier in the context of HIOA for verifying safety of air-traffic control systems [LLL99] and vehicle control systems [Lyn96a, WL96, WLD95]. We adapt these techniques to SHIOA and in Chapter 6 we develop theorem prover-based support for partially automating step (b). In order to prove invariant properties, we introduce another class of properties of $\mathcal{A}$.

Definition 4.2. Let $\mathcal{A}$ be a pre-SHIOA with set of internal variables X and set of states $Q \subseteq val(X)$. A predicate $\mathcal{I}$ on X is an *inductive property* of $\mathcal{A}$ if any execution that starts from a state satisfying $\mathcal{I}$ reaches only states that also satisfy $\mathcal{I}$.

An inductive property that is satisfied by all starting states of $\mathcal{A}$ is an invariant. Inductive properties can be verified by checking that each transition and trajectory of $\mathcal{A}$ preserves the property. The following lemma which is an adaptation of the classical inductive proof schema à la Floyd [Flo67] to the HIOA setting, formalizes this verification task.

Lemma 4.1. *Given an HIOA $\mathcal{A}$, the set of states $\mathcal{I} \subseteq Q$ is an invariant of $\mathcal{A}$ if it satisfies:*

1. *(Start condition) For any starting state $\mathbf{x} \in \Theta$, $\mathbf{x} \in \mathcal{I}$.*
2. *(Transition condition) For any action $a \in A$, if $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ and $\mathbf{x} \in \mathcal{I}$ then $\mathbf{x}' \in \mathcal{I}$.*
3. *(Trajectory condition) For any trajectory $\tau \in \mathcal{T}$, if $\tau.fstate \in \mathcal{I}$ then $\tau.lstate \in \mathcal{I}$.*

Proof. It suffices to show that $\mathsf{Reach}_{\mathcal{A}} \subseteq \mathcal{I}$. Consider any reachable state $\mathbf{x} \in \mathsf{Reach}_{\mathcal{A}}$. By the definition of a reachable state, there exists an execution α of $\mathcal{A}$ such that $\alpha.lstate = \mathbf{x}$. We proceed by induction on the length of the execution α. For the base case, α consists of a single starting state $\mathbf{x} \in \Theta$ and by the *start condition*, $\mathbf{x} \in \mathcal{I}$. For the inductive step, we consider two subcases:

Case 1: $\alpha = \alpha' a \mathbf{x}$ where a is an action of $\mathcal{A}$. By the induction hypothesis we know that $\alpha'.lstate \in \mathcal{I}$. Invoking the *transition condition*, we obtain $\mathbf{x} \in \mathcal{I}$.

Case 2: $\alpha = \alpha' \tau$ where τ is a trajectory of $\mathcal{A}$ and $\tau.lstate = \mathbf{x}$. By the induction hypothesis, $\alpha'.lstate \in \mathcal{I}$ and invoking the *trajectory condition* we deduce that $\tau.lstate = \mathbf{x} \in \mathcal{I}$.

■

This technique has been applied to verify safety properties of several hybrid systems (see, for example, [LLL99, LL96, WL96, DL97, WLD95, Lyn96a], and [HL94]). A similar lemma exists for verifying invariant properties of pre-SHIOAs. In practice, the state model structure of SHIOAs is useful for verifying the trajectory condition through case analysis of the state models.

Lemma 4.2. *Given a pre-SHIOA $\mathcal{A}$, the set of states $\mathcal{I} \subseteq Q$ is an invariant of $\mathcal{A}$ if it satisfies:*

1. *(Start condition) For any starting state $\mathbf{x} \in \Theta$, $\mathbf{x} \in \mathcal{I}$.*
2. *(Transition condition) For any action $a \in A$, if $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ and $\mathbf{x} \in \mathcal{I}$ then $\mathbf{x}' \in \mathcal{I}$.*
3. *(Trajectory condition) For any state model $S \in \mathscr{S}$, any trajectory $\tau \in trajs(\mathcal{S})$, if $\tau.fstate \in \mathcal{I}$ then $\tau.lstate \in \mathcal{I}$.*

Proof. It suffices to show that $\mathsf{Reach}_{\mathcal{A}} \subseteq \mathcal{I}$. For any state $\mathbf{x} \in \mathsf{Reach}_{\mathcal{A}}$ there exists an execution α of $\mathcal{A}$ such that $\alpha.lstate = \mathbf{x}$. The proof is by induction on the length of the execution α. For the base case, α is consisting of a single starting state $\mathbf{x} \in \Theta$ and by the *start condition*, $\mathbf{x} \in \mathcal{I}$. For the inductive step, we consider two subcases:

Case 1: $\alpha = \alpha' a \mathbf{x}$ where a is an action of $\mathcal{A}$. By the induction hypothesis we know that $\alpha'.lstate \in \mathcal{I}$. Invoking the *transition condition* we obtain $\mathbf{x} \in \mathcal{I}$.

Case 2: $\alpha = \alpha' \tau$ where τ is a trajectory of $\mathcal{A}$ and $\tau.lstate = \mathbf{x}$. By the induction hypothesis, $\alpha'.lstate \in \mathcal{I}$. Since τ is a trajectory of $\mathcal{A}$, it follows that there exists a state model $\mathcal{S}$ such that $\tau \in trajs(\mathcal{S})$. Invoking the *trajectory condition* we get that $\tau.lstate = \mathbf{x} \in \mathcal{I}$.

■

Often, it is convenient to prove invariants for SHIOAs which are components of a larger, composed system. The following result ensures that any invariant for a component SHIOA carries over to the composed system.

Lemma 4.3. *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-SHIOAs. If $\mathcal{I}$ is an invariant for pre-SHIOA $\mathcal{A}_1$ then $\mathcal{I} \times Q_2$ is an invariant for $\mathcal{A}_1 || \mathcal{A}_2$.*

Proof. First we show that $\mathsf{Reach}_{\mathcal{A}_1||\mathcal{A}_2} \lceil X_1 \subseteq \mathsf{Reach}_{\mathcal{A}_1}$. Fix a state $\mathbf{x} \in \mathsf{Reach}_{\mathcal{A}_1||\mathcal{A}_2} \lceil X_1$. We know that there exists a closed execution α of $\mathcal{A}_1||\mathcal{A}_2$ such that $\alpha.lstate = \mathbf{x}$. From Lemma 2.2, $\alpha \lceil (A_1, V_1)$ is an execution of $\mathcal{A}_1$. Further, $\alpha.lstate \lceil X_1 = (\alpha \lceil (A_1, V_1)).lstate$ which is a reachable state of $\mathcal{A}_1$. Thus, $\mathsf{Reach}_{\mathcal{A}_1||\mathcal{A}_2} \lceil X_1 \subseteq \mathsf{Reach}_{\mathcal{A}_1}$. Since $\mathcal{I}$ is an invariant for $\mathcal{A}_1$, $\mathsf{Reach}_{\mathcal{A}_1} \subseteq \mathcal{I}$ and therefore, $\mathsf{Reach}_{\mathcal{A}_1||\mathcal{A}_2} \lceil X_1 \subseteq \mathcal{I}$. As $\mathcal{I}$ is independent of the variables of $\mathcal{A}_2$, it follows that $\mathsf{Reach}_{\mathcal{A}_1||\mathcal{A}_2} \subseteq \mathcal{I} \times Q_2$. That is, $\mathcal{I} \times Q_2$ is an invariant of the composed automaton $\mathcal{A}_1||\mathcal{A}_2$. ■

4.3 Proving Implementation Relations

Many different kinds of implementation or abstraction relations and their corresponding proof methods have been developed for timed [AD94] and hybrid automata [KLSV05, TK02, TPL02]. For restricted classes hybrid systems, such as o-minimal hybrid automata,

automatic techniques exist for proving implementation relations, particularly bisimulation relations. As in the case of invariant verification, our approach here is to develop proof techniques that are applicable to the general class of SHIOAs.

Implementation relations (see Definition 2.13) are proved by demonstrating the existence of a *simulation relation* between the concerned pair of automata. The following definition of *forward simulation relation* and the corresponding soundness theorem were presented in [LSV03], in the context of HIOAs. Since implementation of pre-SHIOAs is defined in terms of the implementation of the corresponding pre-HIOAs, the soundness result extends to the case of pre-SHIOAs in a straightforward manner.

Definition 4.3. Consider comparable pre-SHIOAs $\mathcal{A}$ and $\mathcal{B}$. A *simulation relation* from $\mathcal{A}$ to $\mathcal{B}$ is a relation $\mathcal{R} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following conditions, for all states $\mathbf{x}$ and $\mathbf{y}$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. *(Start condition)* If $\mathbf{x} \in \Theta_{\mathcal{A}}$ then there exists a state $\mathbf{y} \in \Theta_{\mathcal{B}}$ such that $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$.
2. *(Transition condition)* If $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ and α is an execution fragment of $\mathcal{A}$ consisting of one single action surrounded by two point trajectories and $\alpha.fstate = \mathbf{x}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{y}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ \mathcal{R}\ \beta.lstate$.
3. *(Trajectory condition)* If $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ and α is an execution fragment of $\mathcal{A}$ with consisting of a single closed trajectory and $\alpha.fstate = \mathbf{x}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{y}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ \mathcal{R}\ \beta.lstate$.

Theorem 4.4. *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable pre-SHIOAs and let $\mathcal{R}$ be a simulation relation from $\mathcal{A}$ to $\mathcal{B}$. Then $\mathcal{A} \leq \mathcal{B}$.*

4.4 Case Study: Safety Verification of Helicopter Testbed

In this section and the next, we present the safety verification of a supervisory controller for a helicopter testbed using the techniques presented in Section 4.2. First, we present the HIOA specification of the components of the system and the safety property of interest. This example showcases several features of the HIOA language. Then, in Section 4.4.2, we present the safety verification of the system. Verification involves proving a sequence of intermediate invariants and lemmas. Almost all the intermediate invariants are proved using Lemma 4.2; the remaining are deduced using other, already proved, invariants and the lemmas that assert timing related behavior of the SHIOAs. Invariant properties of the whole system are derived from those of the components of the system by applying Lemma 4.3. The proofs using Lemma 4.2 illustrate how stylized hand-proofs of invariant properties are constructed by systematic analysis of SHIOA specifications—a first step toward automation in deductive verification. In chapter 6 we will discuss how stylized proofs of this kind can be constructed partially automated using mechanical theorem provers.

This case study appeared earlier in [MWLF03]. The helicopter testbed manufactured by Quanser [Qua] (see Figure 4.4) is driven by two rotors mounted at the two ends of its frame. The frame is suspended from an instrumented joint which in turn is mounted at the end of a long arm. The arm is gimbaled on another instrumented joint and is free to pitch and yaw, giving the helicopter three degrees of freedom. The rotor inputs are either controlled by the user with a joystick, or through software controllers. A complete dynamical model

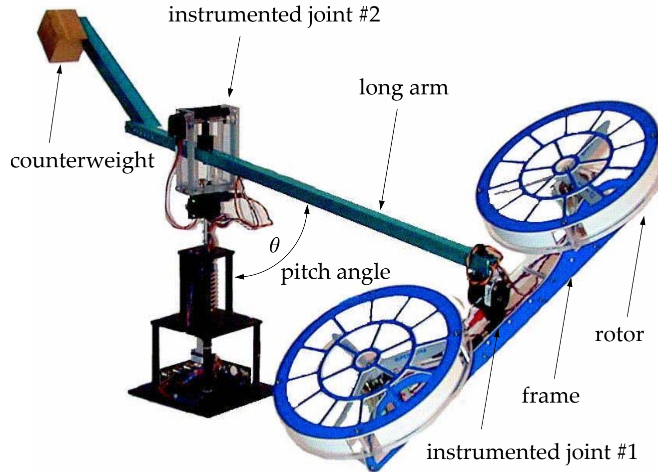

Figure 4-1: Helicopter testbed manufactured by Quanser Inc.

of the helicopter with three degrees of rotational freedom appears in [WIM+02]. Here we consider the pitch dynamics of the helicopter, which are critical for safety. In practice, the roll and yaw effects are eliminated by making the initial conditions along these axes to be zero and giving identical input to the two rotors. The magnitude of the total input to the pitch axis is always restricted to be within $[u_{min}, u_{max}]$ units. If the pitch angle θ is too large the arm stresses the instrumented joint, and if it is too small then the frame hits the ground causing physical damage. The system is safe if the pitch angle remains within a range $[\theta_{min}, \theta_{max}]$ that avoids these extremes.

A supervisory controller is designed to prevent the helicopter from reaching unsafe states. It periodically observes the pitch angle and the pitch velocity of the helicopter, conservatively estimates the worst that may happen if the user is allowed to remain in control, and when necessary for ensuring safety, overrides the user's output with a suitable alternative output. The design of the supervisor is constrained by the actuator bandwidth, the sampling rate, and the sensor inaccuracies. In section 4.4.1, we present the SHIOA specifications of the components that make up the helicopter testbed and state basic properties of these components. In section 4.4.2, we present the verification of the safety of the complete system through a sequence of invariant assertions.

4.4.1 System Specification

Figure 4-2 shows the component SHIOAs of the helicopter testbed, their key state variables, and their interactions through input/output variables and actions. The formal parameters of the actions are not shown in this figure. The helicopter system is the composition of Helicopter, Sensor, UsrCtrl, Actuator, and Supervisor. The resulting composed automaton $\mathcal{A}$ is also a SHIOA. Some of these component automata have state variables with the same name (for example, θ^0 and θ^1). As the set of local variables are required to be disjoint for the automata to be compatible, we rename the overlapping variable names by adding the first letter of the component automaton as a suffix. For the Supervisor automaton we use the suffix c, instead of s.

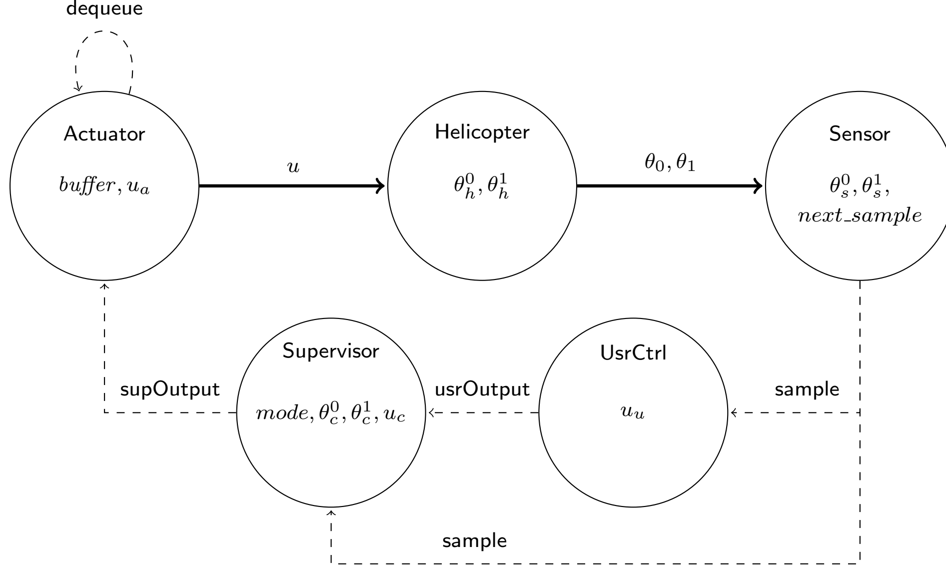

Figure 4-2: Interconnection of SHIOA components in Quanser helicopter system.

Helicopter

The Helicopter SHIOA (Figure 4-3) specifies pitch dynamics of the helicopter with respect to the rotor input voltage u. This automaton does not contain actions or discrete transitions. Its state evolves according to the differential equation: $d(\theta_h^1) + \Omega^2 \cos \theta_h^0 = u$, which relates the pitch angle θ_h^0, the pitch velocity θ_h^1, the characteristic frequency of the system Ω, and the net rotor input u for the pitch axis. The output variables θ_0 and θ_1, which are simply copies of θ_h^0 and θ_h^1, are used for communicating the state of the helicopter to a sensor. The Helicopter is safe if the pitch angle θ_h^0 is within θ_{min} and θ_{max}. The set of safe states S for the composed SHIOA $\mathcal{A}$, is defined as:

$$S \triangleq \{\mathbf{x} \in Q_{\mathcal{A}} \mid \theta_{min} \leq \mathbf{x} \lceil \theta_h^0 \leq \theta_{max}\}. \tag{4.1}$$

---

automaton Helicopter(Ω : Real) **where** $\Omega > 0$
variables
  input u : Real;
  output θ_0 : Real; θ_1 : Real;
  internal θ_h^0 : Real := 0; θ_h^1 : Real := 0;

trajectories
  trajdef pitchDynamics
    evolve $d(\theta_h^0) = \theta_h^1$; $d(\theta_h^1) = -\Omega^2 \cos(\theta_h^0) + u$;
      $\theta_0 = \theta_h^0$; $\theta_1 = \theta_h^1$;

Figure 4-3: Quanser helicopter pitch dynamics.

Sensor

The Sensor automaton (Figure 4-4) models a sensor that reads the pitch angle θ_0 and the pitch velocity θ_1 and produces noisy estimates of these quantities, once every $d_s > 0$ time units. Although the estimates are inaccurate, they are guaranteed to be within $\pm e_0$ and $\pm e_1$ units of the actual values of θ_0 and θ_1, respectively. The estimates for θ_0, θ_1 are encapsulated as the parameters x_0, x_1 of the sample action. The clock variable now_s and the deadline variable $next_sample$ are used to periodically trigger the sample action

```
automaton Sensor(e_0, e_1, d_s : Real)
        where e_0, e_1, d_s > 0
signature
  output sample(x_0, x_1 : Real)

variables
  input θ_0 : Real; θ_1 : Real;
  internal θ_s^0 : Real; θ_s^1 : Real;
    now_s : Real := 0;
    next_sample : AugmentedReal := d_s;
    let time_left := next_sample − now_s

transitions
  output sample(x_0, x_1)
    pre now_s = next_sample
        ∧ x_0 ∈ [θ_s^0 − e_0, θ_s^0 + e_0]
        ∧ x_1 ∈ [θ_s^1 − e_1, θ_s^1 + e_1];
    eff next_sample := next_sample + d_s;

trajectories
  trajdef periodicSample
    stop when now_s = next_sample
    evolve d(now_s) = 1; θ_s^0 = θ_0; θ_s^1 = θ_1;
```

Figure 4-4: Periodic noisy sensor.

in a manner that is similar to the PSend automaton of Figure 3-3. When $next_sample$ equals now_s a set of sample(x_0, x_1) actions are enabled—one for each $x_0 \in [\theta_s^0 - e_0, \theta_s^0 + e_0]$ and $x_1 \in [\theta_s^1 - e_1, \theta_s^1 + e_1]$. Out of all these enabled actions, one particular is chosen nondeterministically. This choice captures the uncertainty in the sensor readings.

Invariant 4.5 captures a key property of Sensor, namely, that the value of the $time_left$ variable (measuring time left to the next sample action) is bounded within 0 and d_s.

Invariant 4.5. In every reachable state of Sensor, $0 \leq time_left \leq d_s$.

Proof. By a straightforward application of Lemma 4.2 it follows that $0 \leq next_sample - now_s \leq d_s$ is an invariant. This suffices because the derived variable $time_left$ is defined to be equal to $next_sample - now_s$. ∎

User-defined Controller

The UsrCtrl automaton (Figure 4-5) models an arbitrary user controller. UsrCtrl receives periodic estimates of the helicopter's state from Sensor through the sample action and responds immediately with a usrOutput(u') action. The parameter u' of the action corresponds to user's choice of input to the helicopter. This choice is modeled by the nondeterministic assignment to the variable u_u over the range $[u_{min}, u_{max}]$ in the effect of Sample. The flag variable $ready_u$ indicates that the sample action has occurred. Similar flags are used in some of the subsequent specifications as well; these variables do not influence the behavior of automata, however by augmenting information content of state, they aid inductive invariant proofs (see, for example, the proof of 4.8).

It is worth remarking that UsrCtrl is a rather abstractly specified user controller which is capable of issuing arbitrarily inputs to the helicopter. A supervisor that ensures safety of the helicopter system with UsrCtrl guarantees safety with respect to any other user controller, because every possible controller implements UsrCtrl. Given a concrete user controller, a formal proof of such a statement can be constructed by applying Theorem 4.4.

Supervisor

The Supervisor automaton (Figure 4-7) models the switched supervisory controller that we have developed for ensuring safety of the helicopter. A second requirement for the Supervisor is to interfere as little as possible with the user controller. There are well known algorithms [FK98, ABD+00, LTS99] for synthesizing controllers for linear hybrid systems.

```
automaton UsrCtrl(u_min, u_max : Real)
signature
  input sample(x_0, x_1 : Real)
  output usrOutput(u' : Real)

variables
  internal now_u : Real := 0;
    u_u : Real := 0;
    ready_u : Bool := false;

transitions
  input sample(x_0, x_1)
    eff u_u := choose k where u_min ≤ k ≤ u_max;
      ready_u := true;

  output usrOutput(u')
    pre u_u = u' ∧ ready_u;
    eff ready_u := false;

trajectories
  trajdef timeElapse
    stop when ready_u
    evolve d(now_u) = 1;
```

Figure 4-5: An arbitrary user-defined controller.

Our design of the switched supervisory controller is based on finding a safe operating region U. The Supervisor receives helicopter state estimates sample(x_0, x_1), user's output usrOutput(u') and decides the input u to the actuator. This input is communicated to the actuator through a supOutput(u) action. Informally, the input u is decided by Supervisor as follows: if $(x_0, x_1) \in U$, the Helicopter is judged to be *very safe*, the user is allowed to remain in control, that is, the supervisor decides u to be be u'. Otherwise, $(x_0, x_1) \notin U$ and the Supervisor decides some other value for u that ensures safety. In order to satisfy the minimal interference requirement, U should be as large as possible. In what follows, we conceptually discuss the derivation of the different switching regions.

Consider a region $C \subseteq S$, from which all the reachable states are contained in S, provided that the input u to the Helicopter is *correct*. By correct we mean that the input is $u = u_{min}$ (or u_{max}) if the pitch angle θ_0 is in the danger of reaching θ_{min} (θ_{max}, resp.). As there is a d_a delay in the actuator (see Figure 4-8), the supervisor cannot bring about instantaneous changes in u, and therefore the region C is not a safe operating region. We define another region $R \subseteq C$ as the set of states from which all reachable states over a period of d_a are within C. Even R is not a safe operating region because the supervisor cannot observe the pitch angle and velocity accurately, and relies on the periodic updates from the inaccurate sensors. Finally, we define the safe operating region U as follows: a sensor-observed state $\mathbf{x}'$ is in U if starting from any actual state $\mathbf{x}$ corresponding to $\mathbf{x}'$, all the reachable states over a d_s interval of time are in R. Switching back to the user controller from the supervisor is performed at the boundary of another region $I \subseteq U$. This hysteresis-like asymmetry in switching prevents high frequency chattering between the user and the supervisory controllers. The regions C, R, U, and I are formally defined as follows:

$$\begin{aligned}
\Gamma^+(\theta, t) &\triangleq -u_{mag}t + \left[2(\Omega^2 \cos\theta_{max} - u_{min})(\theta_{max} - \theta + \frac{1}{2}u_{mag}t^2)\right]^{\frac{1}{2}} \\
\Gamma^-(\theta, t) &\triangleq u_{mag}t - \left[2(u_{max} - \Omega^2)(\theta - \theta_{min} + \frac{1}{2}u_{mag}t^2)\right]^{\frac{1}{2}} \\
U^+(\theta) &\triangleq -e_1 + \Gamma^+(\theta + e_0, d_a + d_s) \\
U^-(\theta) &\triangleq +e_1 + \Gamma^-(\theta - e_0, d_a + d_s) \\
I^+(\theta) &\triangleq -2e_1 + \Gamma^+(\theta + 2e_0, d_a + d_s) \\
I^-(\theta) &\triangleq +2e_1 + \Gamma^-(\theta - 2e_0, d_a + d_s)
\end{aligned}$$

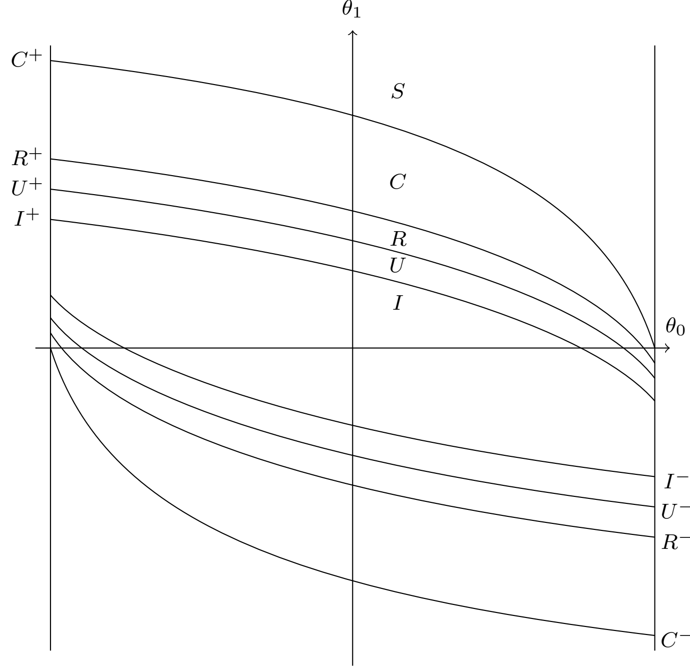

Figure 4-6: Switching regions of supervisory controller.

$$C \triangleq \{\mathbf{x} \in Q_{\mathcal{A}} \mid \mathbf{x} \lceil \theta_h^0 \in [\theta_{min}, \theta_{max}], \mathbf{x} \lceil \theta_h^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_h^0, 0), \Gamma^+(\mathbf{x} \lceil \theta_h^0, 0)]\} \quad (4.2)$$

$$R \triangleq \{\mathbf{x} \in Q_{\mathcal{A}} \mid \mathbf{x} \lceil \theta_h^0 \in [\theta_{min}, \theta_{max}], \mathbf{x} \lceil \theta_h^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_h^0, d_a), \Gamma^+(\mathbf{x} \lceil \theta_h^0, d_a)]\} \quad (4.3)$$

$$U \triangleq \{\mathbf{x} \in Q_{\mathcal{A}} \mid \mathbf{x} \lceil \theta_c^0 \in [\theta_{min} + e_0, \theta_{max} - e_0], \mathbf{x} \lceil \theta_c^1 \in [U^-(\mathbf{x} \lceil \theta_c^0), U^+(\mathbf{x} \lceil \theta_c^0)]\} \quad (4.4)$$

$$I \triangleq \{\mathbf{x} \in Q_{\mathcal{A}} \mid \mathbf{x} \lceil \theta_c^0 \in [\theta_{min} + e_0, \theta_{max} - e_0], \mathbf{x} \lceil \theta_c^1 \in [I^-(\mathbf{x} \lceil \theta_c^0), I^+(\mathbf{x} \lceil \theta_c^0)]\} \quad (4.5)$$

Here $u_{mag} = u_{max} - u_{min}$. The shapes of these regions are as shown in Figure 4-6 for typical values of the parameters u_{mag}, d_a, d_s, e_0, and e_1. The Supervisor reads the sensor-estimated helicopter state through sample(x_0, x_1) and copies x_0, x_1 into internal variables θ_c^0 and θ_c^1. Based on this state information, the value of the tentative output u_{sup} is decided. If the pitch velocity θ_c^1 is greater than or equal to $I^+(\theta_c^0)$, then u_{sup} is set to be u_{max} and if θ_c^0 is less than $I^-(\theta_c^0)$ then u_{sup} is set to be u_{min}. Otherwise, the value of u_{sup} is left unchanged. When usrOutput(u') occurs, u' is copied into u_{usr}. This action also sets the values of the variables u_c and *mode* based on the current values of θ_c^0, θ_c^1, and *mode*. If *mode* $=$ *usr* and the observed state is in U then *mode* remains unchanged and $u_s = u_{usr}$. If *mode* $=$ *usr* but the observed state is not in U then *mode* is changed to *sup* and $u_s = u_{sup}$. If *mode* $=$ *sup* then u_s is copied from u_{sup} and *mode* is set to *usr* only if (θ_c^0, θ_c^1) is in I. The following lemma states the containment relationships among the different switching regions defined by Equations (4.1)-(4.5).

Lemma 4.6. *$I \subseteq U \subseteq R \subseteq C \subseteq S$.*

Proof. Follows from the definitions of C, R, U and I. ∎

```
1  automaton Supervisor(u_min, u_max : Real)
     type
3      Mode = Enumeration [sup, usr]

5    signature
       input sample(x_0, x_1 : Real)
7      input usrOutput(u' : Real)
       output supOutput(u' : Real)
9
     variables
11     internal θ_c^0 : Real := 0; θ_c^1 : Real := 0;
         u_sup : Real; u_usr : Real;
13       u_c : Real := 0; ready_c : Bool := false;
         mode : Mode := usr; rt : Real := 0;
15
     transitions
17     input sample(x_0, x_1)
         eff θ_c^0 := x_0; θ_c^1 := x_1;
19         if θ_c^1 ≥ I^+(θ_c^0) then u_sup := u_min;
           elseif θ_c^1 ≤ I^-(θ_c^0) then u_sup := u_max;
21     else u_sup := 0; fi;
22
     input usrOutput(u')
24     eff u_usr := u'; ready_c := true;
           if mode = usr then
26           if (θ_c^0, θ_c^1) ∈ U
               then u_c := u_usr;
28           else u_c := u_sup; mode := sup; fi;
           elseif mode = sup then
30           if (θ_c^0, θ_c^1) ∈ I
               then u_c := u_usr; mode := usr;
32           else u_c := u_sup; fi; fi;

34     output supOutput(u')
         pre ready_c ∧ u' = u_c;
36       eff ready_c := false;

38   trajectories
       trajdef supMode
40       invariant mode = sup; stop when ready_c
         evolve d(rt) = 1;
42
       trajdef usrMode
44       invariant mode = usr; stop when ready_c
         evolve rt = 0;
```

Figure 4-7: Switched supervisory controller for helicopter testbed.

Actuator

The Actuator automaton (Figure 4-8) models an actuator which delays the output obtained from the supervisory controller by d_a time units. The Actuator also captures a digital-to-analog converter that generates a continuous signal u from the sequence of supOutput(u') actions. Specifically, this conversion works like a zero-order hold circuit: a piece-wise constant signal u is constructed by copying the value of the parameter u' of supOutput actions. The delay in the actuator is modeled by a FIFO queue, *buffer*, of *val* and *deadline* pairs. The *val* is the output u' issued by the Supervisor, *deadline* is the scheduled time at which this output is applied to the helicopter rotors. A supOutput(u') action appends $(u', now_a + d_a)$ to *buffer* and a dequeue action copies $head(buffer).val$ to u_a and removes $head(buffer)$. Starting from 0 which is the *head*, the ith pair in the *buffer*, if present, is denoted by $buffer[i]$. The derived variable *length* is defined to be equal to the length of the *buffer*.

```
automaton Actuator(d_a : Real) where d_a > 0
  signature
    input supOutput(u' : Real)
    output dequeue(u' : Real)

  variables
    output u : Real;
    internal u_a : Discrete Real := 0;
      ready_a : Bool := false;
      buffer: Seq[Tuple[deadline : AugmentedReal,
                        val : Real]] := {};
      now_a : Real := 0;
      let length := length(buffer)

  transitions
    input supOutput(u')
      eff buffer := buffer |− [now_a + d_a, u'];
        ready_a := true;

    internal dequeue(u')
      pre head(buffer).deadline = now_a ∧ ready_a;
      eff u_a := head(buffer).val;
        buffer := tail(buffer); ready_a := false;

  trajectories
    trajdef holding
      stop when head(buffer).deadline = now_a
      evolve d(now_a) = 1; u = u_a;
```

Figure 4-8: Actuator with delay.

Invariant 4.7 states that the *deadlines* corresponding to successive elements in the buffer are non-decreasing, and that all the *deadlines* are between now_a and $now_a + d_a$. This is a typical invariant proof using the induction scheme of Lemma 4.2. Note how the transition and the trajectory conditions are checked by case analysis of the actions and state models of Actuator.

Invariant 4.7. In any reachable state of Actuator, for all $0 \leq i \leq length - 1$, $now_a \leq buffer[i].deadline \leq buffer[i+1].deadline \leq now_a + d_a$.

Proof. The proof is by an application of Lemma 4.2. The start condition holds vacuously because *buffer* is empty. For the transition condition, consider a transition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ such that $\mathbf{x} \lceil buffer$ satisfies the above invariant. Consider two subcases:

Case 1: $a =$ dequeue**.** From the induction hypothesis we know that $\mathbf{x} \lceil buffer$ satisfies the invariant. Since $\mathbf{x}' \lceil buffer = tail(\mathbf{x} \lceil buffer)$, it follows that $\mathbf{x}' \lceil buffer$ also satisfies the invariant.

Case 2: $a =$ supOutput(u')**.** From the transition definition, $\mathbf{x}' \lceil buffer$ equals $\mathbf{x} \lceil buffer$ with one additional pair appended to its end. From the induction hypothesis it is clear that the *deadlines* for all the elements in *buffer*, possibly with the exception of the last one, satisfy the invariant. The *deadline* for the last element is $(\mathbf{x} \lceil now_a) + d_a$, which greater than the *deadline* for the next-to-last element in *buffer*, and therefore $\mathbf{x}'$ satisfies the invariant.

For the trajectory condition, there is just one case to check because the automaton has a single state model holding. Consider a closed trajectory $\tau \in trajs(\text{holding})$, such that $\tau.fstate$ satisfies the invariant. The *buffer* does not change over the trajectory, $\tau.fstate \lceil buffer = \tau.lstate \lceil buffer$, and now_a increases monotonically at the rate of real-time. Thus, it suffices to check that $now_a \leq head(buffer).deadline$ at the last state of τ. Suppose for the sake of contradiction $\tau.lstate \lceil now_a > head(buffer).deadline$. Then, there exists $t \in \tau.dom$, $t < \tau.ltime$, such that $(\tau \downarrow now_a)(t) = head(buffer).deadline$. This means that $\tau(t)$ satisfies the stopping condition for the state model holding and therefore (by Definition 2.11) $t = \tau.ltime$ which contradicts our assumption. ∎

4.4.2 Safety Verification

The composed SHIOA $\mathcal{A}$ is safe if the reachable states of $\mathcal{A}$ are contained in the safe set S, that is, if S is an invariant property for $\mathcal{A}$. As is often the case, however, invariance of S cannot be directly deduced from the specification of $\mathcal{A}$. For this reason, we find a *stronger* invariant property C (defined by Eqution (4.2)).

We verify invariance of C (Theorem 4.22) by proving a sequence of 12 intermediate invariants and 5 lemmas. We have already seen Invariants 4.7, 4.5, and Lemma 4.6. Lemmas 4.6, 4.10,and 4.19, and state relationships between various subsets of C that appear in the invariant properties. These relationships are deduced directly by expanding the definition of the subsets.

The fourth lemma (Lemma 4.11), characterizes the reachable states over any single trajectory in the user mode. The proof of this lemma involves integrating the continuous dynamics of the Helicopter with the worst possible user input. The fifth lemma (Lemma 4.8), asserts a property about the time of occurrence of the sample, usrOutput, and the supOutput

actions. Such timing properties cannot be formalized as invariants, however, in many instances, the design of the system itself suggests a way of proving them. For example, the proof of this Lemma 4.8 is constructed by reasoning about the design of the system, starting from key fact that the sample action occurs periodically (Invariant 4.5).

Almost all of the 12 intermediate invariants are proved using the induction scheme of Lemma 4.2; the rest are deduced from other invariants and lemmas. Applying Lemma 4.2 involves checking the transition condition and the trajectory condition. Checking the transition conditions involves applying the precondition of the action and the invariant on the pre-state, expanding the definition of the post-state as specified by the HIOA program for effects, and deducing that the post-state satisfies the invariant. Most often, this follows from straightforward simplification of the definition of the invariant, the precondition, and the post-state. This also suggests that these proof steps can be automated to a fair degree using theorem provers (This is the topic of Chapter 6).

In general, checking the trajectory conditions involve solving the DAIs describing the state models of $\mathcal{A}$. However in our case study, apart from Invariants 4.13, 4.20, and 4.21, the trajectory conditions are verified easily because the variables involved in the invariant either remain constant over trajectories or they evolve at the constant rate of 1. The proofs of Invariants 4.13 and 4.20 are based on estimating the largest possible envelope of reachable states. The trajectory conditions in these proofs are checked by deriving bounds on the extreme values of the state variables by integrating the differential equations of the Helicopter pitch dynamics, with worst possible user input. The proof of Invariant 4.21 relies on the fact that the input to the Helicopter is *correct* (Invariant 4.15). With this extra information, the trajectory condition of this invariant is checked by showing that the vector field associated with the DAEs of the Helicopter are pointing inwards at the boundary of the invariant set C. This last technique is based on the subtangential condition for checking invariance of continuous time systems as described by Theorem 3.4.11 in [BS67]. This presents a systematic way of checking the trajectory condition for hybrid systems with general state models, without requiring the explicit solution of differential equations.

Our verification methodology does not rely on the exact values of the system parameters $\theta_{min}, \theta_{max}$, u_{min}, u_{max}, d_a, and d_s, however, these parameters must satisfy the following relationships.

$$\theta_{min} < 0 < |\theta_{min}| < \theta_{max}, u_{max} > \Omega^2, \text{ and } u_{min} \leq 0. \tag{4.6}$$

For any trajectory $\tau \in trajs(\mathcal{A})$,

$$\tau.fstate \lceil \theta_c^1 > I^+(\tau.fstate \lceil \theta_c^0) \Rightarrow \tau.lstate \lceil \theta_c^1 \geq I^-(\tau.lstate \lceil \theta_c^0) \tag{4.7}$$

$$\tau.fstate \lceil \theta_c^1 < I^-(\tau.fstate \lceil \theta_c^0) \Rightarrow \tau.lstate \lceil \theta_c^1 \leq I^+(\tau.lsate \lceil \theta_c^0). \tag{4.8}$$

Equations (4.6) are consequences of the physical dimensions of the helicopter. Equations (4.7) and (4.8) bound the length of any closed trajectory in terms of the pitch velocity in its first and last states. Indirectly, this bounds the speed of the helicopter with respect to sampling rate $(1/d_s)$ of the sensor: the state of the helicopter cannot jump across the region I in less than d_s time. We begin by proving some preliminary properties of $\mathcal{A}$; then we proceed to prove safety of $\mathcal{A}$, first in the user mode and finally in the supervisor mode.

4.4.3 Preliminary Properties

Automaton $\mathcal{A}$ has three state variables called now_s, now_a, and now_u, respectively. Each of these variables models real-time and their values are always identical. For simplicity, in the remainder of this section we use a single variable called *now* without any subscript.

The periodic sample action of the Sensor automaton is used to instantly trigger a usrOutput and then a supOutput action. This tight synchronization is formalized by the following lemma.

Lemma 4.8. *In any execution of $\mathcal{A}$,* sample*,* usrOutput*, and* supOutput *actions occur when $now = nd_s$, and* dequeue *actions occur when $timer = d_a + nd_s$ for some integer $n > 0$.*

Proof. Initially $next_sample = d_s$ and every time sample occurs, $next_sample$ is incremented by d_s; $next_sample$ always equals nd_s, where $n \in \mathbb{N}$. The predicate $now = next_sample$ is the enabling condition for sample and it is also the stopping condition for the (only) state model periodicSample. Thus, whenever $next_sample = now = nd_s$, sample is enabled and since time cannot elapse, sample actually occurs.

As an effect of sample, $ready_u$ is set to *true*. Since $ready_u$ is the enabling condition for usrOutput and the stopping condition for the state model timeElapse, usrOutput occurs immediately following sample. Similarly, when usrOutput occurs it sets $ready_c$ to *true* and this forces supOutput to occur immediately.

We have shown that sample, usrOutput, and supOutput always occur in that order, without any time delay between them and whenever they occur $now = nd_s$. Every supOutput inserts an element in the Actuator *buffer*, and the deadline associated with the element is $now+d_a$. Thus, for every element in *buffer* the *deadline* is of the form nd_s+d_a, where $n \in \mathbb{N}$. Finally, $head(buffer).deadline = now$ is an enabling condition for dequeue and a stopping condition for the state model holding. Therefore dequeue occurs when $now = nd_s + d_a$, for some $n \in \mathbb{N}$. ∎

An element is inserted in the *buffer* every d_s time and it is taken out after d_a time. Therefore the length of the *buffer* is bounded. The next invariant uses Lemma 4.8 to limit the length of the Actuator *buffer*.

Invariant 4.9. In every reachable state of $\mathcal{A}$, for all $0 \leq i \leq length - 2$, $buffer[i+1].deadline = buffer[i].deadline + d_s$, and $length \leq \lceil \frac{d_a}{d_s} \rceil$.

Proof. From the proof of Lemma 4.8 we know that the *deadline* for each element in *buffer* has value $nd_s + d_a$ for some $n \in \mathbb{N}$. Furthermore, the difference in the *deadlines* of two consecutive elements is exactly d_s. From Invariant 4.7, the difference between the first and the last *deadline* is at most d_a. Therefore, the maximum number of elements in *buffer* is $\lceil \frac{d_a}{d_s} \rceil$. ∎

We say that a given state $\mathbf{x}$ of $\mathcal{A}$ is in the user mode if $\mathbf{x} \lceil mode = usr$ and it is in the supervisor mode if $\mathbf{x} \lceil mode = sup$. In the next section we prove that $\mathcal{A}$ is safe in the user mode, that is, all reachable states that are in the user mode are contained in S. In Section 4.4.5 we shall prove safety in the supervisor mode.

4.4.4 User Mode

First, we informally describe the main idea underlying the proof. Consider the following worst case scenario: a sample action occurs when the Helicopter is at the boundary of U, say on U^+. The supervisor decides to keep $mode = usr$, and the output from the UsrCtrl continues to drive the Helicopter. That is, the output from UsrCtrl enters the Actuator *buffer*, and then the Helicopter after a delay of d_a time units. This continues for the next d_s time units, after which another sample action occurs, and at this point the supervisor once again gets to decide the mode. Now, what is the worst possible output from UsrCtrl which drives the Helicopter for this d_s interval ? It is u_{max}. This is because the Helicopter is at U^+ at the beginning of the interval, and an input of u_{max} forces the state Helicopter to move most upwards rapidly in the phase-plane. At the next sample point the supervisor detects that the Helicopter is outside U, and starts to feed the appropriate input, namely, u_{min}. But the *buffer* still contains the u_{max} inputs from UsrCtrl, and it takes some more time for u_{min} to actually enter. In this time the state goes outside U, but remains (as we shall shortly prove) within a larger safe set R.

In what follows we shall show that the state of the Helicopter remains within R while the system is in the user mode. In order to accomplish this through invariant assertions, we define a set of regions A_t, for each t, $0 \le t \le d_s$.

$$A_t \triangleq \{\mathbf{x} \mid \mathbf{x} \lceil \theta_h^0 \in [\theta_{min}, \theta_{max}] \wedge \mathbf{x} \lceil \theta_h^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_h^0, d_a + t), \Gamma^+(\mathbf{x} \lceil \theta_h^0, d_a + t)\} \quad (4.9)$$

The boundaries of the A_t regions alongside those of R and U are shown in Figure 4-9 (Left). Lemma 4.10 relates the A_t regions with R and U. Throughout the remainder of this chapter, in all invariant assertions $\mathbf{x}$ denotes a reachable state of $\mathcal{A}$.

Lemma 4.10. *(i)* $A_0 = R$*, (ii)* $U \subseteq A_{d_s}$*, and (iii) for* $0 \le t \le t' \le d_s$*,* $A_{t'} \subseteq A_t$*.*

Proof. For part (i), set $t = 0$ in the definition of A_t and compare with the definition of R. For part (ii), note that $\theta_h^0 \in [\theta_c^0 - e_0, \theta_c^0 + e_0]$ and $\theta_h^1 \in [\theta_c^1 - e_1, \theta_c^1 + e_1]$. Setting $t = d_s$ we have:

$$A_{d_s} = \{\mathbf{x} \mid \mathbf{x} \lceil \theta_c^0 \in [\theta_{min} - \epsilon_0, \theta_{max} + \epsilon_0] \wedge \mathbf{x} \lceil \theta_c^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_h^0, d_a + d_s) - \epsilon_1, \Gamma^+(\mathbf{x} \lceil \theta_h^0, d_a + d_s) + \epsilon_1]\}.$$

Observe that Γ^+ and Γ^- are monotonically increasing and monotonically decreasing, respectively, with respect to θ. Since $\theta_h^0 \ge \theta_c^0 - \epsilon_0$, $\Gamma^-(\theta_h^0, y) \le \Gamma^-(\theta_c^0 - \epsilon_0, y)$. Similarly, $\theta_h^0 \le \theta_h^0 + \epsilon_0$ implies that $\Gamma^+(\theta_h^0, y) \ge \Gamma^+(\theta_c^0 + \epsilon_0, y)$. Therefore,

$$U = \{\mathbf{x} \mid \mathbf{x} \lceil \theta_h^0 \in [\theta_{min}, \theta_{max}] \wedge \\ \mathbf{x} \lceil \theta_h^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_c^0 - \epsilon_0, d_a + d_s) + \epsilon_1, \Gamma^+(\mathbf{x} \lceil \theta_c^0 + \epsilon_0, d_a + d_s) - \epsilon_1]\} \subseteq A_{d_s}.$$

For part (iii), we observe that in the definition for A_t, Γ^+ and Γ^- are monotonically decreasing and monotonically increasing, respectively, with respect to t. Therefore if $0 \le t \le t' \le d_s$ then $A_{t'} \subseteq A_t$. ∎

The proof of safety in the user mode relies on estimating the envelope which contains all states of the Helicopter which can be reached over a d_s interval of time, starting from U, with worst possible user input. The next lemma gives a bound on the reachable states over a single trajectory in terms of the A_t sets. This lemma is used to check the trajectory condition of Invariant 4.12.

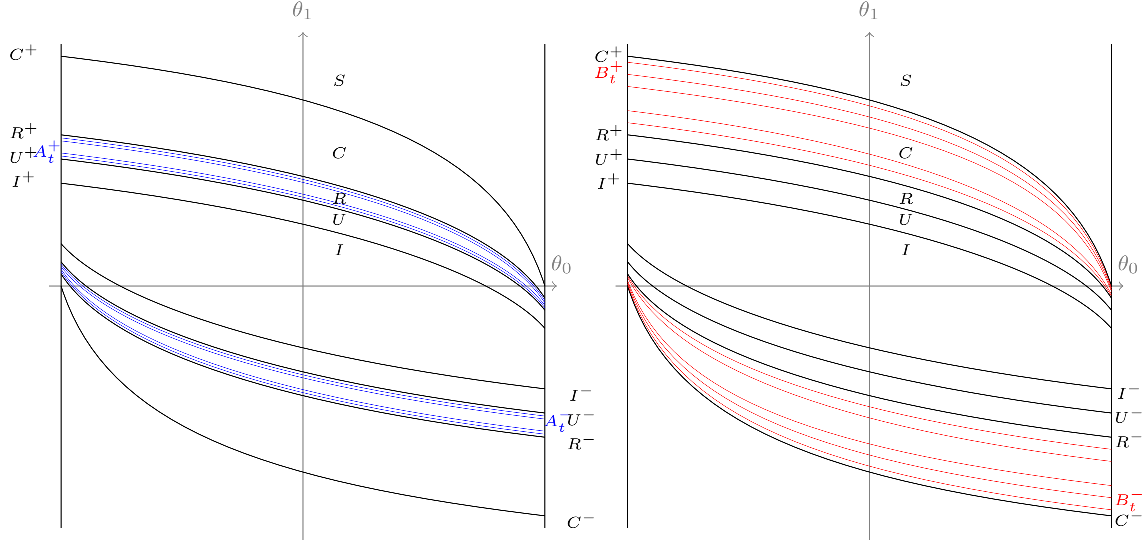

Figure 4-9: *Left:*A_t regions between U and R, *Right:*B_t regions between R and C.

Lemma 4.11. *For any closed trajectory τ of $\mathcal{A}$, if $\tau.fstate \in A_t$ then $\tau.lstate \in A_{t-\tau.ltime}$.*

Proof. Consider a closed trajectory τ. Let $\mathbf{x} = \tau.fstate$, $\mathbf{x}' = \tau.lstate$, and $t' = \tau.ltime$. Suppose $\mathbf{x} \in A_t$, for some t, $0 \leq t \leq d_s$. From the definition of A_t it follows that, $\theta_{min} \leq \mathbf{x} \lceil \theta_h^0 \leq \theta_{max}$ and $\Gamma^-(\mathbf{x} \lceil \theta_h^0, d_a + t) \leq \mathbf{x} \lceil \theta_h^1 \leq \Gamma^+(\mathbf{x} \lceil \theta_h^0, d_a + t)$. We conservatively estimate $\mathbf{x}'$ by considering the worst case input u to Helicopter. First considering the maximum positive input, $u = u_{max}$, from the state model pitchDynamics of Helicopter we get the upper bound on the acceleration at any intermediate state $\mathbf{x}''$ in τ: $d(\mathbf{x}'' \lceil \theta_h^1) \leq -\Omega^2 \cos\theta_{max} + u_{max}$. Integrating from t to t', it follows that $\mathbf{x}' \lceil \theta_h^1 \leq (u_{max} - \Omega^2 \cos\theta_{max})t' + \mathbf{x} \lceil \theta_h^1$, and $\mathbf{x}' \lceil \theta_h^0 \leq \frac{1}{2}(u_{max} - \Omega^2 \cos\theta_{max})t'^2 + (\mathbf{x} \lceil \theta_h^1)t' + \mathbf{x} \lceil \theta_h^0$. Simplifying and using the definition of Γ^+ we get the following bounds on θ_h^0 and θ_h^1: $\mathbf{x}' \lceil \theta_h^0 \leq \theta_{max}$ and $\mathbf{x}' \lceil \theta_h^1 \leq \Gamma^+(\mathbf{x}' \lceil \theta_h^0, d_a + t - t')$. Similarly, considering maximum negative output, $u = u_{min}$, we get the bounds: $\mathbf{x}' \lceil \theta_h^0 \geq \theta_{min}$, and $\mathbf{x}' \lceil \theta_h^1 \geq \Gamma^-(\mathbf{x}' \lceil \theta_h^0, d_a + t - t')$. Combining the above bounds on $\mathbf{x}'$ it follows that $\mathbf{x}' \in A_{t-t'}$. ∎

This gives us the key step in proving Invariant 4.12 which bounds the reachable states in the user mode in terms the A_t sets. From Invariant 4.12, the desired result, namely, the containment of the reachable states of $\mathcal{A}$ in the user mode within R, follows by straightforward application of Lemma 4.2.

Invariant 4.12. If $\mathbf{x} \lceil mode = usr$ and $\neg(\mathbf{x} \lceil ready_u)$ then $\mathbf{x} \in A_{\mathbf{x} \lceil time_left}$.

Proof. The proof is by an application of Lemma 4.2. The start condition holds because for any starting state $\mathbf{x}$, $\mathbf{x} \lceil time_left = d_s$ and $\mathbf{x} \in U \subseteq A_{d_s}$. For the transition condition we consider three possible actions such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$:

Case 1: $a =$ sample(x_0, x_1)**.** $\mathbf{x}'.ready_u = true$ and the invariant holds vacuously.

Case 2: $a =$ usrOutput(u')**.** Assume $\mathbf{x}'.mode = usr$ and consider two sub-cases: if $\mathbf{x} \lceil mode = usr$, then from the code of the usrOutput action in Supervisor, $\mathbf{x} \in U \Rightarrow \mathbf{x}' \in$

$U \subseteq A_{d_s}$. Since $\mathbf{x}' \lceil time_left \leq d_s$, $\mathbf{x}' \in A_{\mathbf{x} \lceil time_left}$. Otherwise, $\mathbf{x} \lceil mode = sup$ and $\mathbf{x} \in I \Rightarrow \mathbf{x}' \in I \subseteq A_{d_s}$. Which implies that $\mathbf{x}' \in A_{\mathbf{x}' \lceil time_left}$.

Case 3: $a = \mathsf{supOutput}(u)$**.** Assume $\mathbf{x}' \lceil mode = usr$ and $\mathbf{x}' \lceil ready_u = false$. Then, $\mathbf{x} \lceil mode = usr$ and $\mathbf{x} \lceil ready_u = false$. By inductive hypothesis $\mathbf{x} \in A_{\mathbf{x} \lceil time_left}$, therefore $\mathbf{x}' \in A_{\mathbf{x}' \lceil time_left}$.

For the trajectory condition, consider a closed trajectory τ. Let $\mathbf{x} = \tau.fstate$, $\mathbf{x}' = \tau.lstate$, and $t' = \tau.ltime$. Assume $\mathbf{x}' \lceil mode = usr$ and $\mathbf{x}' \lceil ready_u = false$. As the valuations of $mode$ and $ready_u$ do not change over τ, $\mathbf{x} \lceil mode = usr$ and $\mathbf{x} \lceil ready_u = false$. From the inductive hypothesis $\mathbf{x} \in A_{\mathbf{x} \lceil time_left}$. Using Lemma 4.11, $\mathbf{x}' \in A_{\mathbf{x} \lceil time_left - t'} = A_{\mathbf{x}' \lceil time_left}$.
∎

Invariant 4.13. If $\mathbf{x} \lceil mode = usr$ then $\mathbf{x} \in R$.

Proof. The proof is by an application of Lemma 4.2. The start condition holds because all starting states are in U and $U \subseteq R$.

For the transition condition, consider any transition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, with $\mathbf{x}' \lceil mode = usr$. We consider two cases based on the value of $ready_u$ in $\mathbf{x}$. If $\mathbf{x} \lceil ready_u = false$, then using Invariant 4.12, $\mathbf{x}' \in A_{\mathbf{x}' \lceil time_left} \subseteq R$. On the other hand, if $\mathbf{x}' \lceil ready_u = true$, then $a \neq \mathsf{usrOutput}$ and $\mathbf{x} \lceil mode = usr$. This is because the $\mathsf{usrOutput}$ action alone can bring about a change of $mode$. So from the inductive hypothesis $\mathbf{x} \in R$ and it follows that $\mathbf{x}' \in R$.

For the trajectory condition, consider a closed trajectory τ. Let $\tau.fstate = \mathbf{x}$, $\tau.lstate = \mathbf{x}'$, and $\mathbf{x}' \lceil mode = usr$. We consider two cases, if $\mathbf{x}' \lceil ready_u = false$ then $\mathbf{x}' \in R$ by Invariant 4.12. Otherwise, $\mathbf{x}' \lceil ready_u = true$. Then $\mathbf{x} \lceil ready_u$ and $\mathbf{x} \lceil mode = usr$ because $ready_u$ and $mode$ does not change over τ. Since $\mathbf{x}$ satisfies the stopping condition for state model $\mathsf{timeElapse}$ in $\mathsf{UsrCtrl}$, τ is a point trajectory. That is, $\mathbf{x}' = \mathbf{x}$. From the inductive hypothesis, $\mathbf{x} \in R$, and so $\mathbf{x}' \in R$. ∎

4.4.5 Supervisor Mode

The $\mathsf{Supervisor}$ receives helicopter state estimates $\mathsf{sample}(x_0, x_1)$, user's output $\mathsf{usrOutput}(u')$ and decides the input u to the actuator. This input is communicated to $\mathsf{Actuator}$ through $\mathsf{supOutput}(u)$, and it finally enters the $\mathsf{Helicopter}$ after d_a time through the occurrence of $\mathsf{dequeue}(u)$. Thus, there are two phases in the supervisor mode. (1) The *settling phase* starts when $mode$ changes from usr to sup and lasts for d_a time units. During this phase the input to the rotors is based on the $\mathsf{usrOutput}$'s issued before the mode switch and therefore the state of the $\mathsf{Helicopter}$ may appear to actually become worse, in the sense that it may continue to move toward the boundary of the safe set S. (2) The *recovery phase* starts d_a time after settling phase. During this phase the input to the rotors is *correct* and is based on the $\mathsf{supOutput}$'s issued after the mode switch and the state of the $\mathsf{Helicopter}$ returns to U. We prove safety of the system in the supervisor mode, by showing that $\mathcal{A}$ remains within C during both these phases.

We begin by proving some basic properties regarding the correctness of the inputs to the $\mathsf{Helicopter}$. Invariant 4.14 states that, in all reachable states with the possible exception of those that are post states of sample actions ($ready_u$ set to $false$), if the sensed helicopter state (θ_c^0, θ_c^1) is within the region bounded by I^+ and I^-, then the system is in the user mode. Invariant 4.15 follows from the code of the sample action. These results are finally

used to prove Invariant 4.17 which relates the state of Helicopter with the position of the correct inputs in the Actuator *buffer*.

Invariant 4.14. If $I^-(\mathbf{x} \lceil \theta_c^0) \leq \mathbf{x} \lceil \theta_c^1 \leq I^+(\mathbf{x} \lceil \theta_c^0) \wedge \neg(\mathbf{x} \lceil ready_u)$ then $\mathbf{x} \lceil mode = usr$.

Proof. The proof is an easy application of Lemma 4.2. The only interesting case to check is the transition condition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ with $a =$ usrOutput and $I^-(\mathbf{x}' \lceil \theta_c^0) \leq \mathbf{x}' \lceil \theta_c^1 \leq I^+(\mathbf{x}' \lceil \theta_c^0)$. It follows from the code that $I^-(\mathbf{x} \lceil \theta_c^0) \leq \mathbf{x} \lceil \theta_c^1 \leq I^+(\mathbf{x} \lceil \theta_c^0)$ and therefore $\mathbf{x}' \lceil mode = usr$. ■

Invariant 4.15. (i) If $\mathbf{x} \lceil \theta_c^1 > U^+(\mathbf{x} \lceil \theta_c^0)$ then $\mathbf{x} \lceil u_{sup} = u_{min}$, and
(ii) if $\mathbf{x} \lceil \theta_c^1 < U^-(\mathbf{x} \lceil \theta_c^0)$ then $\mathbf{x} \lceil u_{sup} = u_{max}$.

Proof. We prove Part (i) using Lemma 4.2. The start condition holds vacuously. For the transition condition, it suffices to check that the invariant is preserved by transitions corresponding to the action sample. From the code for Supervisor, we know that if $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$ when sample is enabled then $\mathbf{x}' \lceil u_{sup} = u_{min}$, where $\mathbf{x}'$ is the post state of the transition corresponding to sample. For the trajectory condition, consider a closed trajectory τ. From the assumption stated in Equation (4.7), for any trajectory τ, if $\tau.fstate \lceil \theta_c^1 > I^+(\tau.fstate \lceil \theta_c^0)$, then $\tau.lstate \lceil \theta_c^1 \geq I^-(\tau.lstate \lceil \theta_c^0)$, which in turn is greater than $U^-(\tau.lstate \lceil \theta_c^0)$. Thus the invariant holds vacuously.

The proof for Part (ii) is symmetric to that of Part (i). ■

The variable rt in Supervisor is a timer measuring duration the system has been in the supervisor mode, since the most recent mode switch. As mode switches occur when $now = nd_s$ (Lemma 4.8), the value of rt can be expressed as $nd_s - time_left$. Invariant 4.16 states this relationship; the proof is a simple application of Lemma 4.2.

Invariant 4.16. $\mathbf{x} \lceil rt = nd_s - \mathbf{x} \lceil time_left$, for some integer $n \geq 1$.

Next, we define two predicates $QPOS_k$ and $QNEG_k$; a state $\mathbf{x}$ satisfies $QPOS_k$ (or $QNEG_k$), if the last k commands in *buffer* are equal to u_{min} (or u_{max} respectively). Formally, for any $k \geq 0$,

$$QPOS_k(\mathbf{x}) \triangleq \forall i,\ \max(0, \mathbf{x} \lceil length - k) \leq i < \mathbf{x} \lceil length, \mathbf{x} \lceil buffer[i].val = u_{min},$$
$$QNEG_k(\mathbf{x}) \triangleq \forall i,\ \max(0, \mathbf{x} \lceil length - k) \leq i < \mathbf{x} \lceil length, \mathbf{x} \lceil buffer[i].val = u_{max}$$

Clearly, for all $k > 0$, $QPOS_k(\mathbf{x})$ implies $QPOS_{k-1}(\mathbf{x})$, and therefore for any $k \geq \mathbf{x} \lceil length$, $QPOS_k(\mathbf{x})$ implies that $QPOS_j(\mathbf{x})$ holds for all $j < \mathbf{x} \lceil length$. Similar results hold for $QNEG_k$. The next invariant states that every reachable state $\mathbf{x}$ in the supervisor mode satisfies either $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(\mathbf{x})$ or $QNEG_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(s)$, depending on whether $\mathbf{x}$ is above I^+ or below I^- respectively. In addition, if $\mathbf{x}$ is in between a supOutput action and a dequeue action ($\mathbf{x} \lceil ready_a$, that is), then $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$ or $QNEG_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$ holds, depending on the position of $\mathbf{x}$ with respect to I^+ and I^-.

Invariant 4.17. If $\mathbf{x} \lceil mode = sup$:

(i) If $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$ then (a) $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(\mathbf{x})$, (b) If $\mathbf{x} \lceil ready_a$ then $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$,

(ii) If $\mathbf{x} \lceil \theta_c^1 < I^-(\mathbf{x} \lceil \theta_c^0)$ then (a) $QNEG_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(\mathbf{x})$, (b) If $\mathbf{x} \lceil ready_a$ then $QNEG_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$

Proof. We shall prove Part (i) of the invariant; the proof for Part (ii) is symmetric. The starting condition holds trivially because $mode = usr$ in all starting states. For the transition condition, we consider the discrete steps $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ with $\mathbf{x}' \lceil mode = sup$ and $\mathbf{x}' \lceil \theta_c^1 > I^+(\mathbf{x}' \lceil \theta_c^0)$.

Case 1: $a =$ **sample.** Since $\mathbf{x} \lceil ready = false$ and $\mathbf{x} \lceil mode = sup$, it follows from the contrapositive of Invariant 4.14 that $\mathbf{x} \lceil \theta_c^1 > I^+(s.\theta_c^0)$ or $\mathbf{x} \lceil \theta_c^1 < I^-(s.\theta_c^0)$. According to Equation (4.7), $\mathbf{x} \lceil \theta_c^1 \geq I^-(s.\theta_c^0)$, therefore $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$. Part (a): from Part (i)(a) of the inductive hypothesis it follows that $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(\mathbf{x})$ holds. Since *buffer* is not changed by sample therefore $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil}(\mathbf{x}')$ holds.

Part (b): assume $\mathbf{x}' \lceil ready_a = true$. Since sample does not change $ready_a$, it follows that $\mathbf{x} \lceil ready_d = true$. Therefore from the inductive hypothesis it follows that $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$ holds. Since *buffer* is not changed by sample therefore $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil + 1}(\mathbf{x}')$ holds.

Case 2: $a =$ **usrOutput.** If $\mathbf{x} \lceil mode = sup$. The invariant is preserved since usrOutput does not change any of the variables involved other than *mode*. If $\mathbf{x} \lceil mode = usr$ then $\mathbf{x} \lceil rt = 0 = \mathbf{x}' \lceil rt$. The invariant is satisfied because $QPOS_0$ is trivially *true*.

Case 3: $a =$ **supOutput.** Part (b): From the code it follows that $\mathbf{x} \lceil mode = sup$ and $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$. Therefore it follows from Invariant 4.15 that $\mathbf{x} \lceil u_{sup} = u_{min}$. Since $\mathbf{x}' \lceil buffer = \mathbf{x} \lceil buffer| - (\mathbf{x} \lceil u_{sup}, \mathbf{x} \lceil now + d_s)$, and $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil}(\mathbf{x})$ holds from the inductive hypothesis, therefore it follows that $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil + 1}(vx')$ holds.

Part (a) follows from the above because $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil + 1}(\mathbf{x}')$ implies that $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil}(\mathbf{x}')$ holds.

Case 4: $a =$ **dequeue.** From the code it follows that $\mathbf{x} \lceil mode = sup$, $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$, $\mathbf{x}' \lceil buffer = tail(\mathbf{x} \lceil buffer)$, and that $\mathbf{x} \lceil ready_d = true$. Part (a): From the inductive hypothesis it follows that $QPOS_{\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil + 1}(\mathbf{x})$ holds, which implies that $QPOS_{\lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil}(\mathbf{x}')$ holds.

Part (b): From the code it follows that $\mathbf{x}'.ready = false$ therefore the invariant holds trivially.

For the trajectory condition, consider a closed trajectory τ. Let $\mathbf{x} = \tau.fstate$, $\mathbf{x}' = \tau.lstate$, and $t' = \tau.ltime$. Suppose $\mathbf{x}' \lceil mode = sup$ and $\mathbf{x}' \lceil \theta_c^1 > I^+(\mathbf{x}' \lceil \theta_c^0)$. From the code it follows that $\mathbf{x}'.buffer = \mathbf{x} \lceil buffer$, $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$ and $\mathbf{x}' \lceil rt = \mathbf{x} \lceil rt + t'$. Using Invariant 4.16 $\mathbf{x} \lceil rt$ can be written as $\mathbf{x} \lceil rt = nd_s - \mathbf{x} \lceil time_left$ for some $n \geq 1$; fix n. Therefore, $\mathbf{x}' \lceil rt = nd_s - \mathbf{x} \lceil time_left + t' = nd_s - \mathbf{x}' \lceil time_left$. Since $0 \leq \mathbf{x} \lceil time_left \leq d_s$ and $0 \leq \mathbf{x}' \lceil time_left \leq d_s$, therefore $\lceil \frac{\mathbf{x} \lceil rt}{d_s} \rceil = \lceil \frac{\mathbf{x}' \lceil rt}{d_s} \rceil = n$. Part (a): from Part (i)(a) of the inductive hypothesis it follows that $QPOS_n(\mathbf{x})$ holds. Since *buffer* is not changed over τ it follows that $QPOS_n(\mathbf{x}')$ holds.

Part (b): Assume $\mathbf{x}' \lceil ready_d = true$. Therefore $\mathbf{x} \lceil ready_d = true$. From Part (i)(b) of the inductive hypothesis it follows that $QPOS_{n+1}(\mathbf{x})$ holds and since *buffer* is not changed over τ it follows that $QPOS_{n+1}(\mathbf{x}')$ holds. ∎

The next invariant formalizes the desired property that after d_a period of time in the supervisor mode the input u to the Helicopter is correct in the following sense: if it is possible for the Helicopter to pitch too high up then $u = u_{min}$; otherwise, if it is possible for it to pitch too low then $u = u_{max}$.

Invariant 4.18. Suppose $\mathbf{x} \lceil mode = sup$ and $\mathbf{x} \lceil rt > d_a$.

(i) If $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$ then $\mathbf{x} \lceil u = u_{min}$, and

(ii) if $\mathbf{x} \lceil \theta_c^1 < I^-(\mathbf{x} \lceil \theta_c^0)$ then $\mathbf{x} \lceil u = u_{max}$.

Proof. We shall prove Part (i); the proof for Part (ii) is symmetric. Consider a reachable state $\mathbf{x}$ and assume that $\mathbf{x} \lceil mode = sup$, $\mathbf{x} \lceil rt > d_a$ and $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$. From part (i) of Invariant 4.17 it follows that $QPOS_{\lceil \frac{d_a}{d_s} \rceil}(\mathbf{x})$ holds. From Invariant 4.9 it is known that the maximum value of $length$ is $\lceil \frac{d_a}{d_s} \rceil$. It follows from the definition of $QPOS$ that $head(\mathbf{x} \lceil buffer) = u_{min}$. ■

Settling Phase of Supervisor Mode

In what follows, we shall show that the state of $\mathcal{A}$ remains within C in the settling phase of the supervisor mode. Our strategy here is similar to the one we adopted in showing safety in the user mode. We define a set of regions B_t, for $0 \leq t \leq d_a$, as follows:

$$B_t \triangleq \{\mathbf{x} \mid \mathbf{x} \lceil \theta_h^0 \in [\theta_{min}, \theta_{max}] \ \wedge \ \ \mathbf{x} \lceil \theta_h^1 \in [\Gamma^-(\mathbf{x} \lceil \theta_h^0, d_a - t), \Gamma^+(\mathbf{x} \lceil \theta_h^0, d_a - t)]\} \quad (4.10)$$

The boundaries of the B_t regions alongside those of R and C are shown in Figure 4-9 (Right). Lemma 4.19 relates the B_t regions with R and C. Invariant 4.20 bounds the location of a state $\mathbf{x}$ in terms of the B_t regions, when $\mathbf{x} \lceil rt \leq d_a$. This implies the safety of the system in the settling phase.

Lemma 4.19. *The regions B_t satisfy: (i) $B_0 = R$, (ii) $B_{d_a} = C$, (iii) for $0 \leq t \leq t' \leq d_a$, $B_t \subseteq B'_t$.*

Proof. Parts (i) and (ii) are proved by setting $t = 0$ and $t = d_a$ in Equation (4.10), respectively. Since $t \leq t'$, Part (iii) follows from monotonicity of Γ^+ and Γ^- with respect to θ. ■

Invariant 4.20. For any reachable state $\mathbf{x}$, if $\mathbf{x} \lceil mode = sup \wedge \mathbf{x} \lceil rt \leq d_a$ then $\mathbf{x} \in B_{\mathbf{x} \lceil rt}$.

Proof. The proof is by an application of Lemma 4.2. The starting condition holds trivially because $\mathbf{x} \lceil mode = usr$. For transition condition, the only interesting case is to check for transitions $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ with $\mathbf{x}' \lceil mode = sup$ and $a =$ usrOutput. There are two subcases to consider: if $\mathbf{x} \lceil mode = sup$ then from the induction hypothesis it follows that $\mathbf{x}' \in B_{\mathbf{x}' \lceil rt}$. Otherwise $\mathbf{x} \lceil mode = usr$, and $\mathbf{x} \in R$ by Invariant 4.12. Thus, $\mathbf{x}' \in R$. Since $R = B_0 \subseteq B_{\mathbf{x}' \lceil rt}$ for any $\mathbf{x}' \lceil rt \geq 0$, and therefore the invariant holds at $\mathbf{x}'$.

For the trajectory condition, consider a closed trajectory τ. Let $\mathbf{x} = \tau.fstate, \mathbf{x}' = \tau.lstate$, and $t' = \tau.ltime$. Assume $\mathbf{x}' \lceil mode = sup$ and $\mathbf{x}' \lceil rt \leq d_a$. Since discrete variables remain constant over trajectories, $\mathbf{x} \lceil mode = sup$ and $\mathbf{x} \lceil rt \leq d_a$. From the induction hypothesis it follows that $\mathbf{x} \in B_{\mathbf{x} \lceil rt}$, that is $\theta_{min} \leq \mathbf{x} \lceil \theta_h^0 \leq \theta_{max}$ and

$\Gamma^-(\mathbf{x} \lceil \theta_p^0, d_a - \mathbf{x} \lceil rt) \leq \mathbf{x} \lceil \theta_h^1 \leq \Gamma^+(\mathbf{x} \lceil \theta_h^0,\ d_a - \mathbf{x} \lceil rt)$. For all intermediate states between $\mathbf{x}$ and $\mathbf{x}'$ the input u to Helicopter is arbitrary. Using the maximum value u_{max}, integrating over $\tau.dom$, and then simplifying we get the upper bounds on θ_h^1 and θ_h^0:

$$\mathbf{x}' \lceil \theta_h^0 \ \leq\ \theta_{max},\ and \tag{4.11}$$
$$\mathbf{x}' \lceil \theta_h^1 \ \leq\ \Gamma^+(\mathbf{x}' \lceil \theta_h^0,\ d_a - \mathbf{x} \lceil rt - t'), \tag{4.12}$$

Similarly, using the lower bound on u, we get

$$\mathbf{x}' \lceil \theta_h^0 \ \geq\ \theta_{min}, \quad and \tag{4.13}$$
$$\mathbf{x}' \lceil \theta_h^1 \ \geq\ \Gamma^-(\mathbf{x}' \lceil \theta_h^0,\ d_a - (\mathbf{x} \lceil rt) - t'). \tag{4.14}$$

Combining equations (4.11)–(4.14) we have $\mathbf{x}' \in B_{\mathbf{x}\lceil rt+t'} = B_{\mathbf{x}'\lceil rt}$. ■

Recovery Phase of Supervisor Mode

In this section we prove Invariant 4.21 which asserts invariance of C in the recovery phase of the supervisor mode. The proof, once again, uses the induction scheme of Lemma 4.2 and in addition uses the fact that the input to the Helicopter is correct (invariant 4.15). Checking the trajectory condition in this proof does not rely on solving the DAEs of the state models. Instead, we show that the vector field corresponding to these differential equations is pointing inwards (subtangential) at the boundary of the set C. The boundary of C is split into four parts, namely, the upper curve C^+, the lower curve C^-, and the segments in the left C^l and right C^r segments, and the above condition is checked for each part.

Invariant 4.21. If $\mathbf{x} \lceil mode = sup$ and $\mathbf{x} \lceil rt \geq d_a$ then $\mathbf{x} \in C$.

Proof. The proof by application of Lemma 4.2. The starting condition is trivially satisfied. For the transition condition, consider discrete transitions $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ with $\mathbf{x}' \lceil mode = sup$. If $a =$ usrOutput there are two subcases: if $\mathbf{x} \lceil mode = sup$ then from the inductive hypothesis $\mathbf{x} \in C$ and therefore $\mathbf{x}' \in C$. Otherwise, $\mathbf{x} \lceil mode = usr$ and $\mathbf{x}' \lceil rt = 0$ and the invariant holds vacuously. For all other actions the invariant is preserved because none of the variables involved are altered.

For the trajectory condition, consider closed trajectory τ. Let $\mathbf{x} = \tau.fstate, \mathbf{x}' = \tau.lstate$, and $t' = \tau.ltime$. Suppose $\mathbf{x}' \lceil mode = sup$ and $\mathbf{x}' \lceil rt \geq d_a$. First, we show that $\mathbf{x} \in C$. Since discrete variables remain constant over τ, $\mathbf{x} \lceil mode = sup$. Consider two possible cases: (1) If $\mathbf{x} \lceil rt < d_a$ then from Invariant 4.20 it follows that $\mathbf{x} \in C$. Otherwise, (2) $\mathbf{x} \lceil rt \geq d_a$ and from the inductive hypothesis it follows that $\mathbf{x} \in C$.

If $\mathbf{x} \in U$, then from Lemma 4.11 it follows that $\mathbf{x}'$ is in R and therefore in C. So it remains to show that if $\mathbf{x} \in C \setminus U$ then $\mathbf{x}' \in C$. We shall prove this by contradiction. Since $\mathbf{x} \lceil \theta_c^1 > I^+(\mathbf{x} \lceil \theta_c^0)$ or $\mathbf{x} \lceil \theta_c^1 < I^+(\mathbf{x} \lceil \theta_c^0)$ it follows from Invariant 4.18 that $\mathbf{x} \lceil u = u_{min}$ or u_{max}, respectively. Now, suppose $\mathbf{x}' \notin C$, then there must exist $t' \in \tau.dom$ such that τ leaves the C at $\tau(t')$. At the boundary of C it must be the case that $d(\theta_h^0(t'), \theta_h^1(t')) \cdot \mathbf{n}(\theta_h^0(t'), \theta_h^1(t')) \geq 0$, where $\mathbf{n}()$ denotes the outer normal to C and $\cdot$ denotes the inner product between the two vectors.

We reach a contradiction by showing that at each point $\mathbf{x}''$ on the boundary of C, $d(\mathbf{x}'' \lceil \theta_h^0, \mathbf{x}'' \lceil \theta_h^1) \cdot \mathbf{n}(\mathbf{x}'' \lceil \theta_h^0, \mathbf{x}'' \lceil \theta_h^1) < 0$. Here onwards we shall write v instead of $\mathbf{x}''.v$ where it is understood that v is the state component of a point in the state space

which is on the boundary of C. We consider the curves defining the boundary of C (refer to Figure 4-6).

Case 1: The upper boundary C^+. This can be expressed as

$$C^+ = \left\{ d(\theta_h^0, \theta_h^1) \mid \theta_{min} \le \theta_h^0 \le \theta_{max} \wedge \theta_h^1 \ge 0 \wedge V_1(\theta_h^0, \theta_h^1) = \left(-u_{min} + \Omega^2 \cos\theta_{max}\right) \theta_{max} \right\},$$

where $V_1(\theta_h^0, \theta_h^1) = \frac{1}{2}{\theta_h^1}^2 + \left(-u_{min} + \Omega^2 \cos\theta_{max}\right)\theta_h^0$. So the outer normal of C^+ is given by

$$\mathbf{n}(\theta_h^0, \theta_h^1) = \nabla V_1 := \left(\frac{\partial V_1}{\partial \theta_p^0}, \frac{\partial V_1}{\partial \theta_h^1} \right) = (-u_{min} + \Omega^2 \cos\theta_{max}, \theta_h^1),$$

where ∇ is the gradient operator. Since $\theta_c^1 \ge I^+(\theta_c^0)$ and $rt > d_a$, $u = u_{min}$ by Invariant 4.18. The pitch dynamics equations are given by: $d(\theta_h^0) = \theta_h^1$, and $d(\theta_h^1) = -\Omega^2 \cos\theta_h^0 + u_{min}$. So we have

$$\begin{aligned} \mathbf{n}(\theta_h^0, \theta_h^1) \cdot d(\theta_h^0, \theta_h^1) &= (-u_{min} + \Omega^2 \cos\theta_{max}, \theta_h^1) \cdot (\theta_h^1, -\Omega^2 \cos\theta_h^0 + u_{min}) \\ &= \Omega^2(\cos\theta_{max} - \cos\theta_h^0)\theta_h^1 \le 0, \end{aligned}$$

for $(\theta_h^0, \theta_h^1) \in C^+$. The equal sign is valid iff $(\theta_h^0, \theta_h^1) = (\theta_{max}, 0)$. So the point $(\theta_h^0, \theta_h^1) = (\theta_{max}, 0)$ needs special treatment. Integrating for initial condition $(\theta_{max}, 0)$, we get

$$\sin\theta_h^0 = \sin\theta_{max} + \frac{1}{\Omega^2}\left[u_{min}(\theta_h^0 - \theta_{max}) - \frac{1}{2}{\theta_h^1}^2 \right]. \tag{4.15}$$

This function defines an integral curve $\theta_h^0 = F_1(\theta_h^1)$. Differentiating (4.15) with respect to θ_h^1,

$$\frac{d\theta_h^0}{d\theta_h^1} = \frac{\theta_h^1}{u_{min} - \Omega^2 \cos\theta_h^0}, \quad \textit{and} \quad \frac{d^2\theta_h^0}{d{\theta_h^1}^2} = \frac{1}{u_{min} - \Omega^2 \cos\theta_h^0} - \frac{\theta_h^1 \sin\theta_h^0}{\left(u_{min} - \Omega^2 \cos\theta_h^0\right)^3}.$$

By evaluating the above derivatives at $(\theta_{max}, 0)$, we have

$$\frac{d\theta_h^0}{d\theta_h^1}(\theta_{max}, 0) = 0, \qquad \frac{d^2\theta_h^0}{d{\theta_h^1}^2}(\theta_{max}, 0) = \frac{1}{u_{min} - \Omega^2 \cos\theta_{max}} < 0.$$

The inequality holds because $u_{min} \le 0$ and $-\frac{\pi}{2} < \theta_h^0 < \frac{\pi}{2}$. So the integral curve $\theta_h^0 = F_1(\theta_h^1)$ achieves a maximum at $(\theta_{max}, 0)$, which implies the trajectory goes inside C.

Case 2: The left boundary. This boundary can be expressed as:

$$C^l \triangleq \left\{ d(\theta_h^0, \theta_h^1) | \theta = \theta_{min} \wedge 0 < \theta_h^1 < \Theta^+ \right\},$$

where $\Theta^+ \triangleq \sqrt{2\left(-u_{min} + \Omega^2 \cos\theta_{max}\right)\left(\theta_{max} - \theta_{min}\right)}$. The outer normal of C^l is given by $\mathbf{n} = (-1, 0)$, and we have $\mathbf{n}(\theta_h^0, \theta_h^1) \cdot d(\theta_p^0, \theta_p^1) = (-1, 0) \cdot (d\theta_h^0, d\theta_h^1) = -d\theta_h^0 = -\theta_h^1 < 0$, for $(\theta_p^0, \theta_p^1) \in C^l$, which implies the trajectory will not leave C through C^l.

Case 3: The lower boundary. The proof is symmetric to that of Case 1.

Case 4: The right boundary. The proof is symmetric to that of Case 2.

By combining all the above cases, we have shown that for any $t'' \in \tau.dom$, at any point on the boundary of C $d(\theta_h^0(t'), \theta_h^1(t')) \cdot \mathbf{n}(\theta_h^0(t'), \theta_h^1(t')) < 0$. Therefore $\mathbf{x}'$ is in C. ∎

Theorem 4.22. *All reachable states of $\mathcal{A}$ are contained in C.*

Proof. For any reachable state $\mathbf{x}$, if $\mathbf{x} \lceil mode = usr$ then $\mathbf{x} \in R \subseteq C$ by Invariant 4.13. Otherwise $\mathbf{x} \lceil mode = sup$, and there are two possibilities: if $\mathbf{x} \lceil rt < d_a$ then, by Invariant 4.20, $\mathbf{x} \in B_{\mathbf{x}\lceil rt} \subseteq C$. Else $\mathbf{x} \lceil rt \geq d_a$ and it follows from Invariant 4.21 that $\mathbf{x} \in C$. ∎

4.5 Summary

We presented techniques for verifying invariants (Lemma 4.2) and implementations (Theorem 4.4) for a very general class of hybrid systems, namely, the class of SHIOAs. Invariant properties can express safety and hence these techniques are useful for safety verification. These techniques have the following features:

(a) Proofs decompose into independent discrete (transition condition) and the continuous (trajectory condition) parts, thus enabling us to employ control-theoretic and deductive techniques within the same proof.

(b) The proof style is purely assertional, that is, based on the current state of the system, rather than complete executions. Experience from verifying complex distributed systems indicate that assertional proofs less error prone and are easier to check.

(c) Where non-assertional proofs are necessary for proving timing dependent properties (for example, Lemma 4.8), the proof follows naturally from the design of the system.

These proof techniques do require the user to supply the inductive properties and simulation relations which typically require some understanding of the system behavior. Indeed, techniques for automatically generating inductive invariants [Meg01, SSM04] could be used in conjunction with our method for this purpose.

The case study presented in this chapter illustrates the generality of the proof techniques in handling complexity that arises discrete transitions and the continuous dynamics. It also illustrates the pattern of proof steps—application of Lemma 4.2, case analysis of actions and state models—that is repeated in most invariant proofs. These patterns form the basis for partially automating proofs in Chapter 6 through theorem provers strategies.

Chapter 5

Verifying Stability Properties

In this chapter we present a set of techniques for verifying stability properties of SHIOAs. Informally, an SHIOA is said to be stable if it converges to an equilibrium state starting from *any* state. The invariance proof techniques of Chapter 4 crucially rely on the assumption that the starting states of the system satisfy the invariant, and therefore those techniques are not directly applicable for verifying stability. Verifying stability of hybrid systems is challenging because the stability of each individual state model does not necessarily imply the stability of the whole automaton.

The techniques presented here rely on results from the literature on switched systems [Lib03, vdSS00]. In the switched system model, details of the discrete mechanisms, namely, the preconditions and the effects of transitions, are neglected. Instead, an exogenous *switching signal* brings about the switches between the different state models. Assuming that the individual state models of a hybrid system are stable one can then characterize the class of switching signals, based only on the rate of switches and not the particular sequence of switches, that guarantee stability of the whole system. The notion of *dwell time* [Mor96] and the more general *average dwell time (ADT)* [HM99] precisely define such restricted classes of switching signals that guarantee stability of the whole system. Analogously, one can develop sufficient conditions for verifying stability of SHIOAs: given an SHIOA $\mathcal{A}$ such that the individual state models of $\mathcal{A}$ are stable, if the ADT of $\mathcal{A}$ is greater than a certain constant (a function of the state model dynamics), then $\mathcal{A}$ is stable. However, application of this criterion relies on checking that the ADT of $\mathcal{A}$ is greater than some constant—a property that depends on the rate of mode switches over *all* executions of $\mathcal{A}$. How does one verify such ADT properties? In this chapter, we develop techniques for accomplishing this verification task.

5.1 Assumptions

The techniques developed in this chapter rely on the following assumptions:

(1) Since we are concerned with internal stability of hybrid systems, we assume input/output variables and input actions are absent, that is, $U = Y = I = \emptyset$.

(2) The collection of state models $\mathscr{S}$ of $\mathcal{A}$ is finite; the state models are indexed by a finite index set $I = \{1, \ldots, m\}$, for some $m \in \mathbb{N}$. The individual state models of $\mathcal{A}$ are $\mathcal{S}_i$, $i \in I$, as in Definition 2.10. Further, suppose $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ is a discrete transition with $\mathbf{x} \in Inv_i$ and $\mathbf{x}' \in Inv_j$, $i, j \in I$. If the transition changes the value of the continuous

variables then $i \neq j$. The last part is without loss of generality because the set of invariants Inv_i, $i \in I$, can be redefined, possibly by adding new elements to I, such that the required condition is met.

(3) For each state model $\mathcal{S}_i, i \in I$ the collection of V-DAIs F_i is described by differential equations in the vector notation of the form $d(\mathbf{x}_c) = f_i(\mathbf{x}_c)$, where f_i is a well behaved (locally Lipschitz) function, and $\mathbf{x}_c$ is a real-valued vector of continuous variables of $\mathcal{A}$.

(4) The individual state models are stable. That is, the trajectories defined by individual state models converge to some equilibrium point in the state-space, say the origin, without loss of generality. Formally, $f_i(0) = 0$ for each $i \in I$. Here f_i is the right-hand side of the V-DAI F_i for the state model $\mathcal{S}_i$, and its argument is the zero vector.

Assumptions (3) and (4) together imply that the differential equations in the state models have solutions defined globally in time. Formally, for each $i \in I$ and $\mathbf{x} \in Inv_i$, there exists a trajectory τ starting from $\mathbf{x}$ that satisfies the V-DAI F_i and if $\tau.dom$ is finite then some local action in $L_{\mathcal{A}}$ is enabled at $\tau.lstate$. Assumptions (2)–(4) are essential for the validity of the average dwell time theorem of Hespanha and Morse (Theorem 5.2) on which our verification techniques rely. Relaxing these assumptions and verifying more general sufficient conditions for stability of SHIOAs is an avenue for future research.

5.2 Stability and Average Dwell Time

We adopt the standard stability definitions [Kha02] and state them in the language of SHIOAs. We remind the reader that the shorthand notation $\alpha(t)$ denotes the valuation of the state variables of an SHIOA $\mathcal{A}$ in the execution α at time $t \in [0, \alpha.ltime]$. Formally, for a closed execution α, $\alpha(t) \triangleq \alpha'.lstate$, where α' is the longest prefix of α with $\alpha'.ltime = t$.

5.2.1 Stability Definitions

Stability is a property of the continuous variables of SHIOA $\mathcal{A}$, with respect to the standard Euclidean norm in $\mathbb{R}^n$. The Euclidean norm of $\alpha(t)$, denoted by $|\alpha(t)|$, is restricted to the set of real-valued continuous variables. We say that a state $\mathbf{x}$ of $\mathcal{A}$ is within a ball of radius δ about the origin, if the norm of the continuous variables at $\mathbf{x}$ is at most δ.

Definition 5.1. The origin is a *stable* equilibrium point of a SHIOA $\mathcal{A}$, in the sense of Lyapunov, if for every $\epsilon > 0$, there exists a $\delta_1 = \delta_1(\epsilon) > 0$, such that for every closed execution α of $\mathcal{A}$, $|\alpha(0)| \leq \delta_1$ implies that $|\alpha(t)| \leq \epsilon$ for all t, $0 \leq t \leq \alpha.ltime$. In this case, we say that $\mathcal{A}$ is *stable*.

For stable $\mathcal{A}$, the state can be bounded in an arbitrarily small ball of radius ϵ, by starting the automaton from a state within a suitably chosen smaller ball of radius δ_1.

Definition 5.2. An SHIOA $\mathcal{A}$ is *asymptotically stable* if it is stable and there exists $\delta_2 > 0$ so that for any execution fragment $|\alpha(0)| \leq \delta_2$ implies that $\alpha(t) \to 0$ as $t \to \infty$. If the above condition holds for all δ_2 then $\mathcal{A}$ is *globally asymptotically stable*.

Assumption (3) ensures that all state models have well-defined global solutions and therefore, any execution can be extended to ∞ A stable SHIOA is also asymptotically stable if we can choose a ball of radius δ_2, such that starting from any initial state within

the ball of radius δ_2, as time goes to infinity, the state converges to the equilibrium state. For general nonlinear hybrid systems, this convergence property alone does not imply stability in the sense of Lyapunov. Global asymptotic stability implies that the above condition holds for executions starting from any state.

Definition 5.3. An SHIOA $\mathcal{A}$ is said to be *exponentially stable* if there exist positive constants δ, c, and λ such that all closed executions fragments with $|\alpha(0)| \leq \delta$ satisfy the inequality $|\alpha(t)| \leq c|\alpha(0)|e^{-\lambda t}$, for all t, $0 \leq t \leq \alpha.ltime$. If the above holds for all δ then $\mathcal{A}$ is said to be *globally exponentially stable*.

In the above definitions, the constants are quantified prior to the executions, and hence, these notions of stability are *uniform over executions*. All stability related discussions in this thesis will be concerned with notions that are uniform in the above sense. We will employ the term "uniform" in the more conventional sense to describe uniformity with respect to the initial time of observation. Thus, *uniform stability* guarantees that the stability property in question holds not just for all executions, but for any suffix of the executions, that is, for all reachable execution fragments.

Definition 5.4. An SHIOA $\mathcal{A}$ is *uniformly stable* in the sense of Lyapunov, if for every $\epsilon > 0$ there exists a constant $\delta_1 = \delta_1(\epsilon) > 0$, such that for any reachable closed execution fragment α, $|\alpha(0)| \leq \delta_1$ implies that $|\alpha(t)| \leq \epsilon$, for all t, $0 \leq t \leq \alpha.ltime$.

Definition 5.5. An SHIOA $\mathcal{A}$ is said to be uniformly asymptotically stable if it is uniformly stable and there exists $\delta_2 > 0$, such that for every $\epsilon > 0$ there exists a $T > 0$, such that for any reachable execution fragment α,

$$|\alpha(0)| \leq \delta_2 \Rightarrow |\alpha(t)| \leq \epsilon, \quad \forall t \geq T \tag{5.1}$$

It is said to be *globally uniformly asymptotically stable* if the above holds for all δ_2, with $T = T(\delta_2, \epsilon)$.

Definition 5.6. An SHIOA $\mathcal{A}$ is *uniformly exponentially stable* if it is uniformly stable and there exist δ, c, and λ, such that for any reachable closed execution fragment α, if $|\alpha(0)| \leq \delta$ then $|\alpha(t)| \leq c|\alpha(0)|e^{-\lambda t}$, for all $0 \leq t \leq \alpha.ltime$. $\mathcal{A}$ is *globally uniformly exponentially stable* if the above holds for all δ with constant c and λ.

5.2.2 ADT Theorem of Heshpanha and Morse

It is well known that a switched system is stable if all the individual subsystems are stable and the switching between state models is sufficiently slow, so as to allow the dissipation of the transient effects after each switch. The *dwell time* [Mor96] and the more general *average dwell time* [HM99] criteria define restricted classes of switching signals, based on switching speeds, and one can conclude the stability of a system with respect to these restricted classes.

Definition 5.7. Let $\mathcal{A}$ be an SHIOA with state models indexed by a finite set I. A discrete transition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ of $\mathcal{A}$ is said to be a *mode switch* if for some $i, j \in I, i \neq j$, $\mathbf{x} \in Inv_i$ and $\mathbf{x}' \in Inv_j$. The set of mode switching transitions of $\mathcal{A}$ is denoted by $\mathcal{M}$. Given an execution fragment α of $\mathcal{A}$, the number of mode switches over α is denoted by $N(\alpha)$.

A discrete transition is a mode switch if its pre- and post-states satisfy invariants of different different state models. This implies that different sets of differential equations guide the evolution of the continuous variables, before and after a mode switch.

Definition 5.8. Given a duration of time $\tau_a > 0$, SHIOA $\mathcal{A}$ has *Average Dwell Time (ADT)* τ_a if there exists a positive constant N_0, such that for every reachable execution fragment α,

$$N(\alpha) \leq N_0 + \alpha.ltime/\tau_a, \tag{5.2}$$

The number of *extra switches* of α with respect to τ_a is defined as $S_{\tau_a}(\alpha) := N(\alpha) - \alpha.ltime/\tau_a$.

Lemma 5.1. *Suppose $\mathcal{A}$ is an SHIOA and $\tau_a > 0$ is an average dwell time for $\mathcal{A}$. Then, any τ_a' that is $0 \leq \tau_a' < \tau_a$ is also an average dwell time of $\mathcal{A}$*

Proof. Inequality (5.2) is satisfied if we replace τ_a with a smaller τ_a'. ∎

Theorem 1 from [HM99], adapted to SHIOA, gives a sufficient condition for stability based on average dwell time. Informally, it states that a hybrid system is stable if the discrete switches are between modes which are individually stable, provided that the switches do not occur too frequently on the average.

Theorem 5.2. *Suppose there exist positive definite, continuously differentiable functions $\mathcal{V}_i : \mathbb{R}^n \to \mathbb{R}^n$, for each $i \in I$, such that we have two positive numbers λ_0 and μ, and two strictly increasing continuous functions β_1, β_2 such that:*

$$\beta_1(|\mathbf{x}_c|) \leq \mathcal{V}_i(\mathbf{x}_c) \leq \beta_2(|\mathbf{x}_c|), \quad \forall \mathbf{x}_c, \quad \forall i \in I, \tag{5.3}$$

$$\frac{\partial \mathcal{V}_i}{\partial \mathbf{x}_c} f_i(\mathbf{x}_c) \leq -2\lambda_0 \mathcal{V}_i(\mathbf{x}_c), \quad \forall \mathbf{x}_c, \quad \forall i \in I, \text{ and} \tag{5.4}$$

$$\mathcal{V}_i(\mathbf{x}_c') \leq \mu \mathcal{V}_j(\mathbf{x}_c), \quad \forall \mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}', \text{ where } i = \mathbf{x}' \lceil \text{mode and } j = \mathbf{x} \lceil \text{mode}. \tag{5.5}$$

Then, $\mathcal{A}$ is globally uniformly asymptotically stable if it has an ADT $\tau_a > \frac{\log \mu}{\lambda_0}$.

Its worth making a few remarks about this theorem. First of all, it is well-known that if the state model $\mathcal{S}_i, i \in I$ is globally asymptotically stable, then there exists a Lyapunov function $\mathcal{V}_i$ that satisfies (5.3) and $\frac{\partial \mathcal{V}_i}{\partial \mathbf{x}_c} f_i(\mathbf{x}_c) \leq -2\lambda_i V_i(\mathbf{x}_c)$, for appropriately chosen $\lambda_i > 0$. As the index set I is finite a λ_0 independent of i can be chosen such that for all $i \in I$, Equation (5.4) holds. The third assumption, Equation 5.5, restricts the maximum increase in the value of the current Lyapunov functions over any mode switch, by a factor of μ.

In [HM99] and [Lib03] this theorem is presented for the switched system model which differs from the more general SHIOA model in two ways: (a) In the switched system model, all variables are continuous except for the *mode* variable which determines the active state model. In SHIOA, there are both discrete and continuous variables. (b) The (discrete) transitions of a switched system correspond to the switching signal changing the value of *mode*; values of continuous variables remain unchanged over transitions. In SHIOAs, transitions can change the value of continuous variables. For example, a stopwatch is typically modeled as a continuous variable that is reset by discrete transitions. The proof of Theorem 5.2 still works for the SHIOA model because for this analysis, it suffices to consider only those discrete transitions of SHIOAs that are also mode switches. Assumption (2) guarantees that non-mode switching transitions do not change the value of the continuous variables. Secondly, resetting continuous variables change the value of the Lyapunov functions but hypothesis 5.5 guarantees that the change is bounded by a factor of μ.

Proof sketch for Theorem 5.2. This proof is adapted from the proof of Theorem 3.2 of [Lib03] which constructs an exponentially decaying bound on the Lyapunov functions of each mode along any execution. Suppose α is any execution of $\mathcal{A}$. Let $T = \alpha.ltime$ and $t_1, \ldots, t_N$ be instants of mode switches in α. We will find an upper-bound on the value of $\mathcal{V}_{\alpha(T)\lceil mode}(\alpha(T))$, where $\alpha(t) \lceil mode \triangleq i, i \in I$ if and only if $\alpha(t) \in Inv_i$. We define a function $W(t) \triangleq e^{2\lambda_0 t}\mathcal{V}_{\alpha(t)\lceil mode}(\alpha(t))$. Using the fact that W is non-increasing between mode switches and Equation 5.5 it can be shown that $W(t_{i+1}) \leq \mu W(t_i)$. Iterating this inequality $N(\alpha)$ times we get $W(T) \leq \mu^{N(\alpha)} W(0)$, that is

$$\begin{aligned} e^{2\lambda_0 T}\mathcal{V}_{\alpha(T)\lceil mode}(\alpha(T)) &\leq \mu^N(\alpha)\mathcal{V}_{\alpha(0)\lceil mode}(\alpha(0)), \\ \mathcal{V}_{\alpha(T)\lceil mode}(\alpha(T)) &\leq e^{-2\lambda_0 T + N(\alpha)\log\mu}\mathcal{V}_{\alpha(0)\lceil mode}(\alpha(0)) \end{aligned}$$

If α has average dwell time τ_a, then

$$\begin{aligned} \mathcal{V}_{\alpha(T)\lceil mode}(\alpha(T)) &\leq e^{-2\lambda_0 T + (N_0 + \frac{T}{\tau_a})\log\mu}\mathcal{V}_{\alpha(0)\lceil mode}(\alpha(0)) \\ &\leq e^{N_0\log\mu} e^{(\frac{\log\mu}{\tau_a} - 2\lambda_0)T}\mathcal{V}_{\alpha(0)\lceil mode}(\alpha(0)). \end{aligned}$$

Now, if $\tau_a > \frac{\log\mu}{2\lambda}$ then $\mathcal{V}_{\alpha(T)\lceil mode}(\alpha(T))$ converges to 0 as $T \to 0$. Then from (5.3) it follows that $\mathcal{A}$ is globally asymptotically stable.

5.3 An Overview

A large average dwell time means that the system spends enough time in each mode, so as to dissipate the transient energy gained through mode switches. This itself is not sufficient for stability; in addition, the individual modes of the automaton must also be stable. In fact, the problem of proving the stability of a hybrid system can be broken down into (a) finding Lyapunov functions for the individual modes, and (b) checking the appropriate ADT property. In this chapter, we assume that a solution to part (a)—a set of Lyapunov functions for the individual modes—is known from existing techniques from systems theory [Kha02], and we present methods for accomplishing (b).

ADT properties have proved to be helpful in analyzing different forms of stability in various contexts other than those we study in this chapter. For example, stability of systems with mixed stable and unstable state models [ZHYM00], input-to-state stability in the presence of inputs [VCL06], and stochastic stability of randomly switched systems [CL06]. In many settings, the question of whether a system have a certain ADT is natural and interesting independent of its connection to stability. For example, in queuing systems the ADT properties are used to characterize burstiness of traffic [Cru91]. Our ADT verification techniques are likely to be valuable in these other contexts as well.

In general, it is hard to prove properties like the ADT property which are quantified over all the executions of an automaton. In Section 2.1, we define what it means for a given SHIOA to be equivalent to another SHIOA with respect to ADT. In order to prove such a relationship between a pair of SHIOAs, we introduce *switching simulation relations*, much like ordinary simulation relations of Section 4.3. This provides us a method for abstracting $\mathcal{A}$ with respect to ADT-properties. In Sections 5.6.2 and 5.6.5 we apply this abstraction

technique for verifying ADT properties

In Section 5.5, we present our first method for ADT verification which relies on checking invariant properties. In order to check if automaton $\mathcal{A}$ has ADT τ_a, we transform it to a new automaton $\texttt{Scount}(\mathcal{A}, \tau_a)$, such that $\mathcal{A}$ has ADT τ_a if and only if $\texttt{Scount}(\mathcal{A}, \tau_a)$ has a particular invariant property $\mathcal{I}(\tau_a)$. This enables us to prove ADT properties using the tools and techniques available for proving invariants. As discussed in Section 4.2, it is known that for certain classes SHIOAs, such as rectangular, initialized SHIOA and o-minimal hybrid automata, the reachability problem is decidable and therefore invariants can be checked automatically. For these classes, this invariant-based technique yields an automatic method for verifying ADT properties. For SHIOAs that lie outside these decidable classes, the invariant proof technique of Chapter 4 can be applied to verify ADT.

The second method, presented in Section 5.6, is based on a complementary approach: we attempt to find an execution of the automaton that violates the ADT property. Failure to find such a counterexample execution indicates that the ADT property is satisfied by the SHIOA. The search for a counterexample execution is formulated as an optimization problem. For checking if automaton $\mathcal{A}$ has ADT τ_a, we formulate and solve an optimization problem $\mathsf{OPT}(\tau_a)$. From the solution of $\mathsf{OPT}(\tau_a)$ we either get a counterexample execution fragment of $\mathcal{A}$ that violates the ADT property τ_a, or else we get a proof that no such counterexample exists, and that $\mathcal{A}$ has ADT τ_a. We show that for certain classes of SHIOAs $\mathsf{OPT}(\tau_a)$ can indeed be formulated and solved using standard mathematical programming techniques.

5.4 ADT Equivalence

In order to check whether τ_a is an ADT for a given SHIOA $\mathcal{A}$, it is often easier to check the same ADT property for another, more abstract, SHIOA $\mathcal{B}$ that is "equivalent" to $\mathcal{A}$ with respect to switching behavior. This notion of equivalence is formalized as follows.

Definition 5.9. Given SHIOAs $\mathcal{A}$ and $\mathcal{B}$, if for all $\tau_a > 0$, τ_a is an ADT for $\mathcal{B}$ implies that τ_a is an ADT for $\mathcal{A}$, then we say that $\mathcal{A}$ *switches slower* than $\mathcal{B}$ and write this as $\mathcal{A} \leq_{switch} \mathcal{B}$. If $\mathcal{B} \leq_{switch} \mathcal{A}$ and $\mathcal{A} \leq_{switch} \mathcal{B}$ then we say $\mathcal{A}$ and $\mathcal{B}$ are *ADT-equivalent*.

We propose an inductive method for proving ADT-equivalence. The key idea is to use a new variety of forward simulation relation that we encountered in Section 4.3, in the context of verification of trace inclusions. Here, instead of the trace of an execution, we are concerned with the number of mode switches that occur and the amount of time that elapses over an execution.

Definition 5.10. Consider SHIOAs $\mathcal{A}$ and $\mathcal{B}$. A *switching simulation relation* from $\mathcal{A}$ to $\mathcal{B}$ is a relation $\mathcal{R} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following conditions, for all states $\mathbf{x}$ and $\mathbf{y}$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. *(Start condition)* If $\mathbf{x} \in \Theta_{\mathcal{A}}$ then there exists a state $\mathbf{y} \in \Theta_{\mathcal{B}}$ such that $\mathbf{x} \mathrel{\mathcal{R}} \mathbf{y}$.

2. *(Transition condition)* If $\mathbf{x} \mathrel{\mathcal{R}} \mathbf{y}$ and α is an execution fragment of $\mathcal{A}$ with $\alpha.fstate = \mathbf{x}$ and consisting of one single action surrounded by two point trajectories, then $\mathcal{B}$ has a closed execution fragment β, such that $\beta.fstate = \mathbf{y}$, $N(\beta) \geq 1$, $\beta.ltime = 0$, and $\alpha.lstate \mathrel{\mathcal{R}} \beta.lstate$.

3. *(Trajectory condition)* If $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ and α is an execution fragment of $\mathcal{A}$ with $\alpha.fstate = \mathbf{x}$ and consisting of a single closed trajectory τ of a particular state model $\mathcal{S}$, then $\mathcal{B}$ has a closed execution fragment β, such that $\beta.fstate = \mathbf{y}$, $\beta.ltime \leq \alpha.ltime$, and $\alpha.lstate\ \mathcal{R}\ \beta.lstate$.

Note that SHIOAs $\mathcal{A}$ and $\mathcal{B}$ are not necessarily comparable.

Lemma 5.3. *Let $\mathcal{A}$ and $\mathcal{B}$ be SHIOAs, and let $\mathcal{R}$ be a switching simulation relation from $\mathcal{A}$ to $\mathcal{B}$, then for all $\tau_a > 0$ and for every execution α of $\mathcal{A}$, there exists an execution β of $\mathcal{B}$ such that $S_{\tau_a}(\alpha) \leq S_{\tau_a}(\beta)$.*

Proof. We fix τ_a and α and construct an execution of $\mathcal{B}$ that has more extra switches than α. Let $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$ and let $\alpha.fstate = \mathbf{x}$. We consider cases:

Case 1: α is an infinite sequence. We can write α as an infinite concatenation $\alpha_0 \frown \alpha_1 \frown \alpha_2 \ldots$, in which the execution fragments α_i with i even consist of a trajectory only, and the execution fragments α_i with i odd consist of a single discrete transition surrounded by two point trajectories.

We define inductively a sequence $\beta_0\beta_1\beta_2\ldots$ of closed execution fragments of $\mathcal{B}$ such that $\mathbf{x}\ \mathcal{R}\ \beta_0.fstate$, $\beta_0.fstate \in \Theta_{\mathcal{B}}$, and for all i, $\beta_i.lstate = \beta_{i+1}.fstate$, $\alpha_i.lstate\ \mathcal{R}\ \beta_i.lstate$, and $S_{\tau_a}(\beta) \geq S_{\tau_a}(\alpha)$. Property 1 of Definition 5.10 ensures that there exists such a $\beta_0.fstate$ because $\alpha_0.fstate \in \Theta_{\mathcal{A}}$. We use Property 3 of Definition 5.10 for the construction of the β_i's with i even. This gives us $\beta_i.ltime \leq \alpha_i.ltime$ for every even i. We use Property 2 of Definition 5.10 for the construction of the β_i's with i odd. This gives us $\beta_i.ltime = \alpha_i.ltime$ and $N(\beta_i) \geq N(\alpha_i)$ for every odd i. Let $\beta = \beta_0 \frown \beta_1 \frown \beta_2 \ldots$. Since $\beta_0.fstate \in \Theta_{\mathcal{B}}$, β is an execution for $\mathcal{B}$. Since $\beta.ltime \leq \alpha.ltime$ and $N(\beta) \geq N(\alpha)$, the required property follows.

Case 2: α is a finite sequence ending with a closed trajectory. Similar to first case.

Case 3: α is a finite sequence ending with an open trajectory. Similar to first case except that the final open trajectory τ of α is constructed using a concatenation of infinitely many closed trajectories of $\mathcal{A}$ such that $\tau = \tau_0 \frown \tau_1 \frown \ldots$. Then, working recursively, we construct a sequence $\beta_0\beta_1\ldots$ of closed execution fragments of $\mathcal{B}$ such that for each i, $\tau_i.lstate \mathcal{R} \beta_i.lstate$, $\beta_i.lstate = \beta_{i+1}.fstate$, and $\beta_i.ltime \leq \tau_i.ltime$. This construction uses induction on i, using Property 3 of Definition 5.10 in the induction step. Now, let $\beta = \beta_0 \frown \beta_1 \frown \ldots$. Clearly, β is an execution fragment of $\mathcal{B}$ and $\tau.fstate \mathcal{R} \beta.fstate$ and $\beta.ltime \leq \tau.ltime$.

■

Theorem 5.4. *If $\mathcal{A}$ and $\mathcal{B}$ are SHIOAs and $\mathcal{R}$ is a switching simulation relation from $\mathcal{A}$ to $\mathcal{B}$, then $\mathcal{A} \leq_{switch} \mathcal{B}$.*

Proof. We fix a τ_a. Given N_0 such that for every execution β of $\mathcal{B}$, $S_{\tau_a}(\beta) \leq N_0$, it suffices to show that for every execution α of $\mathcal{A}$, $S_{\tau_a}(\alpha) \leq N_0$. We fix α. From Lemma 5.3 we know that there exists a β such that $S_{\tau_a}(\beta) \geq S_{\tau_a}(\alpha)$, from which the result follows. ■

Corollary 5.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be SHIOAs. Suppose $\mathcal{R}_1$ and $\mathcal{R}_2$ be a switching forward simulation relation from $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{A}$, respectively. Then, $\mathcal{A}$ and $\mathcal{B}$ are ADT-equivalent.*

Switching simulation relations and Corollary 5.5 give us an inductive method for proving that *any* given pair of SHIOA are equivalent with respect to switching speed, that is, average dwell time. The theorem prover strategies for proving forward simulations (see, Chapter 6) can be used to partially automate switching simulation proofs. Even in such automated proofs the relation $\mathcal{R}$ has to be instantiated by the user which typically requires creativity and understanding of the SHIOAs in question (see case studies in Sections 5.6.2 and 5.6.5). An interesting related question (not addressed in this thesis) is computation of the switching simulation relation $\mathcal{R}$ from the specifications of $\mathcal{A}$ and $\mathcal{B}$.

5.5 Verifying ADT: Invariant approach

In this section we present a method for verifying ADT of SHIOAs which relies on checking invariants. Specifically, in sections 5.5.2 and 5.5.3 we use this method to verify ADT of a simple leaking gas-burner and a scale-independent hysteresis switch.

5.5.1 Transformations for ADT verification

First, we define a transformation $\texttt{Scount}(\mathcal{A}, \tau_a)$, $\tau_a > 0$, which converts a given SHIOA $\mathcal{A}$ to another SHIOA such that the latter keeps track of the number of extra switches, with respect to τ_a, in any execution. Informally, in $\texttt{Scount}(\mathcal{A}, \tau_a)$ the counter variable q increments every time there is a mode switch of $\mathcal{A}$, and the timer y reduces the count by 1 in every τ_a time by triggering the $\mathsf{decrement}$ action. Given SHIOA $\mathcal{A} = (X, Y, U, Q, \Theta, H, O, I, \mathcal{D}, \mathscr{S})$ and $\tau_a > 0$, $\texttt{Scount}(\mathcal{A}, \tau_a) = (X_1, Y_1, U_1, Q_1, \Theta_1, H_1, O_1, I_1, \mathcal{D}_1, \mathscr{S}_1)$ is defined as follows:

(a) $X_1 = X \cup \{q, y\}$, where q is a discrete variable of type $\mathbb{Z}$ and y is a continuous variable of type $\mathbb{R}_{\geq 0}$. $Y_1 = Y$ and $Z_1 = Z$.

(b) $\Theta_1 = \{(\mathbf{x}, q, y) \mid \mathbf{x} \in \Theta, q = 0, y = 0\}$

(c) $H_1 = H \cup \{\mathsf{decrement}\}$, $I_1 = I$, and $O_1 = O$.

(d) For each $\mathbf{x}, \mathbf{x}' \in Q$, $q, q' \in \mathbb{Z}$, $y, y' \in \mathbb{R}_{\geq 0}$, $a \in A \cup \{\mathsf{decrement}\}$, $((\mathbf{x}, q, y), a, (\mathbf{x}', q', y')) \in \mathcal{D}_1$, iff one of the following conditions hold:

  (1) $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D} \setminus \mathcal{M}$, $q' = q$, and $y' = y$,

  (2) $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{M}$, $q' = q + 1$, and $y' = y$,

  (3) $a = \mathsf{decrement}$, $\mathbf{x}' = \mathbf{x}$, $q' = q - 1$, $y = \tau_a$, and $y' = 0$.

  Recall that $\mathcal{M} \subseteq D$ is the set of mode switches of $\mathcal{A}$.

(e) $\mathscr{S}_1$ is obtained by changing each state model $\mathcal{S}$ to $\mathcal{S}_1$ as follows: For state model $\mathcal{S} = (X, Y, U, F, Inv, Stop) \in \mathscr{S}$, the corresponding state model $\mathcal{S}_1 \triangleq (X, Y, U, F_1, Inv, Stop_1)$, where F_1 is obtained by adding the differential equation $d(y) = 1$ to the collection of V-DAIs F, and $Stop_1 = Stop \vee (y = \tau_a)$.

The above definition assumes that $q, y \notin V$ and $\mathsf{decrement} \notin A$. If this is not the case, then the new variables and action are renamed appropriately. For any state model $\mathcal{S}_1$ of $\texttt{Scount}(\mathcal{A}, \tau_a)$, for every trajectory $\tau \in trajs(\mathcal{S}_1)$, the restriction $\tau \downarrow X$ is a trajectory of the corresponding state model of $\mathcal{A}$. Further, from $Stop_1$ and Definition 2.11, we know that along τ, $d(y) = 1$ and $(\tau \downarrow y)(t) = \tau_a$ implies $\tau.ltime = t$.

Lemma 5.6. *If τ_a is not an ADT for automaton $\mathcal{A}$, then for every $N_0 \in \mathbb{N}$ there exists a closed execution α of $\mathcal{A}$, such that $N(\alpha) > N_0 + \alpha.ltime/\tau_a$.*

Proof. Let us fix N_0. Automaton $\mathcal{A}$ does not have ADT τ_a, so we know that there exists an execution α of $\mathcal{A}$ such that $N(\alpha) > N_0 + \alpha.ltime/\tau_a$. If α is infinite, then there is a closed prefix of α that violates (5.2). If α is finite and open, then the closed prefix of α excluding the last trajectory of α violates (5.2). ∎

Theorem 5.7. *Given SHIOA $\mathcal{A}$ and $\tau_a > 0$, the following statements are equivalent:*

(a) τ_a is an ADT of $\mathcal{A}$,

(b) all closed executions of $\mathcal{A}$ satisfy Equation (5.2),

(c) there exists $N_0 \in \mathbb{N}$ such that $q \leq N_0$ is an invariant for $\texttt{Scount}(\mathcal{A}, \tau_a)$.

Proof. Equivalence of (a) and (b) follow from Definition 5.8 and Lemma 5.6.

$(b) \Rightarrow (c)$: Consider a reachable state $\mathbf{x}'$ of $\texttt{Scount}(\mathcal{A}, \tau_a)$. There exists a closed execution α' such that $\mathbf{x}' = \alpha'.lstate$. Let α be an execution of $\mathcal{A}$ "corresponding" to α'. Since $N(\alpha) \leq N_0 + \lfloor \frac{\alpha.time}{\tau_a} \rfloor$ implies $N(\alpha') \leq N_0 + \lfloor \frac{\alpha'.ltime}{\tau_a} \rfloor$, it follows that $\mathbf{x}' \lceil q \leq N_0$.

$(c) \Rightarrow (b)$: Consider a closed execution α of $\mathcal{A}$. Let α' be the "corresponding" execution of $\texttt{Scount}(A, \tau_a)$. Let $\mathbf{x}' = \alpha.lstate$, from the invariant we know that $\mathbf{x}' \lceil q \leq N_0$. From construction of $\texttt{Scount}(\mathcal{A}, \tau_a)$ we know that $N(\alpha) = N(\alpha')$ and $\alpha'.ltime = \alpha.ltime$ and therefore $\mathbf{x}' \lceil q = N(\alpha') - \lfloor \frac{\alpha'.ltime}{\tau_a} \rfloor$. It follows that $N(\alpha) - \frac{\alpha.ltime}{\tau_a} \leq N_0$.

∎

In Equation (5.2), the number N_0 can be arbitrary. Thus to show that a given τ_a is an average dwell time of an automaton, we need to show that q is bounded uniformly over all executions. For any closed execution α, $\alpha.lstate \lceil q$ equals $\lfloor S_{tau_a}(\alpha) \rfloor$. Thus, the invariant $q \leq N_0$, really corresponds to (5.2).

The transformation $\texttt{Scount}(,)$ is acceptable for asymptotic stability, but it does not guarantee uniform stability. For uniform stability we want all reachable execution fragments of $\mathcal{A}$ to satisfy Equation (5.2). Consider an execution α of $\mathcal{A}$ such that $\alpha = \alpha_0 \frown \alpha_1$, where $\alpha_0.ltime = t_1$, and $\alpha.ltime = t_2$. Let $N(\alpha_1)$ and $S_{\tau_a}(\alpha_1)$ denote the number of mode switches and the number of extra mode switches (w.r.t. τ_a) over the execution fragment α_1. For α_1 to satisfy (5.2), we require that

$$N(\alpha_1) \leq N_0 + \frac{t_2 - t_1}{\tau_a}, \text{ or } \quad S_{\tau_a}(\alpha_1) \leq N_0,$$

which is not guaranteed by the invariant $q \leq N_0$. This is because it is possible for q to become negative and then rapidly return to zero, all the time being less than N_0. For uniform stability we need to show that the total change in q between any two reachable states is bounded by N_0. So, we introduce an additional variable q_{min} which stores the magnitude of the smallest value ever attained by q. We call the resulting transformation $\texttt{Smin}(,)$. Instead of introducing the new variable q_{min} we could restrict the variable q to have only non-negative values, to obtain uniform stability[1].

[1] We thank Andy Teel for suggesting this alternative formulation.

Theorem 5.8. *Given SHIOA $\mathcal{A}$ and $\tau_a > 0$ the following two statements are equivalent:*

(a) all reachable execution fragments of $\mathcal{A}$ have ADT τ_a

(b) $q - q_{min} \leq N_0$ is an invariant for $\texttt{Smin}(\mathcal{A}, \tau_a)$*.*

Proof. According to Lemma 5.6 it suffices to consider closed execution fragments only.

$(b) \Rightarrow (a)$: Consider a reachable closed execution fragment α of $\mathcal{A}$. Since α is a reachable execution fragment, there exists an execution α_0 such that $\alpha.fstate = \alpha_0.lstate$. Let $\beta = \alpha_0 \frown \alpha$, $t_1 = \alpha_0.ltime$ and $t_2 = \beta.ltime$. Let α' and β' be the execution (fragment) of $\texttt{Smin}(\mathcal{A}, \tau_a)$ "corresponding" to α and β of $\mathcal{A}$. Suffices to show that the number of extra switches over α is bounded by N_0. Since the counter q keeps track of the number of extra switches, in turn, it suffices to show that $\beta(t_2) \lceil q - \beta(t_1) \lceil q_1 \leq N_0$. Based on whether or not q_{min} changes over the interval over the execution fragment α we consider two cases:

Case 1: q_{min} does not change over α. Then, $\beta'(t_1) \lceil q_{min} = \beta'(t_2) \lceil q_{min} = \beta'(t_{min}) \lceil q$, for some $t_{min} < t_1$.

$$\begin{aligned}
\beta'(t_2) \lceil q - \beta'(t_1) \lceil q &= [\beta'(t_2) \lceil q - \beta'(t_{min}) \lceil q] - [\beta'(t_1) \lceil q - \beta'(t_{min}) \lceil q] \\
&\leq [\beta'(t_2) \lceil q - \beta'(t_{min}) \lceil q] \quad \text{[as the second term is positive.]} \\
&\leq [\beta'(t_2) \lceil q - \beta'(t_2) \lceil q_{min}] \leq N_0,
\end{aligned}$$

because $\beta'(t_2)$ satisfies the invariant.

Case 2: q_{min} changes in α. Then, there exists some $t_{min} \in [t_1, t_2]$, such that $\beta'(t_2) \lceil q_{min} = \beta'(t_{min}) \lceil q < \beta'(t_1) \lceil q_{min}$, and

$$\begin{aligned}
\beta'(t_2) \lceil q - \beta'(t_1) \lceil q &= [\beta'(t_2) \lceil q - \beta'(t_{min}) \lceil q] + [\beta'(t_{min}) \lceil q - \beta'(t_1) \lceil q] \\
&\leq [\beta'(t_2) \lceil q - \beta'(t_{min}) \lceil q] \quad \text{[as the second term is negative.]} \\
&\leq [\beta'(t_2) \lceil q - \beta'(t_2) \lceil q_{min}] \leq N_0,
\end{aligned}$$

because $\beta'(t_2)$ satisfies the invariant.

$(a) \Rightarrow (b)$: Let $\mathbf{x}'$ be a reachable state, and ζ' be a closed execution of $\texttt{Smin}(A, \tau_a)$, such that $\mathbf{x}' = \zeta'.lstate$. Let ζ be the "corresponding" execution of $\mathcal{A}$, and t_{min} be the intermediate point where q attains its minimal value over ξ. That is, $\zeta'(t) \lceil q_{min} = \zeta'(t_{min}) \lceil q$. Since ζ is a reachable execution fragment of $\mathcal{A}$, it satisfies Equation (5.2), and we have: $N(t, t_{min}) \leq N_0 + \frac{t - t_{min}}{\tau_a}$. Rewriting,

$$\begin{aligned}
N(t, 0) \quad &- \quad N(t_{min}, 0) \leq N_0 + \frac{t - t_{min}}{\tau_a} \\
\left[\zeta'(t) \lceil q - \zeta'(0) \lceil q + \frac{t}{\tau_a}\right] \quad &- \quad \left[\zeta'(t_{min}) \lceil q - \zeta'(0) \lceil q + \frac{t_{min}}{\tau_a}\right] \leq N_0 + \frac{t - t_{min}}{\tau_a}, \\
[\zeta'(t) \lceil q - \zeta'(0) \lceil q] \quad &- \quad [\zeta'(t_{min}) \lceil q - \zeta'(0) \lceil q] \leq N_0.
\end{aligned}$$

Thus, we obtain $\zeta'(t) \lceil q - \zeta'(t_{min}) \lceil q \leq N_0$. By assumption, $\zeta'(t_{min}) \lceil q = \zeta'(t) \lceil q_{min}$, and we get $\zeta'(t) \lceil q - \zeta'(t) \lceil q_{min} \leq N_0$, that is, $\mathbf{x}' \lceil q - \mathbf{x}' \lceil q_{min} \leq N_0$.

∎

The above transformations yield a method for verifying ADT properties of SHIOAs, which are properties of executions, by checking invariant properties of transformed SHIOAs. Several tools and techniques exist for checking invariant properties of hybrid systems. Several classes of linear and rectangular hybrid systems have been identified that are amenable to automatic reachable set computation. We refer the reader back to Section 4.1 for an overview. For these classes, invariant properties and therefore ADT properties can also be verified automatically using the software tools, such as HyTech [HHWT97] and PHAVer [Fre05], that have been developed for computing reachable sets. In order to verify ADT properties of hybrid systems that fall outside these classes, the corresponding invariant properties can be verified using the deductive techniques[2] of Chapter 4. In the next two Sections, we illustrate invariant-based ADT verification for representative systems from both these classes.

5.5.2 Case Study: Leaking Gas-burner

Our first case illustrate this with the toy leaking gas-burner system from [AHH93]. The Burner SHIOA (Figure 5-1) specifies a leaking gas-burner. The system operates in two modes, namely, normal and leaking. In each mode the reset timer x increases at the same rate as real-time. The Burner switches between these two modes according to the following rules: Every leak continues for D_2 seconds after which it is repaired and the system returns to the normal mode; no leak occurs within the next D_1 seconds of the previous leak; $D_2 < D_1$. Mode switches are brought about by the leak and repair actions. Indeed, it is easy to see that the ADT of this SHIOA is $\frac{D_1+D_2}{2}$.

```
automaton Burner(D1, D2 : Real) where D2 < D1
  type
    Mode = Enumeration [normal, leaking]

  signature
    internal leak, repair

  variables
    internal mode : Mode := normal;
    x : Real := 0; z : Real;

  transitions
    internal leak
      pre mode = normal ∧ x ≥ D1;
      eff mode := leaking; x := 0;
    internal repair
      pre mode = leaking ∧ x = D2;
      eff mode := leaking; x := 0;

  trajectories
    trajdef normal
      evolve d(x) = 1;

    trajdef leaking
      invariant x ≤ D2;
      stop when x = D2;
      evolve d(x) = 1;
```

Figure 5-1: Leaking gas burner.

In order to check whether a given $\tau_a > 0$ is an average dwell time for Burner we derive the transformed automaton $\mathcal{B} = \texttt{Scount}(\text{Burner}, \tau_a)$. The resulting SHIOA $\mathcal{B}$ is rectangular and initialized and therefore amenable to automatic reachability analysis though existing model checking tools. We used the HyTech tool [HHWT97] to check if the $q \leq N_0$ is an invariant property of the transformed automaton, for particular choice of N_0. For $D_1 = 20, D_2 = 4, N_0 = 1000$ and for different values of τ_a, we check if $q \leq N_0$ is an invariant for the transformed Burner. HyTech tells us that $q \leq N_0$ is indeed an invariant for $\tau_a \leq 12$. It follows that the ADT of Burner with the above parameters is 12.

[2] In Chapter 6 we shall see how invariant proofs in general can be partially automated using mechanical theorem provers.

5.5.3 Case Study: Scale-independent Hysteresis Switch

We verify the ADT property of a scale-independent hysteresis switch unit which is a subsystem of an adaptive supervisory control system taken from [HLM03] (also Chapter 6 of [Lib03]). We will use the invariant-based approach for verifying stability. The proof of the final invariant, as we shall see shortly, uses a sequence of related, albeit simpler, invariants. The invariant proof technique of Chapter 4 is used to prove these intermediate invariants. The ADT property of this switching logic unit guarantees stability of the overall supervisory control system. The above references also present a proof of this property by a different approach.

Let $I = \{1, \ldots, m\}$, $m \in \mathbb{N}$, be the index set for for a family of controllers. An adaptive supervisory controller consists of a family of candidate controllers $u_i, i \in I$, which correspond to the parametric uncertainty range of the plant in a suitable way. Such a controller structure is particularly useful when the parametric uncertainty is so large that robust control design tools are not applicable. The supervisory controller operates in conjunction with a set of on-line estimators that provide *monitoring signals* $\mu_i, i \in I$. Intuitively, smallness of μ_i indicates high likelihood that i is the actual parameter value. Based on these signals, the switching logic unit changes the variable *mode*, which in turn determines the controller to be applied to the plant. Now, if the supervisory controller switches to u_i whenever μ_i is the smallest monitoring signal, then there will be *chattering* or very rapid switching between the controllers, leading to instability. This is avoided by implementing a scale-independent hysteresis based switching logic.

System Specification

The HystSwitch SHIOA (Figure 5-2) specifies a scale-independent hysteresis switch where the monitoring signals are generated by differential equations of the form, $d(\mu) = f_i(\mu)$, where $i \in I$, and μ is the vector of monitoring signals. The formal parameters of HystSwitch include the initial mode i_0, the hysteresis constant h, and a constant C_0 which bounds the initial value of the monitoring signals. The only actions of HystSwitch are the switch actions which bring about the switches between the modes. Conceptually, the mode i, $i \in I$, corresponds to the ith controller being applied to the plant—although this is not captured in HystSwitch. The variable μ models an array of m real-valued monitoring signals, indexed by I. We shall denote the ith monitoring signal $\mu[i]$ as μ_i. All the monitoring signals are initialized to be greater then C_0. The transitions of HystSwitch implement the following switching logic: at an instant of time when $mode = i$ for some $i \in \{1, \ldots, m\}$, if there exists a $j \in \{1, \ldots, m\}$ such that $\mu_j(1+h) \leq \mu_i$, then the switching logic sets $mode = j$. HystSwitch has a collection of state models parameterized by I. The state model $\mathsf{mode}(i), i \in I$, describes the evolution of the monitoring signals in mode i. The evolve clause states a collection of differential equations in the vector notation (μ is a m-vector, f_i is a vector of m functions).

Since the evolution of the monitoring signals are specified in terms of nonlinear differential equations, they are not initialized with every mode switch and there are arbitrary number of modes, we cannot apply automatic reachability analysis to HystSwitch. Instead, we transform automaton HystSwitch to $\texttt{Scount}(\mathsf{HystSwitch}, \tau_a)$ and employ the techniques of Chapter 4 to prove a sequence of invariants. These invariants together with Theorem 5.7 establish that the ADT of HystSwitch is at least $\tau_a = \frac{\log(1+h)}{m\lambda}$.

For the ease of analysis we introduce several history variables to $\texttt{Scount}(\mathsf{HystSwitch}, \tau_a)$.

```
1  automaton HystSwitch(I : type, i0 : I, h : Real, C0 : Real) where h, C0 > 0
2    signature
3      internal switch(i, j : I) where i ≠ j
4
5    variables
6      internal mode : I := i0; μ : Array[I, Real];
7        initially ∀i : I, μ[i] ≥ C0
8
9
10
11   transitions
12     internal switch(i, j) where i ≠ j
13       pre mode = i ∧ (1 + h)μ[j] ≤ μ[i]
14       eff mode := j
15
16   trajectories
17     trajdef mode(i : I)
18       invariant mode = i;
19       stop when ∃j : I, (1 + h)μ[j] ≤ μ[i];
20       evolve d(μ) = f_i(μ);
```

Figure 5-2: Scale-independent hysteresis switch.

The result is the automaton TRHSwitch of Figure 5-3. Lines 8, lines 24–26, and lines 32–33 correspond to the transformation of Section 5.5.1. Note that the timer y is not reset to 0 after every decrement and it records the total time elapsed. The following additional variables are introduced: (1) c is a Nat valued variable that counts the total number of mode switches, (2) $c[]$ is an array of m numbers indexed by I; $c[i]$ counts the number of switches to mode i, (3) $\mu[,]$ is a 2-dimensional array of $\mathbb{R} \cup \{\perp\}$ indexed by I and Nat; $\mu[i, r]$ stores the value of $\mu[i]$ at the instant when *mode* becomes equal to i for the rth time. (4) k is a Nat valued variable that measures the time of occurrence of the next decrement action in multiples of τ_a. Initially, $\mu[i, 0] = \mu[i]$, for all $i \in I$, $\mu[i_0, 1] = \mu[i_0]$, where $i_0 \in I$ is the initial mode; the rest of the $\mu[i, r]$s are set to $\perp$. In the remainder of this section, we shall denote $\mu[i, r]$ and $c[i]$ by μ_i^r and c_i, respectively.

```
1  automaton TRHSwitch(I : type, i0 : I, h, τa, C0 : Real) where h, C0, τa > 0
2    signature
3      internal switch(i, j : I) where i ≠ j
4      internal decrement
5
6    variables
7      internal mode : I := i0; μ : Array[I, Real];
8        q : Int := 0; y : Real := 0;
9        c : Nat := 0; k : Nat := 0;
10       μ : Array[I, Nat, Null[Real]]; c : Array[I, Nat]
11         initially ∀i : I, μ[i] ≥ C0,
12           ∀i : I, (i = i0 ∧ c[i] = 1) ∨ (i ≠ i0 ∧ c[i] = 0)
13           ∀i : I, k : Nat, (k = 0 ∧ μ[i, 0] = μ[i])
14             ∨(k ≠ 0 ∧ μ[i, k] = ⊥),
15     let μmin = min_{i:I}{μ[i]}
16
17
18   transitions
19     internal switch(i, j : I) where i ≠ j
20       pre mode = i ∧ (1 + h)μ[j] ≤ μ[i];
21       eff mode := j; q := q + 1;
22         c := c + 1; c[j] := c[j] + 1; μ[j, c_j] = μ[j];
23
24     internal decrement
25       pre y = (k + 1)τa
26       effect q := q − 1; k := k + 1;
27
28   trajectories
29     trajdef mode(i : I)
30       invariant mode = i;
31       stop when ∃j : I, (1 + h)μ[j] ≤ μ[i]
32         ∨ y = (k + 1)τa;
33       evolve d(μ) = f_i(μ); d(y) = 1;
```

Figure 5-3: Transformed hysteresis switch.

Analysis of Scale-independent Hysteresis Switch

Our analysis does not depend on the exact nature of the differential equations specified by the f_i functions. However, we require the monitoring signals for each μ_i to be continuous, monotonically nondecreasing, satisfying the following lower and upper bounds:

$$\mu_{i^*}(t) \quad \leq \quad C_1 + C_2 e^{\lambda t}, \text{for some } i^* \in I \tag{5.6}$$

where λ, C_1 and C_2 are positive constants. For simplicity of presentation, we prove the invariants required for asymptotic stability, and not uniform asymptotic stability. Accordingly the average dwell time property we get is over executions and not over execution fragments of the automaton. The first two invariants state some basic properties of TRHSwitch and they follow from straightforward applications of Lemma 4.2.

Invariant 5.9. $q \leq c - \frac{y}{\tau_a} + 1$.

Invariant 5.10. For all $i, j \in I$, if *mode* $= j$ then $\mu_j \leq (1+h)\mu_i$, in addition if $c_j > 0$ and $y_j = 0$ then $\mu_j \leq \mu_i$.

Invariant 5.11. For all $i \in I$, $c_i \geq 2 \Rightarrow \mu_i^{c_i} \geq (1+h)\mu_i^{c_i - 1}$.

Proof. Fix $i \in I$. The proof is using the induction schema laid out in Lemma 4.2. The starting condition holds vacuously because in all start states $c_i \leq 1$. Since the invariant involves only discrete variables, we have to check the transition conditions alone. Suppose $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, where $a = \mathsf{switch}(j,i)$, $\mathbf{x} \lceil \mathit{mode} = j$, and $\mathbf{x}' \lceil c_i = r+1$. That is, action a is the $(r+1)^{st}$ switch to mode i. From the code of the switch action we know that $(1+h)(\mathbf{x} \lceil \mu_i) = \mathbf{x} \lceil \mu_j$. It follows that

$$\mathbf{x}' \lceil \mu_i^{r+1} = \mathbf{x}' \lceil \mu_i = (1+h)(\mathbf{x}' \lceil \mu_j) \tag{5.7}$$

Let $\mathbf{x}''$ be the post-state of the transition which led to the rth switch to mode i. Then, From the first part of Invariant 5.10, $(1+h)(\mathbf{x}'' \lceil \mu_j) \geq \mathbf{x}'' \lceil \mu_i = \mathbf{x}'' \lceil \mu_i^r$. From monotonicity of μ_i, $\mathbf{x}' \lceil \mu_i \geq \mathbf{x}'' \lceil \mu_i$. Since $\mathbf{x}'' \lceil \mu_i^r = \mathbf{x}' \lceil \mu_i^r$ (no other switches to mode i in between), we get $\mathbf{x}' \lceil \mu_j \geq \mathbf{x}' \lceil \mu_i^r$. Combining this last inequality with Equation (5.7) we get, $\mathbf{x}' \lceil \mu_i^{r+1} \geq (1+h)(\mathbf{x}' \lceil \mu_i^r)$. ∎

Now we prove the main invariant, which states that for a particular choice of τ_a, the value of the variable q is bounded by some constant.

Invariant 5.12. Let $\tau_a = \frac{\log(1+h)}{\lambda m}$ and $N_0 = 2 + m + \frac{m}{\log(1+h)} \log\left(\frac{C_1 + C_2}{C_0}\right)$. In any reachable state of TRHSwitch, $q \leq N_0$.

Proof. Consider any reachable state $\mathbf{x}$. From the code of HystSwitch we observe that, for every mode switch brought about by $\mathsf{switch}(j,i)$, $i, j \in I$, both c and c_i are incremented by 1. If the total number of mode switches upto $\mathbf{x}$, $\mathbf{x} \lceil c$, is less than m then the result follows immediately from Invariant 5.9. Otherwise, $\mathbf{x} \lceil c \geq m$ and there must be some $j \in I$, such that mode j is visited more than $\lceil \frac{\mathbf{x} \lceil c-1}{m} \rceil$ times. That is, there exists $j \in I$, such that $\mathbf{x} \lceil c_j \geq \lceil \frac{\mathbf{x} \lceil c-1}{m} \rceil$. Applying Invariant 5.11, we deduce that there exists $j \in I$ such that $\mathbf{x} \lceil \mu_j^{c_j} \geq (1+h)^{\lceil \frac{\mathbf{x} \lceil c-1}{m} \rceil - 1}(\mathbf{x} \lceil \mu_j^1)$. Taking logarithm and rearranging we have,

$$\mathbf{x} \lceil c \leq 1 + m + \frac{m}{\log(1+h)} \log\left(\frac{\mathbf{x} \lceil \mu_j^{c_j}}{\mathbf{x} \lceil \mu_j^1}\right).$$

Let $\mathbf{x}'$ be the post state of the transition corresponding to the c_j^{th} switch to mode j. Then, $\mathbf{x} \lceil \mu_j^{c_j} = \mathbf{x}' \lceil \mu_j$. From the second part of Invariant 5.10 and monotonicity of the monitoring signals, it follows that, for all $k \in I$, $\mathbf{x} \lceil \mu_j^{c_j} = \mathbf{x}' \lceil \mu_j^{c_j} \leq \mathbf{x}' \lceil \mu_k \leq \mathbf{x} \lceil \mu_k$. It follows that,

for all $k \in I$,

$$\mathbf{x} \lceil c \leq 1 + m + \frac{m}{\log(1+h)} \log\left(\frac{\mathbf{x} \lceil \mu_k}{\mathbf{x} \lceil \mu_j^1}\right).$$

Form monotonicity and initialization of the monitoring signals, $\mu_j^1 \geq \mu_j^0 \geq C_0$. Therefore, for all $k \in I$,

$$\begin{aligned}
\mathbf{x} \lceil c &\leq 1 + m + \frac{m}{\log(1+h)} \log\left(\frac{\mathbf{x} \lceil \mu_k}{C_0}\right). \\
&\leq 1 + m + \frac{m}{\log(1+h)} \log\left(\frac{C_1 + C_2 e^{\lambda\ (\mathbf{x} \lceil y)}}{C_0}\right), \quad \text{replacing } k \text{ with } i^* \text{ of (5.6)} \\
&\leq 1 + m + \frac{m}{\log(1+h)} \log\left(\frac{C_1 + C_2}{C_0}\right) + \frac{\lambda m\ (\mathbf{x} \lceil y)}{\log(1+h)}
\end{aligned}$$

Using Invariant 5.9, and putting $\tau_a = \frac{\log(1+h)}{\lambda m}$, we get the result. ∎

From the above invariant and Theorem 5.7 it is established that HystSwitch has an average dwell time of at least $\frac{\log(1+h)}{\lambda m}$. The following property of the switch is a consequence of the ADT property and the assumptions on the monitoring signals, and it states the desirable property in terms of switches between controllers.

Theorem 5.13. *If there exists an index $i \in I$ such that the monitoring signal μ_i is bounded then the the switching between controllers stop in finite time at some index $j \in I$ and μ_j is bounded.*

To ensure stability of the overall supervisory control system, the parameters h and λ must be such that this average dwell time satisfies the inequality of Theorem 5.2.

5.6 Verifying ADT: Optimization-based Approach

In this section we develop a second method for verifying ADT properties. From Definition 5.8 it follows that $\tau_a > 0$ is *not* an ADT of a given SHIOA $\mathcal{A}$ if and only if, for every $N_0 > 0$ there exists a reachable execution fragment α of $\mathcal{A}$ such that $S_{\tau_a}(\alpha) > N_0$. If we solve the following optimization problem:

$$\mathsf{OPT}(\tau_\mathsf{a}): \qquad \alpha^* \in \arg \max_{\alpha \in \mathsf{Execs}_{\mathcal{A}}} S_{\tau_a}(\alpha),$$

and the optimal value $S_{\tau_a}(\alpha^*)$ turns out to be bounded, then we can conclude that $\mathcal{A}$ has ADT τ_a. Otherwise, if $S_{\tau_a}(\alpha^*)$ is unbounded then we can conclude that τ_a is not an ADT for $\mathcal{A}$. In fact, any execution α that gives an unbounded value of $\mathsf{OPT}(\tau_a)$ would serve as a counterexample execution violating the average dwell time property. The optimization problem $\mathsf{OPT}(\tau_a)$, however, may not be directly solvable because, among other things, the executions of $\mathcal{A}$ may not have finite descriptions. In the remainder of this chapter we study particular classes of SHIOA for which $\mathsf{OPT}(\tau_a)$ can be formulated and solved efficiently.

5.6.1 One-clock Initialized SHIOA

Recall the definition of initialized SHIOAs from Section 2.15. Here we consider a special class of initialized SHIOA, called *one-clock initialized SHIOA*. As the name suggests, such automata have a single clock variable which is reset at every mode switch. Clearly, reachability is decidable for one-clock initialized SHIOAs and therefore, we can apply the invariant based technique of Section 5.5 and model-checking to automatically verify ADT properties. Yet we apply our optimization-based ADT verification technique to one-clock initialized SHIOAs because, as we shall see shortly, $\mathsf{OPT}(\tau_a)$ can be solved efficiently using classical graph algorithms for this class. In Section 5.6.3, we consider the case of general initialized SHIOAs.

A weighted directed graph uniquely defines a one-clock initialized SHIOA. Consider a directed graph $G = (\mathcal{V}, \mathcal{E}, w, e_0)$, where (1) $\mathcal{V}$ is a finite set of vertices, (2) $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed edges, (3) $w : \mathcal{E} \to \mathbb{R}_{\geq 0}$ is a cost function for the edges, and (4) $e_0 \in \mathcal{E}$ is a special *start edge*. The cost of a path in G is the sum of the costs of the edges in the path. Given the graph G, the corresponding one-clock initialized SHIOA $\mathtt{Aut}(G)$ is specified by the HIOA -like code in Figure 5-4. The source and the target vertices of an edge e are denoted by $e[1]$ and $e[2]$, respectively.

---

automaton $Aut(G)$ **where** $G = (\mathcal{V}, \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}, w : \mathcal{E} \to \mathbb{R}_{\geq 0}, e_0 \in \mathcal{E})$
  signature
    internal switch$(e, e' : \mathcal{E})$ **where** $e \neq e'$

  variables
    internal $mode : \mathcal{E} := e_0$; x : Real $:= 0$;

  transitions
    internal switch(e, e')
      pre $mode = e \wedge e[2] = e'[1] \wedge x = w(e)$;
      eff $mode := e'$; $x := 0$

  trajectories
    trajdef edge$(e : \mathcal{E})$
      invariant $mode = e \wedge x \leq w(mode)$;
      stop when $x = w(mode)$;
      evolve $d(x) = 1$;

Figure 5-4: One-clock initialized SHIOA $\mathtt{Aut}(G)$ defined by directed graph G.

Intuitively, the state of $\mathtt{Aut}(G)$ captures the motion of a particle moving with unit speed along the edges of the graph G. When the particle in on edge e, its *mode* is said to be e. The value of the state variable x is the distance of the particle from the source vertex of its current mode. A switch from mode e to mode e' corresponds to the particle arriving at vertex $e[2]$ via edge e, and departing on edge e'. Within edge e the particle moves at unit speed from $e[1]$, where $x = 0$ to $e[2]$, where $x = w(e)$.

The next theorem implies that in order to search for an execution of $\mathtt{Aut}(G)$ that maximizes $\mathsf{OPT}(\tau_a)$, it is necessary and sufficient to search over the space of the cycles of G.

Theorem 5.14. *Consider $\tau_a > 0$ and a one-clock initialized SHIOA $\mathtt{Aut}(G)$. $\mathsf{OPT}(\tau_a)$ for $\mathtt{Aut}(G)$ is bounded if and only if for all $m > 1$, the cost of any reachable cycle of G with m segments is at least $m\tau_a$.*

Proof. It is easy to see that if there is a cycle of G, $\beta = v_0 e_1 v_1 \ldots e_m v_m$, such that the cost $\sum_{i=1}^{m} w(e_i) < m\tau_a$, then $\mathsf{OPT}(\tau_a)$ is unbounded. Since β is a cycle with $v_0 = v_m$, we can construct an execution γ of $\mathtt{Aut}(G)$ by concatenating $\beta \frown \beta \frown \beta \ldots$, k times. Therefore, the total number of extra mode switches in γ is $S_{\tau_a}(\gamma) = N(\gamma) - \frac{\gamma.ltime}{\tau_a} = km - \frac{k}{\tau_a} \sum_{i=1}^{m} w(e_i)$ $= \frac{k}{\tau_a}(m\tau_a - \sum_{i=1}^{m} w(e_i))$. If $m\tau_a > \sum_{i=1}^{m} w(e_i)$, then the right hand side can be made arbitrarily large by increasing k.

Next, suppose $\mathsf{OPT}(\tau_a)$ is unbounded for $\mathtt{Aut}(G)$. We choose N_0 to be larger than the number of vertices $|\mathcal{V}|$ of G. Let β be the shortest execution of $\mathtt{Aut}(G)$ with more than N_0 extra switches. Suppose the length of β is l. Since $S_{\tau_a}(\beta) > N_0$, $l - \frac{1}{\tau_a}\sum_{i=1}^{l} w_i > N_0$. Since N_0 is larger than the number of vertices $\mathtt{Aut}(G)$, some of the vertices must be repeated in β. That is, β must contain a cycle. Suppose $\beta = \beta_p.\gamma.\beta_s$, where γ is cycle, and let l_1, l_2, l_3 be the lengths of β_p, γ, and β_s, respectively. Then,

$$l_1 + l_2 + l_3 > N_0 + \frac{1}{\tau_a}\sum_{i=1}^{l_1} w_i + \frac{1}{\tau_a}\sum_{i=1}^{l_2} w_i + \frac{1}{\tau_a}\sum_{i=1}^{l_3} w_i$$

For the sake of contradiction we assume that the cost of the cycle γ, $\sum_{i=1}^{l_2} w_i \geq l_2\tau_a$. Therefore,

$$l_1 + l_3 > N_0 + \frac{1}{\tau_a}\left[\sum_{i=1}^{l_1} w_i + \sum_{i=1}^{l_3} w_i\right] \tag{5.8}$$

From Equation (5.8), $S_{\tau_a}(\beta_p \frown \beta_s) > N_0$, and we already know that $\beta_p \frown \beta_s$ is shorter that β, which contradicts our assumption that β is the shortest execution with more than N_0 extra switches. ■

Corollary 5.15. *Suppose $\mathcal{A}$ is a one-clock initialized SHIOA such that $\mathcal{A} = \mathtt{Aut}(G)$, where G is a weighted directed graph. Any $\tau_a > 0$ is an average dwell time of $\mathcal{A}$ if and only if the mean-cost of any reachable cycle of G is at least τ_a.*

Proof. Follows from Definition 5.8 and Theorem 5.14. ■

The problem of solving $\mathsf{OPT}(\tau_a)$ for $\mathtt{Aut}(G)$ reduces to checking whether G contains a cycle of length m, for any $m > 1$, with cost less than $m\tau_a$. This is the well known mean-cost cycle problem for directed graphs and can be solved in $O(|\mathcal{V}||\mathcal{E}|)$ time using Bellman-Ford algorithm or Karp's minimum mean-weight cycle algorithm [CLR90].

5.6.2 Case Study: Linear Hysteresis Switch

Consider a linear version of the HystSwitch automaton of Figure 5-2. Here the monitoring signals are generated by linear differential equations: for each $i \in I$, $d(\mu_i) = c_i\mu_i$ if $mode = i$, otherwise $d(\mu_i) = 0$; $c_i, i \in I$, is a positive constant. The switching logic unit implements the same scale independent hysteresis switching as in HystSwitch. The resulting SHIOA, LinHSwitch, is shown in Figure 5-7, and we are interested to verify its ADT properties independent of Invariant 5.12. The LinHSwitch automaton is not a one-clock initialized SHIOA because the continuous variables $\mu[i], i \in I$, are not reset by the switch actions. So we cannot apply Theorem 5.14 to verify its ADT directly. The switching behavior of LinHSwitch, however, does not depend on the value of the μ_i's but only on the ratio of $\frac{\mu_i}{\mu_{min}}$, which is always within $[1, (1+h)]$. Specifically, when LinHSwitch is in mode i, all the ratios remain constant, except $\frac{\mu_i}{\mu_{min}}$. The ratio $\frac{\mu_i}{\mu_{min}}$ increases monotonically from 1 to either $(1+h)$ or to $(1+h)^2$, in time $\frac{1}{c_i}\ln(1+h)$ or $\frac{2}{c_i}\ln(1+h)$, respectively.

Based on this observation, we will first show that there exists a one-clock initialized automaton $\mathcal{B}$, such that LinHSwitch $\leq_{switch}$ $\mathcal{B}$, using a switching simulation relation of Section 5.4. Next we apply Corollary 5.15 to $\mathcal{B}$ and verify that ADT of $\mathcal{B}$ is at least τ_a,

```
automaton LinHSwitch(I : type, i_0 : I, h : Real, c : Array[I, Real]) where h ≥ 0
  signature
    internal switch(i, j : I) where i ≠ j

  variables
    internal mode : I := i_0; μ : Array[I, Real];
        initially ∀i : I, (i = i_0 ∧ μ[i] = (1 + h)C_0)
          ∨(i ≠ i_0 ∧ μ[i] = C_0)
    let μ_min := min_{i:I}{μ[i]}

  transitions
    internal switch(i, j)
      pre mode = i ∧ (1 + h)μ[j] ≤ μ[i];
      eff mode := j;

  trajectories
    trajdef mode(i : I)
      invariant mode = i;
      stop when ∃j : I, (1 + h)μ[j] ≤ μ[i];
      evolve ∀ j : I, (j = i ∧ d(μ[j]) = c[j]μ[j])
          ∨(j ≠ i ∧ d(μ[j]) = 0);
```

Figure 5-5: Linear hysteresis switch.

where τ_a is a constant derived from the coefficients c_i, $i \in I$. Then it follows that, the ADT of LinHSwitch is also at least τ_a, by Definition 5.9.

We begin by constructing the abstract automaton $\mathcal{B}$. Consider a graph $G = (\mathcal{V}, \mathcal{E}, w, e_0)$, where:

1. $\mathcal{V} \subset \{1, (1+h)\}^m$, such that for any $v \in V$, all the m-components are not equal. We denote the ith component of $v \in V$ by $v[i]$.

2. An edge $(u, v) \in \mathcal{E}$ if and only if, one of the following conditions hold:
   (a) There exists $j \in \{1, \ldots, m\}$, such that, $u[j] \neq v[j]$ and for all $i \in \{1, \ldots, m\}$, $i \neq j$, $u[i] = v[i]$. The cost of the edge $w(u, v) := \frac{1}{c_j} \ln(1+h)$ and we define $\zeta(u, v) := j$.
   (b) There exists $j \in \{1, \ldots, m\}$ such that $u[j] = 1, v[j] = (1+h)$ and for all $i \in \{1, \ldots, m\}$, $i \neq j$ implies $u[i] = (1+h)$ and $v[i] = 1$. The cost of the edge $w(u, v) := \frac{2}{c_j} \ln(1+h)$ and we define $\zeta(u, v) := j$. The ith component of the source (destination) vertex of edge e is denoted by $e[1][i]$ ($e[2][i]$, respectively).

3. $e_0 \in \mathcal{E}$, such that $e_0[1][i_0] = (1+h)$ and for all $i \neq i_0$, $e_0[1][i] = 1$.

The graph G_3 for $I = \{1, 2, 3\}$ is shown in Figure 5.6.2. $\mathtt{Aut}(G)$ is the one-clock initialized automaton that is ADT-equivalent to LinHSwitch. We prove this in Lemma 5.16. But before we give the simulation proof we discuss the main idea of execution correspondence. A typical execution $\alpha = \tau_0, a_1, \tau_1, a_2, \tau_2$ of LinHSwitch is as follows: τ_0 is a point trajectory that maps to the state $(mode = 1, [\mu_1 = (1+h)C_0, \mu_2 = C_0, \mu_3 = C_0])$, $a_1 = \mathsf{switch}(1, 3)$, $\tau_1.dom = [0, \frac{1}{c_3} \ln(1+h)]$, $(\tau_1 \downarrow \mu_3)(t) = C_0 e^{c_3 t}$, $a_2 = \mathsf{switch}(3, 2)$, $\tau_2.dom = [0, \frac{2}{c_2} \ln(1+h)]$, $(\tau_2 \downarrow \mu_2)(t) = C_0 e^{c_2 t}$. Note that each edge e of G corresponds to a mode of LinHSwitch; this correspondence is captured by the ζ function in the definition of G.

We define a relation $\mathcal{R}$ on the state spaces on $\mathcal{A} =$ LinHSwitch and $\mathcal{B} = \mathtt{Aut}(G)$. This relation essentially scales the monitoring signals in LinHSwitch by an appropriate factor and equates them with the variable x of $\mathtt{Aut}(G)$. The switching pattern of LinHSwitch is governed by the multiplicative hysteresis constant h and is independent of this scaling.

Definition 5.11. For any $\mathbf{x} \in Q_{\mathcal{A}}$ and $\mathbf{y} \in Q_{\mathcal{B}}$, $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ if and only if:

1. $\zeta(\mathbf{y} \lceil mode) = \mathbf{x} \lceil mode$

2. For all $j \in \{1, \ldots, n\}$,
   (a) $\frac{\mathbf{x} \lceil \mu_j}{\mathbf{x} \lceil \mu_{min}} = e^{c_j (\mathbf{y} \lceil x)}$, if $j = \zeta(\mathbf{y} \lceil mode)$,
   (b) $\frac{\mathbf{x} \lceil \mu_j}{\mathbf{x} \lceil \mu_{min}} = (\mathbf{y} \lceil mode)[k][j]$, $k \in \{1, 2\}$.

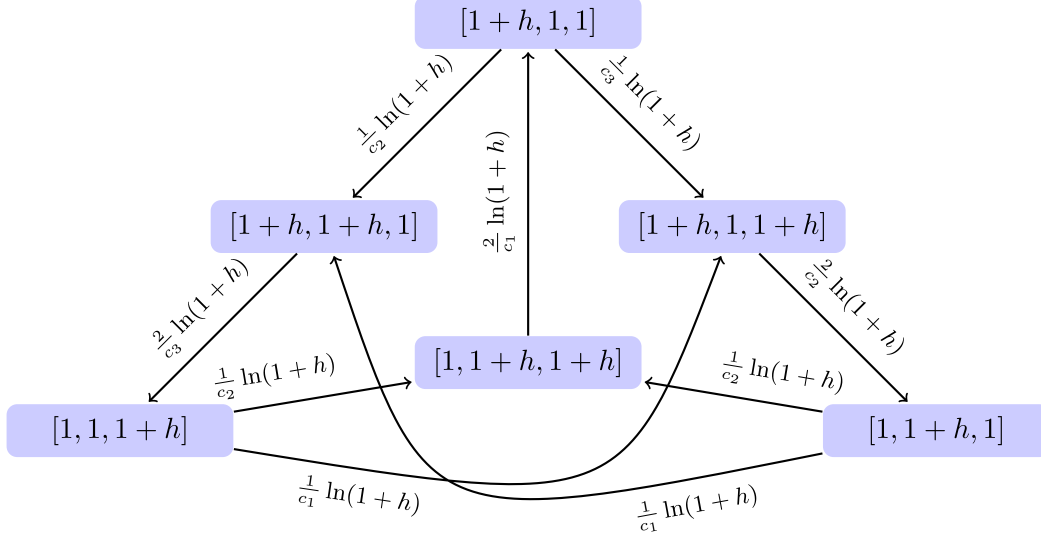

Figure 5-6: ADT-equivalent graph ($m = 3$) for LinHSwitch.

Part 1 of Definition 5.11 states that if $\mathcal{A}$ is in *mode* j and $\mathcal{B}$ is in mode e, then $\zeta(e) = j$. Part 2 states that for all $j \neq \zeta(e)$, the jth component of $e[1]$ and $e[2]$ are the same, and are equal to μ_j/μ_{min}, and for $j = \zeta(e)$, $\mu_j = \mu_{min}e^{c_j x}$.

Lemma 5.16 states that $\mathcal{R}$ is a switching simulation relation from $\mathcal{A}$ and $\mathcal{B}$. The proof follows the typical pattern of simulation proofs. We show by a case analysis that every action and state model of automaton $\mathcal{A}$ can be simulated by an execution fragment of $\mathcal{B}$ with at least as many extra switches.

Lemma 5.16. *$\mathcal{R}$ is a switching simulation relation from $\mathcal{A}$ to $\mathcal{B}$.*

Proof. We check the conditions in Definition 5.10. At a given state $\mathbf{x}$ of $\mathcal{A}$, we say that $i \in I$ is the *unique minimum at* $\mathbf{x}$, if $\min_{j\in I}\{\mathbf{x} \lceil \mu_j\}$ is unique and $\mu_i = \arg\min_{j\in I}\{\mathbf{x} \lceil \mu_j\}$. $\mathcal{A}$ has a unique start state and it is easy to see that it is related to all the start states of $\mathcal{B}$. Next we show by cases that given any state $\mathbf{x} \in Q_{\mathcal{A}}, \mathbf{y} \in Q_{\mathcal{B}}$, $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$, and an execution fragment α of $\mathcal{A}$ starting from $\mathbf{x}$ and consisting of either a single action or a single trajectory, there exists a corresponding execution fragment β of $\mathcal{B}$, starting from $\mathbf{y}$ that satisfies the conditions required for $\mathcal{R}$ to be a switching simulation relation.

Case 1: α is a $(\mathbf{x}, \mathsf{switch}(i, j), \mathbf{x}')$ transition of $\mathcal{A}$ and i is not the unique minimum at $\mathbf{x}$ and j is not unique minimum at $\mathbf{x}'$.

We choose β to be $(\mathbf{y}, \mathsf{switch}(e, e'), \mathbf{y}')$ action of $\mathcal{B}$, where e and e' are determined by the following rules:

$$e[1][i] = 1, e[2][i] = 1+h, \forall k, k \neq i, e[1][k] = e[2][k] = \frac{\mathbf{x}\lceil\mu_k}{\mathbf{x}\lceil\mu_{min}} \quad (5.9)$$

$$e'[1][j] = 1, e'[2][j] = 1+h, \forall k, k \neq j, e'[1][k] = e'[2][k] = \frac{\mathbf{x}'\lceil\mu_k}{\mathbf{x}'\lceil\mu_{min}} \quad (5.10)$$

We have to show that $\mathsf{switch}(e, e')$ is enabled at $\mathbf{y}$; this involves showing that the three conjuncts in the precondition of the switch action of $\mathcal{B}$ are satisfied at $\mathbf{y}$. First of all, since $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ we know that $\zeta(\mathbf{y} \lceil \mathit{mode}) = \mathbf{x} \lceil \mathit{mode} = i$. Further, i is not a unique

minimum at $\mathbf{x}$, so from the definition of the edges of G it follows that:

$$(\mathbf{y} \lceil mode)[1][i] = 1, (\mathbf{y} \lceil mode)[2][i] = i + h,$$
$$\forall k, k \neq i, (\mathbf{y} \lceil mode)[1][k] = (\mathbf{y} \lceil mode)[2][k] = \frac{\mathbf{x} \lceil \mu_k}{\mathbf{x} \lceil \mu_{min}} \quad (5.11)$$

Comparing Equations (5.11) and (5.17) we conclude that $\mathbf{y} \lceil mode = e$.

Secondly, using the definitions of e, e' and $\mathcal{R}$ it follows that:

$$e[2][i] = 1 + h = \frac{\mathbf{x} \lceil \mu_i}{\mathbf{x} \lceil \mu_{min}} = \frac{\mathbf{x}' \lceil \mu_i}{\mathbf{x}' \lceil \mu_{min}} = e'[1][i] \quad (5.12)$$

The second equality holds because $\mathsf{switch}(i, j)$ is enabled at $\mathbf{x}$. The third equality follows from the fact that the $\mathsf{switch}(i, j)$ transition of $\mathcal{A}$ does not alter the value of the μ_k's. Likewise, we have:

$$e[2][j] = \frac{\mathbf{x} \lceil \mu_j}{\mathbf{x} \lceil \mu_{min}} = \frac{\mathbf{x}' \lceil \mu_j}{\mathbf{x}' \lceil \mu_{min}} = 1 = e'[1][j] \quad (5.13)$$

$$\forall k, k \neq j, k \neq i, e[2][k] = \frac{\mathbf{x} \lceil \mu_k}{\mathbf{x} \lceil \mu_{min}} = \frac{\mathbf{x}' \lceil \mu_k}{\mathbf{x}' \lceil \mu_{min}} = e'[1][k] \quad (5.14)$$

Combining Equations (5.12),(5.12) and (5.14) it follows that $e[2] = e'[1]$.

Finally, from the switching simulation relation $\mathcal{R}$, we know that $\mathbf{y} \lceil x = \frac{1}{c_i} \ln \frac{\mathbf{x} \lceil \mu_i}{\mathbf{x} \lceil \mu_{min}} = \frac{1}{c_i} \ln(1 + h)$. And since $\zeta(e) = i$ and i is not the unique minimum at $\mathbf{x}$, from the definition of the edge costs of G it follows that $\mathbf{y} \lceil x = w(e)$. Thus, we have shown that $\mathsf{switch}(e, e')$ is indeed enabled at $\mathbf{y}$.

Next, we have to show that $\mathbf{x}' \mathcal{R} \mathbf{y}'$. First of all, $\mathbf{x}' \lceil mode = j$ and $\mathbf{y}' \lceil mode = e'$ from the effect parts of the $\mathsf{switch}(i, j)$ and $\mathsf{switch}(e, e')$ actions, respectively. Also, $\zeta(e') = j$ from Equation (5.17). It follows that $\mathbf{x}' \lceil mode = \zeta(\mathbf{y}' \lceil mode)$. Secondly, $\frac{\mathbf{x}' \lceil \mu_j}{\mathbf{x}' \lceil \mu_{min}} = \frac{\mathbf{x} \lceil \mu_j}{\mathbf{x} \lceil \mu_{min}} = 1$, from the precondition of $\mathsf{switch}(i, j)$. Since $\mathbf{y}' \lceil x = 0$ it follows that $\frac{\mathbf{x} \lceil \mu_j}{\mathbf{x} \lceil \mu_{min}} = e^{c_j \mathbf{y}' \lceil x}$. Finally, for all $k \neq j$, again from Equation (5.17) it follows that $\frac{\mathbf{x}' \lceil \mu_k}{\mathbf{x}' \lceil \mu_{min}} = e'[1][k] = e'[2][k]$.

Case 2: α is a $(\mathbf{x}, \mathsf{switch}(i, j), \mathbf{x}')$ transition of $\mathcal{A}$ and i is the unique minimum at $\mathbf{x}$ and j is not unique minimum at $\mathbf{x}'$.

We choose β to be $(\mathbf{y}, \mathsf{switch}(e, e'), \mathbf{y}')$ action of $\mathcal{B}$, where e and e' are determined by the following rules:

$$e[1][i] = 1, e[2][i] = 1 + h, \forall k, k \neq i, e[1][k] = (1 + h), e[2][k] = 1 \quad (5.15)$$

$$e'[1][j] = 1, e'[2][j] = 1 + h, \forall k, k \neq j, e'[1][k] = e'[2][k] = \frac{\mathbf{x}' \lceil \mu_k}{\mathbf{x}' \lceil \mu_{min}} \quad (5.16)$$

The rest of the proof is similar to that of case 1.

Case 3: α is a $(\mathbf{x}, \mathsf{switch}(i, j), \mathbf{x}')$ transition of $\mathcal{A}$ and i is not the unique minimum at $\mathbf{x}$ and j is the unique minimum at $\mathbf{x}'$.

We choose β to be $(\mathbf{y}, \mathsf{switch}(e, e'), \mathbf{y}')$ action of $\mathcal{B}$, where e and e' are determined by

the following rules:

$$e[1][i] = 1, e[2][i] = 1 + h, \forall k, k \neq i, e[1][k] = e[2][k] = \tfrac{\mathbf{x}\lceil \mu_k}{\mathbf{x}\lceil \mu_{min}}$$
$$e'[1][j] = 1, e'[2][j] = 1 + h, \forall k, k \neq j, e'[1][k] = (1 + h), e'[2][k] = 1$$

The rest of the proof is similar to that of case 1.

Case 4: α is a closed trajectory τ of $\mathcal{A}$ with $(\tau \downarrow mode)(0) = i$ for some $i \in \mathscr{S}$, such that i is not unique minimum at $\tau.fstate$.
We choose β to be the trajectory τ' of $\mathcal{B}$ with $\tau'.dom = \tau.dom$ determined by the following rules. Let $\mathbf{x} = \tau.fstate, \mathbf{x}' = \tau.lstate, \mathbf{y} = \tau'.fstate$ and $\mathbf{y}' = \tau'.lstate$.

$$\begin{aligned} &(\mathbf{y} \lceil mode)[1][i] = 1, (\mathbf{y} \lceil mode)[2][i] = 1 + h, \\ &\forall k, k \neq i, (\mathbf{y} \lceil mode)[1][k] = (\mathbf{y} \lceil mode)[1][k] = \tfrac{\mathbf{x}\lceil \mu_i}{\mathbf{x}\lceil \mu_{min}}, \\ &\forall\ t \in \tau'.dom, (\tau' \downarrow x)(t) = \tfrac{1}{c_i} \ln \tfrac{\mathbf{x}\lceil \mu_i}{\mathbf{x}\lceil \mu_{min}} + t \end{aligned} \tag{5.17}$$

We first show that τ' is a valid trajectory of $\mathcal{B}$. First of all, it is easy to check that τ' satisfies the constant differential equation $d(x) = 1$ and that the *mode* of $\mathcal{B}$ remains constant. Next, we show that τ' satisfies the stopping condition "$x = w(mode)$". Suppose there exists $t \in \tau'.dom$ such that $(\tau \lceil x)(t) = w(\mathbf{x} \lceil mode)$, then $t = w(\mathbf{x} \lceil mode) - (\mathbf{y} \lceil x)$. Then,

$$\begin{aligned} \mathbf{x}' \lceil \mu_i &= \mathbf{x} \lceil \mu_i e^{c_i t} \\ \frac{1}{c_i} \ln \frac{\mathbf{x}' \lceil \mu_i}{\mathbf{x} \lceil \mu_i} &= w(\mathbf{x} \lceil mode) - (\mathbf{y} \lceil x) \\ &= \frac{1}{c_i} \ln(1+h) - (\mathbf{y} \lceil x) \quad \text{[by replacing } w(\mathbf{x} \lceil mode)\text{]} \\ &= \frac{1}{c_i} \left[\ln(1+h) - \ln \frac{\mathbf{x} \lceil \mu_i}{\mathbf{x} \lceil \mu_{min}} \right] \quad \text{[from (5.18)]} \\ \mathbf{x}' \lceil \mu_i &= (1+h)(\mathbf{x} \lceil \mu_{min}) \\ &= (1+h)(\mathbf{x}' \lceil \mu_{min}) \quad [i \text{ not unique min } \Rightarrow \mu_{min} \text{ constant over } \tau'.] \end{aligned}$$

Last equation implies that $\mathbf{x}'$ satisfies the stopping condition for *trajdef* mode(i) for automaton $\mathcal{A}$. Therefore, $t = \tau.ltime = \tau'.ltime$. Thus we have shown that τ' is a valid trajectory of automaton $\mathcal{B}$.

We show that $\mathbf{x}' \ \mathcal{R}\ \mathbf{y}'$. First, $\mathbf{x}' \lceil mode = \zeta(\mathbf{y}' \lceil mode)$ because $\mathbf{x}' \lceil mode = \mathbf{x} \lceil mode = \zeta(\mathbf{y} \lceil mode) = \zeta(\mathbf{y}' \lceil mode)$. Secondly, for all k, $k \neq i$, $\mathbf{x} \lceil \mu_i = \mathbf{x}' \lceil \mu_i$ and $(\mathbf{y} \lceil mode)[1][k] = (\mathbf{y}' \lceil mode)[1][k]$. Finally, we show that $\mathbf{x}' \lceil \mu_i = (\mathbf{x}' \lceil \mu_{min}) e^{c_i (\mathbf{y}' \lceil x)}$ by reasoning as follows:

$$\begin{aligned} \mathbf{x}' \lceil \mu_i &= (\mathbf{x} \lceil \mu_i) e^{c_i \tau.ltime} \\ &= (\mathbf{x} \lceil \mu_{min}) e^{c_i (\mathbf{y} \lceil x + \tau.ltime)} \\ &= (\mathbf{x}' \lceil \mu_{min}) e^{c_i (\mathbf{y} \lceil x + \tau'.ltime)} \end{aligned}$$

Case 5: α is a closed trajectory τ of $\mathcal{A}$ with $(\tau \downarrow mode)(0) = i$ for some $i \in \mathscr{S}$, such that

i is the unique minimum at $\tau.fstate$.

We choose β to be the trajectory τ' of $\mathcal{B}$ with $\tau'.dom = \tau.dom$ determined by the following rules. Let $\mathbf{x} = \tau.fstate$ and $\mathbf{y} = \tau'.fstate$.

$$(\mathbf{y} \lceil mode)[1][i] = 1, (\mathbf{y} \lceil mode)[2][i] = 1 + h,$$
$$\forall k, k \neq i, (\mathbf{y} \lceil mode)[1][k] = (1 + h), (\mathbf{y} \lceil mode)[2][k] = 1$$
$$\forall\ t \in \tau'.dom, (\tau' \downarrow x)(t) = \frac{1}{c_i} \ln \frac{\mathbf{x}\lceil \mu_i}{\mathbf{x}\lceil \mu_{min}} + t$$

The rest of the proof for this case is similar to that for case 4.

∎

From Theorem 5.4 it follows that SHIOA $\mathsf{LinHSwitch} \leq_{switch} \mathtt{Aut}(G)$, and therefore if τ_a is an ADT for $\mathtt{Aut}(G)$ then it is also an ADT for $\mathsf{LinHSwitch}$. As $\mathtt{Aut}(G)$ is one-clock initialized SHIOA, from the results in Section 5.6.1, we know that we can verify whether τ_a is an ADT of $\mathtt{Aut}(G)$ efficiently by finding the minimum mean cost cycle of G. If it is, we can conclude that ADT of $\mathsf{LinHSwitch}$ is also at least τ_a. In particular, for $\mathsf{LinHSwitch}$ with $m = 3, c_1 = 2, c_2 = 4$, and $c_3 = 5$, we compute the minimum mean-cost cycle. The cost of this cycle, which is also the ADT of this automaton, is $\frac{19}{40}\log(1 + h)$. We can also use Theorem 5.12 to get an estimate of the ADT of $\mathsf{LinHSwitch}$. If we plug in $\lambda = c_1 = 2$, we get that ADT of this automaton is at least $\frac{1}{6}\log(1 + h)$. The discrepancy in the two quantities is because of the fact that the mean-cost cycle analysis uses exact information about the behavior of the monitoring signals whereas the Theorem 5.12 is based on upper bound given by Equation (5.6).

5.6.3 Initialized SHIOA

In this section, we develop techniques for verifying ADT properties for the general class of initialized SHIOA. Of course, the class of initialized SHIOAs subsumes the class of one-clock initialized SHIOAs. Consequently, the techniques developed here rely on solving more general and relatively complex optimization problems.

An closed execution fragment α of an SHIOA is said to be a *cyclic fragment* if $\alpha.fstate = \alpha.lstate$. The next theorem implies that for an initialized SHIOA $\mathcal{A}$, it is necessary and sufficient to solve $\mathsf{OPT}(\tau_a)$ over the space of the cyclic fragments of $\mathcal{A}$ instead of the larger space of all execution fragments.

Theorem 5.17. *Given $\tau_a > 0$ and initialized SHIOA $\mathcal{A}$, $\mathsf{OPT}(\tau_a)$ is bounded if and only if $\mathcal{A}$ does not have any cycles with extra switches with respect to τ_a.*

Proof. For simplicity we assume that all discrete transitions of the automaton $\mathcal{A}$ are mode switches and that for any pair of modes i, j, there exists at most one action which can bring about a mode switch from i to j. Existence of a reachable cycle α with extra switches with respect to τ_a is sufficient to show that τ_a is not an ADT for $\mathcal{A}$. This is because by concatenating a sequence of α's, we can construct an execution fragment $\alpha^\frown\alpha^\frown\alpha\ldots$ with an arbitrarily large number of extra switches.

We prove by contradiction that existence of a cycle with extra switches is necessary for making $\mathsf{OPT}(\tau_a)$ unbounded. We assume that $\mathsf{OPT}(\tau_a)$ is unbounded for $\mathcal{A}$ and that $\mathcal{A}$ does not have any cycles with extra switches. By the definition of OPT, for any constant N_0 there exists an execution that has more than N_0 extra switches with respect to τ_a. Let

us choose $N_0 > |I|^3$. Of all the executions that have more than N_0 extra switches, let $\alpha = \tau_0 a_1 \tau_1 \ldots \tau_n$ be a closed execution that has the smallest number of mode switches. From α, we construct $\beta = \tau_0^* a_1 \tau_1^* \ldots \tau_n^*$, using the following two rules:

1. Each τ_i of α is replaced by: $\tau_i^* = \arg\min\{\tau.ltime \mid \tau.fstate \in R_{a_i}, \tau.lstate \in Pre_{a_{i+1}}\}$.
2. If there exists $i, j \in I$, such that $a_i = a_j$ and $a_{i+1} = a_{j+1}$, then we make $\tau_i^* = \tau_j^*$.

Claim 5.18. *The sequence β is an execution fragment of $\mathcal{A}$ and $S_{\tau_a}(\beta) > |I|^3$.*

Proof of claim: We prove the first part of the claim by showing that each application of the above rules to an execution fragment of $\mathcal{A}$ results in another execution fragment. Consider *Rule (1)* and fix i. Since $\tau_i^*.fstate \in R_{a_i}$ and $\tau_{i-1}.lstate \in Pre_{a_i}$, $\tau_{i-1}.lstate \xrightarrow{a_i} \tau_i^*.fstate$. And, since $\tau_i^*.lstate \in Pre_{a_{i+1}}$ and $\tau_{i+1}.fstate \in R_{a_{i+1}}$, we know that $\tau_i^*.lstate \xrightarrow{a_{i+1}} \tau_{i+1}.fstate$. Now for *Rule (2)*, we assume there exist i and j such that the hypothesis of the rule holds and suppose $\tau_j^* = \tau_i^* = \tau_i$. We know that even if $\tau_j^* \neq \tau_j$, the first states of both are in R_{a_j} and the last states are in $Pre_{a_{j+1}}$. Therefore, a_j matches up the states of τ_{j-1} and τ_j^* and likewise a_{j+1} matches the states of τ_j^* and τ_{j+1}.

The second part of the claim follows from the fact that each trajectory τ_i is replaced by the shortest trajectory τ_i^* from the initialization set of the previous transition R_{a_i} to the guard set of the next transition $Pre_{a_{i+1}}$. That is, for each i, $0 < i < n$, $\tau_i^*.ltime \leq \tau_i.ltime$ and therefore $\beta.ltime \leq \alpha.ltime$ and $S_{\tau_a}(\beta) > N_0 > |I|^3$.

Since $N(\beta) > |I|^3$, there must be a sequence of 3 consecutive modes that appear multiple times in β. That is, there exist $i, j \in \{1, \ldots, m\}$, and $p, q, r \in I$, such that $\tau_i^*.fstate \lceil mode = \tau_j^*.fstate \lceil mode = p$, $\tau_{i+1}^*.fstate \lceil mode = \tau_{j+1}^*.fstate \lceil mode = q$, and $\tau_{i+2}^*.fstate \lceil mode = \tau_{j+2}^*.fstate \lceil mode = r$. Then, from *Rule (2)* we know that $\tau_{i+1}^* = \tau_{j+1}^*$. In particular, $\tau_{i+1}^*.fstate = \tau_{j+1}^*.fstate$, that is, we can write $\beta = \beta_p \frown \gamma \frown \beta_s$, where γ is a cycle. Then we have the following:

$$N(\beta_p) + N(\gamma) + N(\beta_s) > N_0 + \beta_p.ltime/\tau_a + \gamma.ltime/\tau_a + \beta_s.ltime/\tau_a$$
$$N(\beta_p) + N(\beta_s) + S_{\tau_a}(\gamma) > N_0 + \beta_p.ltime/\tau_a + \beta_s.ltime/\tau_a$$
$$N(\beta_p \frown \beta_s) > N_0 + \beta_p \frown \beta_s.ltime/\tau_a \qquad [\beta_p.lstate = \beta_s.fstate]$$

The last step follows from the assumption that $S_{\tau_a}(\gamma) \leq 0$. Therefore, we have $S_{\tau_a}(\beta_p \frown \beta_s) > N_0$ which contradicts our assumption that β has the smallest number of mode switches among all the executions that have more than N_0 extra switches with respect to τ_a. ■

The following corollary allows us to limit the search for cycles with extra switches to cycles with at most $|I|^3$ mode switches. It is proved by showing that any cycle with extra switches that has more than $|I|^3$ mode switches can be decomposed into two smaller cycles, one of which must also have extra switches.

Corollary 5.19. *Suppose $\mathcal{A}$ is an initialized SHIOA with state models indexed by I. If $\mathcal{A}$ has a cycle with extra switches, then it has a cycle with extra switches that has fewer than $|I|^3$ mode switches.*

Theorem 5.20. *Suppose $\mathcal{A}$ is an initialized SHIOA with state models indexed by I. For any $\tau_a > 0$, τ_a is an ADT for $\mathcal{A}$ if an only if all cycles of length at most $|I|$ are free of extra switches.*

Proof. Follows from Corollary 5.19 and the definition of the optimization problem $\mathsf{OPT}(\tau_a)$. ∎

This theorem gives us a method for verifying ADT of initialized SHIOAs by maximizing $\mathsf{OPT}(\tau_a)$ over all cycles of length at most $|I|$. In other words, for verify ADT of initialized hybrid systems it suffices to solve the optimization problem over a much smaller set of executions than we set out with at the beginning of Section 5.6. For non-initialized SHIOA $\mathcal{A}$, the first part of Theorem 5.17 holds. That is, solving $\mathsf{OPT}(\tau_a)$ over all cycles of length at most $|I|$, if a cycle with extra switches is found, then we can conclude that τ_a is not an ADT for $\mathcal{A}$. Solving $\mathsf{OPT}(\tau_a)$ relies on formulating it as a mathematical program such that standard mathematical programming tools can be used. This is the topic of the next section.

5.6.4 MILP formulation of OPT(τ_a)

In this section, we show that the problem of solving $\mathsf{OPT}(\tau_a)$ over the cyclic executions of a rectangular SHIOA can be formulated as a Mixed Integer Linear Programming (MILP). Recall Definition 2.16, where we defined rectangular SHIOAs to have rectangular dynamics and linier guards, invariants, and reset maps.

The Rectangular automaton (Figure 5-7) shows the specification of a generic initialized rectangular SHIOA with n continuous variables, $n \in \mathbb{N}$, and $m = |I|$ state models. For the sake of avoiding clutter, the code in this figure omits the following type information for the automaton parameters. (1) I is a type parameter which serves as the index set for the state models. (2) $i_0 \in I$ is the index of the initial state model. (3) $\mathbf{x}_0$ is a real-valued n vector which serves as the initial valuation for the continuous variables. (4) For each $i, j \in I$, $G[i,j], R[i,j]$, and $A[i]$ are real-valued $n \times n$ matrices, (5) For each $i, j \in I$, $g[i,j], r[i,j], a[i]$, and $c[i]$ are real-valued n vectors. The automaton Rectangular has a single discrete variable called *mode* which takes values in the index set $I = \{1, \dots, m\}$, and a continuous variable vector $\mathbf{x} \in \mathbb{R}^n$. For any $i, j \in I$, the $\mathsf{switch}(i,j)$ action changes *mode* from i to j. The precondition and the initialization predicates of this action are given by sets of linear inequalities on the continuous variables, represented by: $G[i,j]\mathbf{x} \leq g[i,j]$ and $R[i,j]\mathbf{x} \leq r[i,j]$, respectively. For each state model $i \in I$, the invariant is stated in terms of linear inequalities of the continuous variables $A[i]\mathbf{x} \leq a[i]$. The evolve clause is given by a single differential equation $d(\mathbf{x}) = c[i]$.

```
automaton Rectangular(I, i0, x0, G, g, R, r, A, a, c)
  signature
    internal switch(i, j : I), where i ≠ j

  variables
    internal mode : I := i;
    x : ℝ^n := x0;

  transitions
    internal switch(i, j)
      pre mode = i ∧ G[i, j]x ≤ g[i, j];
      eff mode := j; x := x' where R[i, j]x' ≤ r[i, j];

  trajectories
    trajdef mode(i : I)
    invariant A[i]x ≤ a[i];
    evolve d(x) = c[i];
```

Figure 5-7: Generic rectangular initialized SHIOA.

We describe a MILP formulation $\mathsf{MOPT}(K, \tau_a)$ for finding a cyclic execution with K mode switches that maximizes the number of extra switches with respect to τ_a. If the optimal value is positive, then the optimal solution represents a cycle with extra switches

with respect to τ_a and we conclude from Corollary 5.19 that τ_a is not an ADT for $\mathcal{A}$. On the other hand, if the optimal value is not positive, then we conclude that there are no cycles with extra switches of length K. To verify ADT of $\mathcal{A}$, we solve a sequence of $\mathsf{MOPT}(K, \tau_a)$'s with $K = 2, \ldots, m^3$. If the optimal values are not positive for any of these, then we conclude that τ_a is an ADT for $\mathcal{A}$. By adding extra variables and constraints we are able to formulate a single MILP that maximizes the extra switches over all cycles with K or less mode switches, but for simplicity of presentation we discuss $\mathsf{MOPT}(K, \tau_a)$ instead of this latter formulation. The following are the decision variables for $\mathsf{MOPT}(K, \tau_a)$.

- $\mathbf{x}_u \in \mathbb{R}^n$, $u \in \{0 \ldots, K\}$, value of continuous variables
- $t_u \in \mathbb{R}$, $u \in \{0, 2, 4, \ldots, K\}$, length of uth trajectory
- $b_{uj} = \begin{cases} 1, & \text{if mode over } u^{th} \text{ trajectory is } j \\ 0, & \text{otherwise.} \quad \text{for each } u \in \{0, 2, \ldots, K\},\ j \in \{1, \ldots, m\} \end{cases}$
- $p_{ujk} = \begin{cases} 1, & \text{if mode over } (u-1)^{st} \text{ trajectory is } j \text{ and over } (u+1)^{st} \text{ trajectory is } k \\ 0, & \text{otherwise.} \quad \text{for each } u \in \{0, 2, 4, \ldots, K\}, j, k \in \{1, \ldots, m\} \end{cases}$

The objective function and the constraints are shown in Figure 5-8. In $\mathsf{MOPT}(K, \tau_a)$, an execution fragment with K mode switches is represented as a sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_K$ of K valuations for the continuous variables. For each even u, $\mathbf{x}_u$ goes to $\mathbf{x}_{u+1}$ by a trajectory of length t_u. If this trajectory is in mode j, for some $j \in \{1, \ldots, m\}$, then $b_{uj} = 1$, else $b_{uj} = 0$. For each odd u, $\mathbf{x}_u$ goes to $\mathbf{x}_{u+1}$ by a discrete transition. If this transition is from mode j to mode k, for some $j, k \in \{1, \ldots, m\}$, then $p_{ujk} = 1$, else $p_{ujk} = 0$. These constraints are specified by Equation (5.18) in Figure 5-8. For each odd u, Constraints (5.20) and (5.21)

Objective function: $\quad S_{\tau_a} : \dfrac{K}{2} - \dfrac{1}{\tau_a} \displaystyle\sum_{u=0,2,\ldots}^{K} t_u$

$$\text{Mode: } \forall\, u \in \{0, 2, \ldots, K\},\ \sum_{j=1}^{m} b_{uj} = 1 \textit{ and } \forall\, u \in \{1, 3, \ldots, K-1\},\ \sum_{j=1}^{m}\sum_{k=1}^{m} p_{ujk} = 1 \tag{5.18}$$

$$\text{Cycle: } \mathbf{x}_0 = \mathbf{x}_K \text{ and } \forall\, j \in \{1, \ldots, m\}, b_{0j} = b_{Kj} \tag{5.19}$$

$$\text{Preconds: } \forall\, u \in \{1, 3, \ldots, K-1\},\ \sum_{j=1}^{m}\sum_{k=1}^{m} G[j,k].p_{ujk}.\mathbf{x}_u \leq \sum_{j=1}^{m}\sum_{k=1}^{m} p_{ujk}.g[j,k] \tag{5.20}$$

$$\text{Initialize: } \forall\, u \in \{1, 3, \ldots, K-1\},\ \sum_{j=1}^{m}\sum_{k=1}^{m} R[j,k].p_{ujk}.\mathbf{x}_{u+1} \leq \sum_{j=1}^{m}\sum_{k=1}^{m} p_{ujk}.r[j,k] \tag{5.21}$$

$$\text{Invariants: } \forall\, u \in \{0, 2, \ldots, K\},\ \sum_{j=1}^{m} A[j].b_{uj}.\mathbf{x}_u \leq \sum_{j=1}^{m} b_{uj}.a[j] \tag{5.22}$$

$$\text{Evolve: } \forall\, u \in \{0, 2 \ldots, K\},\ \mathbf{x}_{u+1} = \mathbf{x}_u + \sum_{j=1}^{m} c[j].b_{uj}.t_u \tag{5.23}$$

Figure 5-8: Objective function and constraints for $\mathsf{MOPT}(K, \tau_a)$

ensure that $(\mathbf{x}_u, \mathsf{switch}(j,k), \mathbf{x}_{u+1})$ is a valid mode switching transition. These constraints

simplify to the inequalities $G[j,k]\mathbf{x}_u \leq g[j,k]$ and $R[j,k]\mathbf{x}_{u+1} \leq r[j,k]$ which correspond to the precondition and the initialization conditions on the pre and the post-state of the transition. For each even u, $\mathbf{x}_u$ evolves to $\mathbf{x}_{u+1}$ through a trajectory in some mode, say j. Constraint (5.22) ensures that $\mathbf{x}_u$ satisfies the invariant of mode j described by the inequality $A[j]\mathbf{x}_u \leq a[j]$. An identical constraint for $\mathbf{x}_{u+1}$ is written by replacing $\mathbf{x}_u$ with $\mathbf{x}_{u+1}$ in (5.22). Since the differential equations have constant right hand sides and the invariants describe polyhedra in $\mathbb{R}^n$, the above conditions ensure that all the intermediate states in the trajectory satisfy the mode invariant. Equation (5.23) ensures that, for each even u, $\mathbf{x}_u$ evolves to $\mathbf{x}_{u+1}$ in t_u time according to the differential equation $d(\mathbf{x}) = c[j]$.

Some of these constrains involve nonlinear terms. Using the "big M" method [Wil90] we can linearize these equation and inequalities. For example, $b_{uj}\mathbf{x}_u$ in (5.22) is the product of real variable $\mathbf{x}_u$ and boolean variable b_{uj}. We linearize it by replacing $b_{uj}\mathbf{x}_u$ with $\mathbf{y}_u$, and adding the following linear inequalities: $\mathbf{y}_u \geq b_{uj}\delta$, $\mathbf{y}_u \leq b_{uj}\Delta$, $\mathbf{y}_u \leq \mathbf{x}_u - (1-b_{uj})\delta$, and $\mathbf{y}_u \geq \mathbf{x}_u - (1-b_{uj})\Delta$, where δ and Δ are the lower and upper bounds on the values of $\mathbf{x}_u$.

5.6.5 Case Study: Thermostat

We use the MILP technique together with switching simulation relations to verify the ADT of a thermostat with nondeterministic switches. The Thermostat2 automaton (Figure 5-9 *Left*) has two modes *on* and *off*, two continuous variables x and z, and real parameters $h, K, \theta_1, \theta_2, \theta_3, \theta_4$, where $0 < \theta_1 < \theta_2 < \theta_3 < \theta_4 < h$. In *off* mode the heater is off and the temperature x decreases according to the differential equation $d(x) = -Kx$. This is described by the state model heaterOff. While the temperature x is between θ_2 and θ_1, the switchOn action *may* occur, and it *must* occur when x drops to θ_1. As an effect of switchOn, the mode changes to *on*, where the heater is on and the behavior is described by state model heaterOn. Here, x rises according to the differential equation $d(x) = K(h-x)$. While x is between θ_3 and θ_4, the switchOff action may occur, and it must occur when x increases to θ_4. The continuous variable z measures the total time spent in mode *on*.

SHIOA Thermostat2 is not initialized because the continuous variable x is not reset with every mode switching transition; nor is it a rectangular SHIOA because the differential equations do not have constant right-hand sides. There exists, however, a rectangular initialized SHIOA, we call it ThermAbs, such that Thermostat2 $\leq_{switch}$ ThermAbs. Automaton ThermAbs (Figure 5-9 *Right*) has real-valued parameters L_0 and L_1, a clock variable t, and two modes *on* and *off*. In each mode, t increases at a unit rate. When t reaches L_i in mode l_i, a switch to the other mode *may* occur and if it does then t is set to zero. We define a relation $\mathcal{R}$ on the state spaces of Thermostat2 and ThermAbs such that with appropriately chosen values of L_0 and L_1, ThermAbs captures the fastest switching behavior of Thermostat2.

Definition 5.12. For any $\mathbf{x} \in Q_{\mathsf{Thermostat}}$ and $\mathbf{y} \in Q_{\mathsf{ThermAbs}}$, $\mathbf{x}\ \mathcal{R}\ \mathbf{y}$ if and only if:

(1) $\mathbf{x} \lceil mode = \mathbf{y} \lceil mode$, and

(2) if $\mathbf{x} \lceil mode = l_0$ then $\mathbf{y} \lceil t \geq \frac{1}{k} \ln \frac{\theta_3}{\mathbf{x}\lceil x}$ else $\mathbf{y} \lceil t \geq \frac{1}{k} \ln \left(\frac{h-\theta_2}{h-\mathbf{x}\lceil x}\right)$.

Lemma 5.21. *If we set* $L_0 = \frac{1}{k}\ln\frac{\theta_3}{\theta_2}$ *and* $L_1 = \frac{1}{k}\ln\frac{h-\theta_2}{h-\theta_3}$*, then the relation* $\mathcal{R}$ *of Definition 5.12 is a switching simulation relation from* Thermostat2 *to* ThermAbs.

```
automaton Thermostat2(θ1, θ2, θ3, θ4, K, h : Real)
    where 0 < θ1 < θ2 < θ3 < θ4 < h
  type Status enumeration [on, off]

  signature
    internal switchOn, switchOff

  variables
    internal loc : Status := off;
      x : Real := θ4; z : Real := 0;

  transitions
    internal heaterOn
      pre loc = off ∧ x ≤ θ2;
      eff loc := on;

    internal switchOff
      pre loc = on ∧ x ≥ θ3;
      eff loc := off;

  trajectories
    trajdef heaterOn
      invariant x ≤ θ4 ∧ loc = on;
      stop when x = θ4;
      evolve d(x) = K(h − x); d(z) = 1;

    trajdef heaterOff
      invariant x ≥ θ1 ∧ loc = off;
      stop when x = θ1;
      evolve d(x) = −Kx; d(z) = 0;
```

```
automaton ThermAbs(L0, L1 : Real)
  type Status enumeration [on, off]

  signature
    internal switchOn, switchOff

  variables
    internal mode : Status := off; r : Real := L1;

  transitions
    internal switchtOn
      pre mode = off ∧ r ≥ L0;
      eff mode := on; r := 0;

    internal switchOff
      pre mode = on ∧ r ≥ L1;
      eff mode := off; r := 0;

  trajectories
    trajdef always
    evolve d(r) = 1;
```

Figure 5-9: Thermostat2 SHIOA and its rectangular initialized abstraction ThermAbs.

Proof. The proof is by induction on the length of an execution of Thermostat2 and is structurally similar to the proof of Lemma 5.16. The base case follows immediately as the start state of Thermostat2 is related to the start state of ThermAbs. Next, for the inductive step we show by cases that given any state $\mathbf{x} \in Q_{\mathsf{Thermostat}}, \mathbf{y} \in Q_{\mathsf{ThermAbs}}$, $\mathbf{x} \mathcal{R} \mathbf{y}$, and an execution fragment α of Thermostat2 starting from $\mathbf{x}$ and consisting of either a single action or a single trajectory, there exists a corresponding execution fragment β of ThermAbs, starting from $\mathbf{y}$ that satisfies the conditions required for $\mathcal{R}$ to be a switching simulation relation.

Case 1: α is a $(\mathbf{x}, \mathsf{on}, \mathbf{x}')$ transition of Thermostat2.
We choose β to be the $(\mathbf{y}, \mathsf{on}, \mathbf{y}')$ transition of ThermAbs. First we show that the action on is enabled at $\mathbf{y}'$. Since $\mathbf{x} \mathcal{R} \mathbf{y}$, we know that $\mathbf{y} \lceil \mathit{mode} = \mathbf{x} \lceil \mathit{mode} = l_0$ and further, $\mathbf{y} \lceil r \geq \frac{1}{k} \ln \frac{\theta_3}{\mathbf{x} \lceil x}$. But we know that, $\mathbf{x} \lceil x \leq \theta_2$ because the on action of Thermostat2 is enabled at $\mathbf{x}$. It follows that $\mathbf{y} \lceil r \geq \frac{1}{k} \ln \frac{\theta_3}{\theta_2} = L_0$, that is, the on action is enabled at $\mathbf{y}$. Next we show that $vx' \mathcal{R} \mathbf{y}'$. Its immediate that $\mathbf{x}' \lceil \mathit{mode} = l_1 = \mathbf{y}' \lceil \mathit{mode}$. From the transition definition of both the on actions, it follows that $\mathbf{y}' \lceil r = 0$ and $\mathbf{x}' \lceil x = \mathbf{x} \lceil x \leq \theta_2$ and $\ln \frac{h-\theta_2}{h-\mathbf{x}' \lceil x} \leq 0$. Therefore, $\mathbf{y}' \lceil r \leq \frac{1}{K} \ln \frac{h-\theta_2}{h-\mathbf{x}' \lceil x}$.

Case 2: α is a $(\mathbf{x}, \mathsf{off}, \mathbf{x}')$ transition of Thermostat2.
We choose β to be the $(\mathbf{y}, \mathsf{on}, \mathbf{y}')$ transition of ThermAbs and the rest of the proof is similar to that of case 1.

Case 3: α is a single trajectory τ of Thermostat2 such that $(\tau \downarrow \mathit{mode})(0) = l_0$. We choose

β to be a trajectory τ' of ThermAbs defined as follows:

$$\forall\ t \in \tau'.dom,\ (\tau' \downarrow mode)(t) = l_0 \text{ and } (\tau' \downarrow r)(t) = (\tau' \downarrow r)(0) + t$$

It follows immediately that τ' is a valid trajectory for ThermAbs because it satisfies the the differential equation $d(r)) = 1$ and the *mode* remains constant.

Case 4: α is a single trajectory τ of Thermostat2 such that $(\tau \downarrow mode)(0) = l_1$. The proof for this case is the same as that of the previous case.

■

Lemma 5.21 implies that Thermostat2 $\leq_{switch}$ ThermAbs, that is, for any $\tau_a > 0$ if τ_a is an ADT for ThermAbs then τ_a is also an ADT for Thermostat2. Since ThermAbs is rectangular and initialized, we can use the MILP technique to check any ADT property of ThermAbs.

We formulated the $\mathsf{MOPT}(K, \tau_a)$ for automaton ThermAbs and used the GNU Linear Programming Kit [GNU] to solve it. Solving for $K = 4, L_0 = 40, L_1 = 15$, and $\tau_a = 25, 27, 28$, we get optimal costs -0.4, $-4.358E^{-13}(\approx 0)$ and 0.071, respectively. We conclude that the ADT of ThermAbs is $\geq 25, \geq 27$, and < 28. Since ThermAbs $\geq_{switch}$ Thermostat2, we conclude that the ADT of the thermostat is no less than 27.

For finding counterexample execution fragments for the proposed ADT properties, the MILP approach can be applied to non-initialized rectangular SHIOA as well. In such applications, the necessity part of Theorem 5.17 does not hold and therefore from the failure to find a counterexample we cannot conclude that the automaton satisfies the ADT property in question.

5.7 Summary

Using results from the literature on switched systems, we find that stability verification of SHIOAs can be decomposed into two independent tasks, namely, (a) finding the Lyapunov functions for the individual state models, and (b) verifying the appropriate average dwell time property. Task (a) can be accomplished using existing techniques in systems theory. In this chapter we have presented two techniques for accomplishing task (b).

The first method transforms the given SHIOA by adding history variables, such that the transformed automaton satisfies an invariant property if and only if the original automaton has the ADT property. In order to prove the resulting invariant properties, we appeal to the large body of tools available for proving invariants for hybrid systems. This method for verifying ADT properties is automatic only for classes of hybrid systems for which reachability is decidable. For hybrid systems that are outside this class, the method is applicable, but the resulting invariants will have to be proved using the methods described in Chapter 4 of this thesis.

The second method relies on solving an optimization problem over the set of cyclic executions of the given SHIOA. For the class of one-clock initialized SHIOAs this involves finding the maximum mean-cost cycle of a graph (of size comparable to the given SHIOA), and can be solved very efficiently by classical graph algorithms. For rectangular initialized SHIOAs, the optimization problem can be formulated as a MILP. For initialized SHIOAs with more general dynamics, the same technique applies but harder optimization problems will have to be solved.

For non-initialized hybrid automata, the solution of the optimization problem can give executions that serve as counterexamples to the ADT property in question, but the failure in finding such executions *does not* indicate that the ADT property in question is satisfied. That is, for non-initialized SHIOAs, the method is incomplete. We have defined equivalence of SHIOAs with respect to switching speed and proposed switching simulations for establishing ADT-equivalence of two SHIOAs. This can serve as a method for abstracting non-initialized SHIOAs with initialized ones.

The two methods for verifying ADT can be combined as follows: we can start with some candidate value of $\tau_a > 0$ and search for a counterexample execution fragment for it using the optimization-based approach. If such an execution fragment is found, then we decrease τ_a (say, by a factor of 2) and try again. If eventually the optimization approach fails to find a counterexample execution fragment for a particular value of τ_a, then we use the invariant approach to try to prove that this value of τ_a is an ADT for the given system.

There are several research directions to be pursued related to the general area of stability verification of SHIOAs. One interesting problem is to develop stability verification techniques for the general class of SHIOAs that have both stable and unstable state models. Sufficient conditions for stability of such systems already exist in the control theory literature (see, for example [ZHYM00]). These conditions, however, take the form of switching time related properties and are hard to verify, just like the ADT property, and hence they call for the development of new verification techniques. Another direction, is to relax assumption (1) and explore verification of input-output and input-to-state stability properties of SHIOAs with input/output variables, using the results from [VCL06]. Yet another direction of future research is to extend these techniques to stochastic hybrid systems, by combining the probabilistic timed I/O automata of Chapter 7 with ADT-like stability results for stochastic switched systems from [CL06].

Chapter 6

Mechanizing Proofs

In this chapter we describe techniques for mechanizing invariance and implementation proofs for SHIOAs. The examples of Chapters 4 and 5 illustrate that the SHIOA framework provides a systematic way of constructing such proofs, by way of induction over the length of the executions followed by case analysis of the actions and the state models. Often, however, such proofs are lengthy, tedious to construct, and they always require careful attention to specification details. This suggests that software tools for construction and management of proofs are needed.

Mechanical theorem provers, such as, PVS [ORR+96], Isabelle/HOL [Pau93, GCMF93], COQ [The04], provide their own specification languages, typically some variant of high-order logic, and very general semi-decision procedures for interactively checking veracity of formulae written in that language. In order to use a theorem prover for construction and management of SHIOA proofs, we have to answer two questions: (a) translate `HIOA` specifications to the language of the theorem prover, and (b) minimize interaction of the user with the theorem prover. In Section 6.2, we address (a) by describing a scheme for translating a large class of `HIOA` specifications to the language of the PVS theorem prover. This work builds on and generalizes previous work on translating of Timed I/O Automaton specifications to PVS [LKLM05]. In Section 6.3, we address (b) by describing a set of specialized *proof strategies*—programs for constructing proofs—that exploit the knowledge of our translation scheme and partially automate proofs. This section is based on a sequence of papers on strategy development [MA03, MA04, MA05, ALL+06]. We conclude the chapter by discussing our experiences in developing and using these software tools.

6.1 An Overview

PVS [ORR+96] is an environment for formal specification and verification developed at SRI. It consists of a specification language (henceforth `PVS`), a number of predefined theories, a type checker, an interactive theorem prover that supports the use of several decision procedures and a symbolic model checker. By exploiting the synergy between a highly expressive specification language and powerful automated deduction routines, PVS serves as an environment for constructing and maintaining large formalizations and proofs. Our preference of PVS over other existing theorem provers is based on several of its features:

(a) The `PVS` language [OSRSC99] is expressive, natural, and syntactically close to classical, typed high-order logic. It has many attractive features including parameterized specifications and predicate subtyping.

(b) Theory interpretations [OS01], a new feature in PVS, is used to instantiate the declared types in a specifications with definitions. Theory interpretations can be used for exhibiting models for an axiomatic theory and to refine a abstract specifications in terms of a concrete one. This feature is useful for defining simulation relations between a pair of SHIOAs (see Section 6.2.9).

(c) The PVS theorem prover [SORSC99] provides a set of powerful decision procedures. New state-of-the-art techniques are continually added to make the decision procedures efficient. And with the excellent tutorials available [COR+95], this prover sports a relatively flatter learning curve.

(d) PVS provides a way of developing *strategies* [AVM03] for partially automating proofs. A *proof strategy* or a *strategy* is a Lisp program that accesses the state of an ongoing proof, constructs a sequence of proof steps on-the-fly, and applies it to the proof. Our techniques for partially automating proof construction are embodied by a set of strategies.

(e) PVS has been widely used in many verification projects throughout the world and has a lively user community [ORSSC98, Int06]. The current version of PVS is open source (under the GPL license).

An alternative to translating `HIOA` specifications to `PVS` is to write this SHIOA specification directly in `PVS` . The resulting `PVS` specification will be a set of definitions for the variables, actions, transitions, and trajectories, and a set of axioms defining the semantics of these objects. We have decided not to opt for this direct route for the following reasons: (a) `HIOA` has the variables, transitions, and state model structure, which is a natural representation of SHIOAs, (b) `HIOA` allows us to describe the transitions using operational semantics, whereas in `PVS` , transition definitions have to be functions or relations, (c) `HIOA` provides a natural way for describing trajectories using differential equations, and (d) having the theorem-prover-independent `HIOA` language allows one to combine use of different tools designed for the SHIOA framework. Hence, SHIOA specifications are written in the `HIOA` language, these are translated to `PVS`, and then the PVS theorem prover is invoked on the translated `PVS` specification.

There has been several prior proposals for translating certain subsets of the IOA language [] to theorem provers, such as, Larch [BGL02, GG91], PVS [Dev99], and Isabelle [NW03, Pau93].

Our design of the `HIOA` to `PVS` translator builds upon [BGL02] and [Mit01], but the first to translate the continuous aspects of the behavior of hybrid systems to the language of a theorem prover. The key insight gained from the previous works is to convert nondeterministic transition (and consequently trajectory) definitions into deterministic functions (of the pre-state) by adding extra parameters. This enables the theorem prover to compute the post-state of a transition (or a trajectory) and simplifies theorem proving. This procedure of "determinizing" is further discussed in Section 6.2.6.

In the PVS parlance, definition of types, operators, lemmas, etc., constitute a *theory*. The Timed Automata Modeling Environment (TAME) of [Arc01] provides a `PVS` *theory template* for describing a special kind of automata called *MMT Automata* (see Section 1.4 or [MMT91] for details). One manually instantiates this theory template with the states, actions, and transitions of an MMT automaton, say $\mathcal{A}$, to obtain the `PVS` theory that specifies $\mathcal{A}$. Instantiating a theory template is also the mechanism underlying our translation

scheme, although our theory template is closer to the one in [LKLM05]. The key difference between the TAME-template and the template [LKLM05] has to do with how time is modeled in the underlying mathematical models. In a timed I/O automaton, the model of [LKLM05], progress of time and continuous evolution of variables are modeled by trajectories (as in SHIOAs). MMT-automata, on the other hand, do not have general continuous variables; there are upper and lower time bounds associated with actions, and actions must occur within those time-bounds. Hence, in the template of [Arc01], advancement of t units of time is modeled by the occurrence of an action $\nu(t)$. The time-bounds for the actions are enforced by deadline variables that enable the $\nu(t)$ action only when t amount of time can elapse without violating those bounds. This approach does not allow us to specify interesting trajectories by asserting of properties that must hold *throughout the duration of a trajectory*. In [LKLM05] we proposed a template that allows translation of such trajectories by embedding the trajectory τ as a functional parameter of $\nu(t, \tau)$. The enabling condition of $\nu(t, \tau)$ asserts the conditions that must hold at each point in the domain of τ. For example, we can say that a particular $\nu(\tau, t)$ action is enabled only when $\tau.ltime < ubound$ and $\tau(t) \lceil x = \tau(0) \lceil x + 4t$. Here, *ubound* is a discrete deadline variable and x is a continuous variable increasing at four times the rate of real time.

The above translation scheme has been used to design the software tool for translating TIOA specifications to PVS . This tool has been implemented by Lim [Lim01]. Recall, that the TIOA language is a restriction of HIOA which does not allow input/output variables. In Section 6.2, we present the design of a scheme that generalizes the theory template of [LKLM05] and allows translation of a general class of HIOA specification to PVS . Although the basic mechanism for translation remains the same, namely, instantiation of template theories, several changes have been made in the templates to accommodate external variables.

The possibility of partially automating proofs by using an interactive theorem prover was demonstrated earlier by the TAME system [Arc01] in the context of the MMT automaton model. TAME strategies primarily support proofs of invariant properties. These strategies are compatible with the translation scheme described here and we describe them briefly in Section 6.3.1. Apart from partial automation, the TAME strategies illustrate that theorem prover support is useful for (a) managing large proofs, (b) re-checking proofs after minor changes in the specification, and (c) generating human readable proofs from proof scripts. These strategies demonstrate how careful design of theory templates aid the development of proof strategies. Much of our later work [MA03, MA04, MA05, ALL+06] on developing proof strategies for implementation relations are informed by the lessons learned from the TAME experience. The technical challenges we faced during the development of these new strategies are discussed in Section 6.2.9.

We have applied the translator and the strategies in several case studies: a tree-based leader election protocol [MA04, MA05], Fischer's mutual exclusion algorithm [ALL+06], a two-task race system, and a failure detector [LKLM05]. In Chapters 4 and 5 we have shown how simulation relations are used for proving implementation and ADT-equivalence of SHIOAs. A specific type of implementation that occurs quite often in real-time systems involves showing that the actions of a given SHIOA $\mathcal{A}$ occur within certain time bounds. The bounds themselves may be complex functions of the parameters of $\mathcal{A}$ or dependent on external inputs. Such timing properties can be cast as implementation relations: first an abstract SHIOA $\mathcal{B}$ is specified that has exactly the required timing behavior—this is easy because *we know* the timing requirements that we wish to prove. Then, we show that that $\mathcal{A}$ implements $\mathcal{B}$. In two of our case studies (Sections 6.4.1 and 6.4.2), we follow this

recipe and prove timing properties using simulation relations. In constructing the simulation proofs the strategies we have developed prove to be effective. In addition, the strategies for manipulation of real-inequalities provided by the Field [MM01] and the Manip [Vit02] packages are useful because these simulation relations involve inequalities between variables of the system and its abstraction. Overall, our experience with the tool suggests that it is possible (but not routine) to express realistic systems in HIOA , translate the HIOA specification to PVS , and then prove properties of the system using PVS on the translator output.

6.2 Translation

In this section we describe the design of a scheme for translating HIOA specifications to PVS. A HIOA specification file called say, `xyz.hioa`, when translated, produces a set of PVS files containing several PVS theories. A summary of these files and their contents are shown in Table 6.1. Essentially these generated theories contain definitions specifying the variables, actions, transitions, trajectories, and putative invariant predicates of the input HIOA specification. Some theories such as *xyz_invariant* and *xyz_2_XYZ* also contain lemmas and theorems asserting properties such as invariance of predicates and simulation relations. For generating these theories, the translator instantiates a PVS theory-template with the specifics obtained from `xyz.hioa`. These generated theories by themselves are not complete; they import a set of generic SHIOA library theories, that we have developed, for defining the semantics of SHIOAs. One library theory, for example, defines what it means for a state of any SHIOA to be reachable in terms of abstract transitions and trajectories. The combination of the translated theories and the generic library theories completely describe the behavior of the particular SHIOA in question, and the PVS prover can be invoked to prove theorems about its behavior. In what follows, we take a closer look at the translation

Filename	Theory name	Description
`xyz_decls.pvs`	*xyz*	Definition of the components of automaton xyz
`common_decls.pvs`	*common_decls*	Automaton parameters and where clause
`xyz_invariants.pvs`	*xyz_invariants*	Lemmas asserting invariants of xyz
`xyz_2_XYZ.pvs`	*xyz_2_XYZ*	Lemma asserting simulation from xyz to XYZ

Table 6.1: PVS files generated by translator.

of HIOA specifications to PVS theories and the definitions provided by the generic SHIOA library theories.

6.2.1 Assumptions

The correct translation to PVS assumes the following two restrictions on the SHIOA $\mathcal{A}$ that is specified in HIOA :

(1) $\mathcal{A}$ does not have input variables. In specifying $\mathcal{A}$ in PVS , we have to define its trajectories. Trajectories of input variables are restricted only by their dynamic types. For specifying dynamic types of continuous input variables we have to define the class of

functions that is piece-wise continuous. This poses certain difficulties because the theory of real analysis including continuity is not completely developed in PVS. Of course, when such a theory becomes available we can relax this assumption.

(2) HIOA allows us to define transitions of $\mathcal{A}$ by writing arbitrary relations relating the pre- and post-states. This is not allowed. The restriction allows us to convert transition relations to transition functions; thus we can compute the post state of a transition (or a trajectory) from its pre-state. Note that transitions (and trajectories) of $\mathcal{A}$ can still have nondeterminism through **choose** statements (and inequalities).

It is worth noting that HIOA specifications allow us to write arbitrarily complicated differential equations in the state models. But, we cannot assert conditions involving derivatives and integrals of functions directly in PVS (see Section 6.2.7 for details). Hence, our translation of state models relies on actually solving these differential equations, and therefore, we cannot automatically translate arbitrarily complex differential equations. In fact, the current implementation of the TIOA to PVS translator can only translate specifications with state models that have differential equations constant right hand sides. This is not a restriction imposed by the translation scheme, but it reflects the limited ability of our current translator software in solving differential equations. For classes of differential equations outside the above, solutions may be provided manually by the user. In the future, this step could also be automated by invoking external DAE solving tools such as Maple [GGC81] and Macsyma [Mat74].

6.2.2 Types and Vocabularies

Simple static types of HIOA , such as, `Bool`, `Char`, `Int`, `Nat`, `Real`, and `String` have their equivalents in PVS. The HIOA types defined using the type constructors `Enumeration`, `Tuple`, `Union`, `Set`, `Map`, and `Array` are directly translated to type declarations in PVS . The type `AugmentedReal` is translated to a type called *time* which is defined as a DATATYPE that is the union of two subtypes: non-negative reals (`nnreal` in PVS) and a singleton type with an element called *infinity*.

HIOA vocabularies are directly translated to PVS theories. The semantics of these type and operator declarations are written in PVS library theories, which are imported by the translator output.

Example 6.1. The directedGraph vocabulary of Figure 3-2 is reproduced in Figure 6-1 alongside its PVS translation. The *directedGraph* theory is parameterized by the type parameter T. The *sequences* theory is a PVS library theory for finite sequences. The result of importing the *sequences* with parameter T into *directedGraph* is that the type `sequence` becomes defined as a finite sequence of elements of type T, and the usual operators on objects of type `sequence` become available in *directedGraph*. Tuples with accessors for the individual components are called *records* in PVS and they are declared with $[\# \ldots \#]$ syntax. By importing *directedGraph*$[I]$ into another PVS theory, directed graphs with vertices of type I can be used. Notice that PVS allows us to declare *connected* as an abstract (uninterpreted) function. This is is sufficient to use it in other specifications, however, if we are interested in proving theorems about a specification that relies on the behavior of *connected*, then we have two options: (1) We write axioms describing the of *connected*, for example, $connected(a, b) \wedge connected(b, c) \Rightarrow connected(a, c)$. (2) We provide a concrete definition for computing *connected*; in this case, this would be an inductive definition.

vocabulary directedGraphs(T :**type**) **types** $Edge$ = `Tuple` [$src : T, dst : T$]; $Digraph$ = `Tuple` [$vset$:`Set`[T], $eset$:`Set`[$Edge$]]; $Path$ = `Seq` [T]; **operators** $connected : T, T \to$ `Bool`; $addEdge : Digraph, Edge \to Digraph$; **end**	*directedGraphs*[T: TYPE]:THEORY BEGIN IMPORTING *sequences*[T] *Edge*: TYPE = [# *src*: T, *dst*: T #] *Digraph*: TYPE = [# *vset*: setof[T], *eset*:setof[*Edge*] #] *Path*: TYPE = sequence *connected*:[$T, T \to$ bool] *addEdge*: [*Digraph*, *Edge* $\to$ *Digraph*] END *directedGraphs*

Figure 6-1: directedGraphs vocabulary in HIOA translated to *directedGraphs* theory in PVS.

Automaton parameters in HIOA are declared as constants in a separate PVS theory called *common_decls*. The where clause relating the automaton parameters is stated as an axiom relating the constants.

6.2.3 Variables and Initial States

The set of valuations of output and internal variables of an SHIOA $\mathcal{A}$ are translated to two separate records types in PVS . A collection of output variables $y_1, y_2, \ldots, y_n$, of respective types $Y_1, Y_2, \ldots, Y_n$, is translated to a record type declaration called *OutputVals*:

OutputVals: TYPE = [# $y_1 : Y_1$, $y_2 : Y_2$, $\cdots$, $y_n : Y_n$ #]

The type *OutputVals* corresponds to the set of all valuations of the output variables of $\mathcal{A}$. The internal variable declarations are translated to similar record types called *States*. The *ExternalVals*, *LocalVals*, and *Vals* types, corresponding to the valuations of all the external variables, all the local variables, and all the variables of $\mathcal{A}$, are defined as:

ExternalVals: TYPE = [# *output*: *OutputVals* #]
LocalVals: TYPE = [# *states*: *States*, *output*: *OutputVals* #]
Vals: TYPE = [# *states*: *States*, *output*: *OutputVals* #].

There are two overlapping mechanisms for specifying the starting states of a SHIOA in the HIOA language: (a) individual variables are assigned particular values and (b) an predicate on the variables defines the set of possible starting valuations. Both these features of HIOA are translated to a single predicate called *Start* in PVS.

Example 6.2. The HIOA code fragment from Section 3.2.4 and its PVS translation is shown in Figure 6-2. In translating variable types to PVS, the `DiscreteReal` variables are treated in the same way as `Real` variables; additional constrains on the former variables appear when their trajectories are defined (see Section 6.2.7). Since initial values can be defined only for internal variables in HIOA , *Start* is predicate on *States* and not *Vals*.

6.2.4 Trajectory Types

The PVS translation defines the following trajectory types derived from the variables types. These types are used in the subsequent definition of actions, transitions, and trajectories.

interval(t:`nnreal`):TYPE = {*t1*:`nnreal` | *t1* $\leq$ *t*}
Trajs(t:`nnreal`):TYPE = [*interval*(t) $\to$ *Vals*]
ExtTrajs(t:`nnreal`):TYPE = [*interval*(t) $\to$ *ExternalVars*]

HIOA	PVS
variables **output** u : `Discrete Real` **internal** x : `Real` := 5, y : `Real` := **choose** k **where** $-5 \leq k \leq 5$, z : `Real` **initially** $z \times z + y \times y \leq 10$	*OutputVals*: TYPE = [# *u*:real #] *States*: TYPE = [# *x*:real, *y*:real, *z*:real #] *ExternalVals*: TYPE = [# *output*: *OutputVals* #] *LocalVals*: TYPE = [# *states*:*States*, *output*:*OutputVals* #] *Vals*: TYPE = [# *states*:*States*, *output*:*OutputVals* #] *Start*(*s*:*States*): bool = *x*(*s*) = 5 AND *y*(*s*) ≥ -5 AND *y*(*s*) ≤ 5 AND exp(*z*(*s*),2) + exp(*y*(*s*),2) ≤ 10

Figure 6-2: Variable declarations in HIOA translated to type declarations in PVS .

StateTrajs(*t*:nnreal):TYPE = [*interval*(*t*) → *States*]

The type $interval(t)$, for a non-negative real t, equals the real interval $[0, t]$. The type $Trajs(t)$ equals the set of functions that map the interval $[0, t]$ to *Vals*, that is, valuations of all the variables of $\mathcal{A}$. Here we remark that the PVS type *Trajs* does not necessarily correspond to the mathematical object $trajs(V)$, where V is the set of variables of the automaton being specified. The set $trajs(V)$ is the set of trajectories that respect the dynamics types of the individual variables in V. These functions in $Trajs$ do not necessarily belong to the dynamic types; dynamic type constraints cannot be stated in PVS at present.

The type $ExtTrajs(t)$ equals the set of functions that map the interval $[0, t]$ to *ExternalVals*, that is, valuations of the external variables of $\mathcal{A}$. The parameterized types $StateTrajs(t)$ is defined analogously.

6.2.5 Actions, State Models, and Moves

The set of action names and state model names of an SHIOA are translated to a single PVS datatype called *Moves*. Although actions and state models describe very different kinds of objects, they both define how the variables of an SHIOA changes, and at a syntactic level they play similar roles in defining the reachable states and traces of an SHIOA. Thus, putting them together in a common type simplifies the PVS theory. Informally, the *Moves* datatype defines the names of all the possible ways in which the variables of the SHIOA $\mathcal{A}$ can change. For each action of $\mathcal{A}$, *Moves* datatype has a separate constructor; and all the state models of $\mathcal{A}$, are encapsulated by a single constructor called *smodels*. The parameters of *smodels* constructor are explained in the following example. If an actions and state models have formal parameters, the corresponding constructors have additional parameters.

Four predicates are defined on *Moves* to distinguish the actions from the state models, and to distinguish the different kinds of actions from one another. For example, (*Actions*) is a subtype of *Moves* also (*Trajectories*) is a subtype of *Moves*. The following example illustrates these definitions.

Example 6.3. The PVS code fragment in Figure 6-3 shows the definition of the *Moves* datatype for the Supervisor automaton from Chapter 4 (Figure 4-7). The enumeration type *SModelNames* is derived from the names of the state models in the HIOA code, that is, the identifiers following the **trajdef** keyword. In PVS , a datatype is declared by giving a list of constructors-recognizers pairs. A datatype constructor is a function that returns an object of the type that is being defined. And a recognizer returns true when it is applied to

an object that has been constructed using its constructor. For example, the second entry in the list for the *Moves* datatype is the constructor *sample* with parameters $x0, x1$, and its recognizer *sample*?. Hence, $sample(\frac{\pi}{4}, 5)$ has type *Moves*. And for any object a of type *Moves*, $sample?(a)$ returns true if and only if a has been constructed using the *sample* constructor.

```
automaton Supervisor(u_min, u_max : Real)
  type
    Mode = Enumeration [sup, usr]

  signature
    input sample(x_0, x_1 : Real)
    input usrOutput(u' : Real)
    output supOutput(u' : Real)

  trajectories
    trajdef supMode
      ...

    trajdef usrMode
      ...
```

```
SModelNames:TYPE = {supMode, usrMode}

Moves: Datatype BEGIN
    smodel(name:Modes,
      ltime:nnreal, tau:Trajs(ltime)):smodel?
    sample(x0, x1:real):sample?
    usrOutput(u:real):usrOutput?
    supOutput(u:real):supOutput?
END

Trajectories(a:Moves):bool = smodel?(a)
Actions(a:Moves):bool = NOT smodel?(a)
InternalActions(a:(Actions)):bool = true
ExternalActions(a:(Actions)):bool = false
```

Figure 6-3: Actions and State models in HIOA translated to Moves in PVS .

Each object defined by the *smodel* constructor corresponds to a trajectory of the variables of $\mathcal{A}$. The *smodel* constructor has three parameters: (1) *name* is the name of the state model that defines the trajectory, (2) *ltime* is the limit time of the trajectory, and (3) *tau* is the trajectory itself.

The *Trajectories* predicate distinguish the *Moves* that correspond to *smodel* and from the others, namely those corresponding to actions. In PVS, a predicate p on a type T readily defines a predicate subtype $(p) \triangleq \{x : T \mid p(a)\}$. In the above example, the (*Actions*) subtype is used to define the predicates the *InternalActions* and *ExternalActions*.

6.2.6 Transitions

Discrete transitions in HIOA are specified using preconditions and effects, and the state models are specified using invariants, stopping conditions, and evolve-clauses. The PVS translation of a HIOA specification converts these definitions into two independent definitions, namely, $Enabled(m : Moves, v0 : Vals)$: bool, and $Trans(m : Moves, v0 : States) : Vals$. The predicate $Enabled(m, v0)$ returns true if and only if the move m is *allowed* at from valuation $v0$. The function $Trans(m, v0)$ returns the valuations of the variables that result from the occurrence of m at $v0$.

A couple of points are to be noted before we proceed to describe further details of the translation. First, in an execution of $\mathcal{A}$, $Trans(m, v0)$ is applicable for a given valuation $v0$ and a given move m, only if $Enabled(m, v0)$ is in fact true. This information, however, is not available in the definition of $Trans(m, v0)$ and hence in some cases the PVS type checker produces additional proof obligations called Type Correctness Conditions (TCCs). For example, consider a receive action that is *Enabled* when a *queue* is non-empty, and when the action occurs it removes the first element of the *queue*. In type-checking the *Trans* function for receive, PVS will generate a TCC asserting the non-emptiness of *queue* (because the first element can be removed only for non-empty queues). This TCC can only be proved if we add the precondition of receive as a conditional in the eff program. Secondly, in HIOA

the valuations of the variables that result from the occurrence of a move (transition or trajectory) may be chosen non-deterministically, however, $Trans(m, v0)$ is a function which uniquely determines the valuations. The missing step here is to introduce additional formal parameter(s) for such nondeterministic moves, whereby the choice of the values for the parameter(s) uniquely determine the valuation of the variables. This technique of pushing internal nondeterminism (choices within a transition definition) to external nondeterminism (choice over several transitions) has been employed previously in [BGL02] and [Mit01] for the purpose of transforming transition relations to transition functions. In this section we describe the translation of the discrete transitions and in Section 6.2.7 we describe the translation of the state models.

The *Enabled* predicate for actions is defined using a CASE statement which contains the preconditions from the HIOA specification. The HIOA program specifying the effects of transitions consists of three types of sequential statements, namely, assignments, conditionals, and loops. Each of these statements is translated to its corresponding functional relation between states, as shown in Table 6.2.6. The term P is a HIOA program, while $trans_P(v0)$ is a PVS function that returns the valuation obtained by performing program P on $v0$. Sequential statements like $P_1; P_2$ are translated to a composition of the corresponding func-

Statement type	HIOA program P	PVS translation $trans_P(v)$
Assignment	$x := exp$	v WITH $[x := exp]$
Conditional	**if** $pred$ **then** P_1 **fi**	IF $pred$ THEN $trans_{P_1}(v)$ ELSE v ENDIF
Conditional	**if** $pred$ **then** P_1 **else** P_2 **fi**	IF $pred$ THEN $trans_{P_1}(v)$ ELSE $trans_{P_2}(v)$ ENDIF
Loop	**for** x **in** A **do** P_1 **od**	$forloop(A, v)$: RECURSIVE $Vals$ = IF `empty?`(A) THEN v ELSE v = `choose`(A), $s' = forloop($`remove`$(x, A), v)$ IN $trans_{P_1}(v')$ ENDIF MEASURE `card`(A)

Table 6.2: Translation of program statements. v is a valuation of variables; exp is an expression of type $type(v)$; $pred$ is a predicate; A is a finite set; $choose$ picks an element from A. WITH makes a copy of the record s, assigning the field v with a new value t.

tions $trans_{P_2}(trans_{P_1}(v))$. The translator can performs composition using the *Let* keyword of PVS for writing complex expressions through recursive substitutions. The program P_1; P_2 can be written as $LET\ v = trans_{P_1}(v)$ IN $trans_{P_2}(v)$.

Example 6.4. Figure 6-4 shows an example that illustrates translation of assignments and conditionals. The effect of the foo transition is a set of arithmetic operations on the state variables x and y. And the effect of bar swaps x and y if they are not equal. The PVS translation is shown in the right column of Figure 6-4. In the transition of bar, x and y are assigned new values only when their values are not equal in the pre-state. Otherwise, they are assigned their previous values.

In [LKLM05, Lim01], an alternative translation method using explicit substitutions is proposed. This approach explicitly specifies the resulting value of each variable in the post-state in terms of the variables in the pre-state [BGL02]. In the substitution method, the translator does the work of expressing the final value of a variable in terms of the values of the variables in the pre-state. In the LET method, the prover has to perform these substitutions to obtain an expression for the post-state in an interactive proof. Therefore, the substitution method is more efficient for theorem proving, whereas the LET method pre-

```
transitions
  internal foo(i : Nat)
    pre ...
    eff x := x + i; y := x × x;
      x := x − 1; y := y + 1;

  internal bar
    pre ...
    eff t := x;
      if x ≠ y
        then x := y;
        else y := t;
      fi;
```

```
trans(m: Moves, v: Vals): Vals =
CASES m OF
  foo(i): v WITH [states :=
    (LET s: States = states(v) WITH [x := x(v) + i] IN
    (LET s: States = states(v) WITH [y := x(v) × x(v)] IN
    (LET s: States = states(v) WITH [x := x(v) -1] IN
    (LET s: States = states(v) WITH [y := y(v) + 1] IN s))))]

  bar: v WITH [states :=
    (LET s: States = states(v) WITH [t := x(v)] IN
    (LET s: States =
      IF x(v) ≠ q y(v) THEN
        (LET s: States = states(v) WITH [x := y(v)] IN
        (LET s: States = states(v) WITH [y := t(v)] IN s))
      ELSE s))]
  ...
ENDCASES
```

Figure 6-4: Discrete transitions in HIOA and their PVS translation.

serves the sequential structure of the program, which is lost with the substitution method. Since the style of translation in some cases may be a matter of preference, we currently support both approaches as an option for the user.

6.2.7 Trajectories

Recall from Section 6.2.5, that the names of the state models of SHIOA $\mathcal{A}$ are declared by the *smodel* constructor in the *Moves* datatype. Each object of type *Moves* that is constructed by *smodel* corresponds to a trajectory of the variables of $\mathcal{A}$ and has three parameters: (1) *name*: the name of the state model, (2) *ltime*: the length of the trajectory, and (3) *tau*: the trajectory itself. The set of all possible trajectories of the variables of $\mathcal{A}$ is defined as the type *Trajs*. Of all these trajectories, those that are actually permitted by $\mathcal{A}$—as defined by its state model invariants, stopping conditions, and the evolve clauses—are specified by the enabling condition of the corresponding move. Given a valuation $v0$ of type *Vals* and an object m of type *Moves* constructed using *smodel*, $\mathit{Enabled}(m, v0)$ returns true only if $\mathit{tau}(m)$ is a trajectory of $\mathcal{A}$ defined by the state model $\mathit{name}(m)$ such that the starting valuation of $\tau(m)$ is $v0$. Hence, $\mathit{Enabled}(m, v0)$ encodes the conditions imposed by the invariant, the stopping condition, and the evolve clause of the state model $\mathit{name}(m)$. And the $\mathit{Trans}(m, v0)$ function for *smodel* is simply $\mathit{tau}(\mathit{ltime})$ which returns the valuation of the variables at the last point in the trajectory.

The conditions imposed by the invariants and stopping conditions can be written directly using logical expressions involving the variables, however, the conditions imposed by the evolve clause typically involve derivatives and integrals of the variables evaluated over $\mathit{tau}(m)$. Currently it is not possible to express such differential and integral constraints on *tau*, because theory of differential and integral calculus is not completely formalized in PVS. Instead, we state the relationship among the variables that result from solving the of the differential and algebraic equations in the evolve clause. The translator solves algebraic equations and a relatively small class of differential equations, namely, those with constant right hand sides. Although limited, this class of DAIs can express systems with drifting clocks; as in the case of distributed, real-time systems. For classes of differential equations outside the above, solutions may be provided manually by the user. In the future, this manual step could be eliminated by directly invoking tools for solving DAEs [GGC81, Mat74].

```
automaton Bounce(ρ : Real)
  where 0 < ρ < 1
 signature
  output bounce

 variables
  internal v : Real := 0;
       x : Real := 100;
  let g = 9.8;

 transitions
  output bounce
   pre x = 0 ∧ v < 0;
   eff v := −ρ × v;

 trajectories
  trajdef motion
  stop when x = 0 ∧ v < 0
  evolve d(v) = −g; d(x) = v;
```

```
Bounce_decls: THEORY BEGIN                                                1

  SModelNames: TYPE = {motion}                                            3
  States: TYPE = [# v:real, x:real #]
  LocalVals: TYPE = [# states:States #]                                   5
  Vals: TYPE = [# states:States #]
  ...                                                                     7
  Start(s:States):bool = (v(s) = 0 AND x(s) = 0)
  g: posreal = 9.8                                                        9
  ...
  Moves: Datatype BEGIN                                                   11
     bounce: bounce?
     smodel(name:SModelNames,ltime:nnreal,tau:Trajs):smodel?              13
  END Moves
                                                                          15
  Enabled(m:Moves, v0:Vals):bool =
   CASES m OF                                                             17
     bounce: x(v0) = 0 AND v(v0) < 0,
     smodel(name,ltime,tau):                                              19
      CASES name OF
       motion: FORALL (t:interval(ltime)):                                21
           x(tau(t)) = 0 AND v(tau(t)) < 0 ⇒ t = ltime
         AND FORALL (t:interval(ltime)):                                  23
           v(tau(t)) = v(v0) −g × t AND
           x(tau(t)) = x(v0) + v(tau(t)) × t                              25
      ENDCASES
   ENDCASES                                                               27

  Trans(m:Moves, v0:Vals):Vals =                                          29
   CASES m OF
     bounce: v0 WITH [states:= states(v0) WITH [v := −rho × v(v0)]]       31
     smodel(name,ltime,tau): tau(ltime)
   ENDCASES                                                               33
  ...
  END Bounce                                                              35
```

Figure 6-5: Bounce automaton translated to *Bounce* theory in PVS.

Example 6.5. In Figure 6-5 we present the key definitions in the PVS translation of the bounce automaton of Chapter 3. The `common_decls.pvs` file (not shown here) defines a real variable, *rho*, which takes values in the range $(0, 1)$. Corresponding to the single state model motion of the Bounce automaton, *SModelNames* is a singleton type. Since this automaton has no external variables, the record for *Vals* consists of a tuple with only states but no output types. The *Moves* datatype declares the names of the actions, in this case bounce, and the state models. The *States* and *SModelNames* types differ for different automata, but the dependent type definitions for *interval*, *Trajs* (see, Section 6.2.4), and the constructor *smodel* are always the same.

The *Enabled* predicate defines whether at a valuation $v0$, a move corresponding to a bounce-transition or a motion-trajectory, is possible. The first case, line 18, is a direct translation of the precondition of the bounce action, namely, $x(v0) = 0 \wedge v(v0) < 0$. Here $x(v0)$ actually means $x(states(v0))$; this shorthand is possible because accessor functions (not shown in the figure) are defined in the translation for accessing the component variables inside the *States* and *OutputVals* records.

The second case (lines 19–26) encodes the invariant, the stopping condition, and the evolve clause of the motion state model. For SHIOAs with multiple state models, the individual models are defined through a nested case statement on the names of the state models. Bounce has just one state model, namely, motion, and its stopping condition and

evolve clause are encoded as a conjunction of two clauses (lines 21–25). The first clause (lines 21–22), encodes the stopping condition. The second clause (lines 33–25), asserts that the values of the state variables v and x over the trajectory tau are according to the solution of the differential equations $d(v) = -g; d(x) = v$ with initial value $v0$.

The *Trans* function defines the valuation of variables when a move m corresponding to a bounce transition or a motion trajectory, occurs at $v0$. The first case (line 31) states that the valuation of the variables after a bounce transition is the same as their valuations at $v0$, except that the states component is modified by replacing the value of v by $-rho \times v(v0)$. The second case (line 32 states that the valuation of the variables after *any* trajectory tau equals their valuation at $tau(ltime)$. The complete *Bounce* theory contains several other standard definitions that are not shown in Figure 6-5.

Example 6.6. In this example we show the PVS translations of the failure detector and its abstract specification from Chapter 3. Figure 6-6 shows PVS theory *Spec_decl*—the translation of the abstract Spec automaton (Figure 3-7). For this example, the *common_decls* theory defines a nonempty type M, a dependent type $Packet = [\#message : M, deadline : nnreal\#]$, and three nonnegative real parameters $u1, u2$, and b. It also imports the theory $sequences[Packets]$. Spec does not have external variables, and so the definitions for *Vals* and *LocalVals* are identical to those in Example 6.5. The *ExternalActions* predicate declares the *Moves* constructed with the $fail$ and the *timeout* constructors to be external actions. The Spec automaton has two state models, namely, normal and failed, and their *Enabled* predicates are defined in the usual way. The expression $clock := clock(v0) + t$ models the solution of the differential equation $d(clock) = 1$ with initial value $clock(v0)$.

The failure detector is implemented by the composition of SHIOAs PeriodicSend and Detector of Figure 3-8, and SHIOA TimedChannel of Figure 3-5. The translation scheme is able to translate composed SHIOAs to PVS , however, the current implementation of the translator requires the input HIOA specification to be a primitive automaton. So, we hand-compose these three automata specification to obtain the Timeout automaton of Figure 6-7. Each of the automata PeriodicSend and Detector has a variable called *clock*. In mechanical composition, these variables are renamed as PeriodicSend.*clock* and Detector.*clock*, and therefore, are distinguished. Here, for the sake of brevity, we call these variables $clocp_p$ and $clock_d$, respectively. Note that this automaton has a single state model, namely normal, which is in fact the combination of three state models: timepassage of TimedChannel, normal of PeriodicSend, and normal of Detector. The stopping condition for the combined state model is the disjunction of the stopping conditions of the component state models.

Figure 6-8 shows PVS theory *Timeout_decl*—the PVS translation of Timeout. We discuss translation of the invariant properties of Timeout in the next section. Two points worth noting in this translation: (1) The *common_decls* theory imports $sequences[Packet]$. This defines the type `sequences` as finite sequences of *Packet*s, and the operations `first`, `rest`, and `add` as the usual operations on `sequences`. (2) The actions send and receive are parameterized and the corresponding constructors, $send(m : M)$ and $receive(m : M)$, have the same formal parameter. The definitions for *Enabled* and *Trans* for *Moves* constructed using $send(m)$ and $receive(m)$ depend on value of the parameter m.

If the **evolve** of an SHIOA state model clause contains a constant differential inclusion of the form $d(x) \leq k$, an additional parameter x_r is introduced in the corresponding move constructor. Then, a fourth conjunction is added with the *Enabled* predicate to assert the restriction $x_r \leq k$. For example, the state model

```
trajdef progress invariant x ≥ 0 stop when x = 10 evolve d(x) ≥ 0; d(x) ≤ 2
```

```
1   Spec_decls: THEORY BEGIN
      IMPORTING common_decls
3
      SModelNames = {normal, failed}
5     States: TYPE = [# last_timeout:time, clock:nnreal, suspected:bool, failed:bool #]

7     Start(s:States):bool = last_timeout(s)= infinity AND clock(s) = 0 AND NOT (suspected(s) OR failed(s))

9     Moves: Datatype BEGIN
        fail: fail?
11      timeout: timeout?
        smodel(name:SModelNames,ltime:nnreal,tau:Trajs):smodel?
13    END Moves

15    Trajectory(m:Moves): bool = smodel?(m)
      Actions(m:Moves): bool = NOT Trajectory(m)
17    ExternalActions(a:(Actions)): bool = fail?(a) OR timeout?(a)

19    Enabled(m:Moves, v0:Vals):bool =
        CASES m OF
21        fail: NOT failed(s),
          timeout: failed(s) AND NOT suspected(s),
23        smodel(name,ltime,tau):
            CASES OF name
25            normal: FORALL (t:interval(ltime)):
                  NOT failed(tau(t)) OR (failed(tau(t)) AND suspected(tau(t)))
27                AND tau(t) = v0 WITH [states := states(v0) WITH [clock := clock(v0) + t]],
              failed: FORALL (t:interval(ltime)): failed(tau(t))
29                AND clock(tau(t)) = last_timeout(tau(t)) ⇒ t = ltime
                  AND tau(t) = v0 WITH [states := states(v0) WITH [clock := clock(v0) + t]]
31          ENDCASES
        ENDCASES
33
      Trans(m:Moves, v0:Vals):Vals =
35      CASES m OF
          fail: v0 WITH [states := states(v0) WITH [failed := true, last_timeout := clock(v0) + u2 + b ]],
37        timeout: v0 WITH [states := states(v0) WITH ⟨last_timeout := infinity, suspected := true] ],
          smodel: tau(ltime)
39      ENDCASES

41    IMPORTING hybrid_machine[States,(Actions),(ExternalActions),(Trajectories),Enabled,Trans,Start]

43  END Spec_decls
```

Figure 6-6: Spec of Figure 3-7 translated to *Spec_decls* theory.

is translated to a move constructor $smodel(name : SModelNames, ltime : \texttt{nnreal}, tau : Trajs, x_r : \texttt{real})$ with the enabling condition:

```
Enabled(m,v0):bool = CASES m OF
    smodel(name,ltime,tau,x_r):
      CASES name OF
        progress: FORALL (t:interval(ltime)):
          x(tau(t)) ≥ 0
          AND (x(tau(t)) = 10 ⇒ t = ltime)
          AND x(tau(t)) = x(v0) + x_r × t
          AND 0 ≤ x_r ≤ 2
```

```
1   automaton Timeout(M :type,u1, u2, b : Real)
      where u1 ≥ 0 ∧ u2 ≥ 0 ∧ b ≥ 0 ∧ u2 > (u1 + b)
3     signature
        internal send(m : M), receive(m : M)
5       output fail, timeout

7     variables
      internal next_send : AugmentedReal := 0;
9       next_recv : AugmentedReal := u2;
        suspected : Bool := false;
11      failed : Bool := false;
        now, clock_p, clock_d : Real := 0;
13      queue : Seq[Packet] := {};

15  transitions
      internal send(m)
17    pre ≠ failed ∧ next_send = clock_p;
      eff next_send := clock_p + u1;
19      queue := queue ⊢ [m, now + b];

21    internal receive(m)
        pre m = head(queue);
23      eff next_recv := clock_d + u2;
          queue := tail(queue);
25
      output fail
27      pre ¬failed;
        eff failed := true;
29        clock_p := ∞;

31    output timeout
        pre ¬suspected ∧ clock_d = next_recv;
33      eff suspected := true;
          next_recv := ∞;

    trajectories                                              36
      trajdef normal
      stop when (next_send = clock_p) ∨                       38
        (next_recv = clock_d) ∨
        (now = head(queue).deadline);                         40
      evolve d(now) = 1;
        d(clock_p) = 1; d(clock_d) = 1;                       42

      invariant of Timeout:                                   44
        clock_p = clock_d = now >= 0;
                                                              46
      invariant of Timeout: clock_d ≤ next_recv ∧
        clock_p ≤ next_send ∧                                 48
        now <= head(queue).deadline;
                                                              50
      invariant of Timeout: (now + u2) >= 0 ∧
        ¬suspected ⇒ next_recv ≠ ∞ ∧                          52
        next_recv ≤ (now + u2);
                                                              54
      invariant of Timeout: (now + u1) >= 0 ∧
        ¬failed ⇒ next_send ≠ ∞ ∧                             56
        next_send ≤ (now + u1);
                                                              58
      invariant of Timeout: ∀n : Nat
        (n ≤ (length(queue) − 1) ⇒                            60
        deadline(queue[n]) ≤ (now + b));
                                                              62
      invariant of Timeout: ¬failed ⇒ if queue ≠ {}
        then head(queue).deadline < next_recv                 64
        else (next_send + b) < next_recv;
                                                              66
      invariant of Timeout: suspected ≥ failed;
```

Figure 6-7: Timeout automaton in HIOA .

6.2.8 Invariants

The *hybrid_machine* theory is a generic SHIOA library theory which provides several essential definitions and lemmas for SHIOAs. This theory is quite similar to the existing TAME theory [AHS98] for timed I/O automaton. In Section 6.3 we will describe in detail how these definitions and lemmas are used by our specialized strategies that automate proofs. In this subsection and the next one, we describe how these definitions are used for stating invariant properties and implementation relations.

Each translated theory imports *hybrid_machine* with the parameters $States, (Actions), (ExternalActions), (Trajectory), Enabled, Trans$, and $Start$, based on which the following functions are defined.

```
Reachable_hidden(v,n:nat): RECURSIVE bool = IF n = 0 THEN Start(local(v))
  ELSE (EXISTS m, v1: Vals): Reachable_hidden(v1,n -1) AND
    Enabled(m,v1) AND v = Trans(m,v1)) ENDIF
MEASURE LAMBDA v,n: n
Reachable(s):bool = EXISTS (n:nat,v): Reachable_hidden(v,n) AND states(v) = s
```

In the above fragment, s, m, and v are global variables of type *States*, *Moves*, and *Vals*, respectively, and *Inv* is any predicate on *States*. The predicate $Reachable_hidden(n, v)$ is defined recursively and it returns true if the valuation v of the variables can be reached through a sequence of n *Moves*. The predicate $Reachable(s)$ returns true if there exists a

```
  Timeout_decls: THEORY BEGIN
2   IMPORTING common_decls

4   SModelNames: TYPE = {normal}
    States: TYPE = [# next_send: time, next_recv: time, suspected: bool,
6     failed: bool, clock_p, clock_d, now: nnreal, queue: sequence #]

8   Start(s: States): bool = next_send(s) = 0 AND next_recv(s) = u2 AND NOT (suspected(s) OR failed(s))
      AND now(s) = 0 AND clock_p(s) = 0 AND clock_d(s) AND queue(s) := {}]
10
    Moves: Datatype BEGIN
12    send(m: M): send?
      receive(m: M): receive?
14    fail: fail?
      timeout: timeout?
16    smodel(name: SModelNames, ltime: nnreal, tau: Trajs): smodel?
    END Moves
18
    Trajectory(m: Moves): bool = smodel?(m)
20  ExternalActions(a: (Actions)): bool = fail?(a) OR timeout?(a)

22  Enabled(m: Moves, v0: Vals): bool =
      CASES m OF
24      send(m): NOT failed(v0) AND next_send(v0) = clock_p(v0),
        receive(m): m = message(first(queue(v0))),
26      fail: NOT failed(v0),
        timeout: NOT suspected(v0) AND next_recv(v0) = clock_d(v0)
28      smodel(name, ltime, tau):
          CASES name OF
30          normal: FORALL (t: interval(ltime)):
            ((next_send(tau(t)) = clock_p(tau(t))) OR (next_recv(tau(t)) = clock_d(tau(t)))
32          OR (now(tau(t)) = head(queue(tau(t))).deadline)) => t = ltime
            AND tau(t) = v0 WITH [states := states(v0)
34            WITH [clock_p := clock_p(v0) + t, clock_d := clock_d(v0) + t, now := now(v0) + t] ]
          ENDCASES
36    ENDCASES

38  Trans(m: Moves, v0: Vals): Vals =
      CASES m OF
40      send(m): v0 WITH [states := states(v0) WITH [next_send := clock_p(v0) + u1,
            queue := add(queue(v0), (# message := m, deadline := clock_p(v0) + b #))] ],
42      receive(m): v0 WITH [states := states(v0) WITH
          [queue := rest(queue(v0)), next_recv := clock_d(v0) + u2] ],
44      fail: v0 WITH [states := states(v0) WITH [failed := true, next_send := infinity],
        timeout: v0 WITH [states := states(v0) WITH [suspected := true, next_recv := infinity],
46      smodel(name, ltime, tau): tau(ltime)
      ENDCASES
48
    IMPORTING hybrid_machine[States, (Actions), (ExternalActions), (Trajectory), Enabled, Trans, Start]
50 END Timeout_decls
```

Figure 6-8: Timeout of Figure 6-7 translated to *Timeout_decls* theory.

valuation v that can be reached in some finite number of *Moves* and its restriction to the set of state variables equals s. Using the above definition of reachability, the 7 invariant assertions of Timeout are translated to PVS as follows: Each invariants of HIOA is translated to a named predicate on *States*. This involves straightforward syntactic transformations. The lemma following each predicate asserts that the predicate is indeed an invariant of Timeout, that is, every reachable state satisfies it (see, Definition 4.2). These lemmas appear in a separate theory called *Timeout_invariants* (Figure 6-9) which is also produced by the translator.

```
Timeout_invariants: THEORY BEGIN
  Inv_1(s:States):bool = clock_p(s) = clock_d(s) = now(s) ≥ 0
  Lemma_1: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_1(s);

  Inv_2(s:States):bool = clock_d(s) ≤ next_recv(s) AND clock_p(s) ≤ next_send(s)
    AND now(s) ≤ deadline(first(queue(s)))
  Lemma_2: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_2(s);

  Inv_3(s:States):bool = now(s) + u2 ≥ 0 AND NOT suspected(s)
    ⇒ next_recv(s) ≠ infinity AND next_recv(s) ≤ now(s) + u2
  Lemma_3: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_3(s);

  Inv_4(s:States):bool = now(s) + u1 > = 0 AND NOT failed(s)
    ⇒ next_send(s) /=  infinity AND next_send(s) ≤ now(s) + u1
  Lemma_4: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_4(s);

  Inv_5(s:States):bool = FORALL (n: nat):
    n ≤ length(queue(s)) -1 ⇒ deadline(nth(queue(s), n)) ≤ now(s) + b
  Lemma_5: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_5(s);

  Inv_6(s:States):bool = b > = 0 AND NOT failed(s) ⇒
    IF queue(s) ≠ {} THEN deadline(first((queue(s)))) < next_recv(s)
      ELSE next_send(s) + b < next_recv(s) ENDIF
  Lemma_6: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_6(s);

  Inv_7(s:States):bool = suspected(s) ⇒ failed(s)
  Lemma_7: LEMMA FORALL (s:States): Reachable(s) ⇒ Inv_7(s);
END Timeout_invariants
```

Figure 6-9: Translation of invariant assertions for Timeout.

6.2.9 Simulation Relations

In this section, we describe how simulation relations between pairs of SHIOAs are translated to PVS. Simulation relations involve a pair of automata, and hence to express them, one needs a way to represent abstract automata objects. In order to motivate our approach, we first discuss the difficulties one encounters in accomplishing this in PVS.

An SHIOA in PVS is determined by instantiations of the types *States*, *OutputVals*, *Moves*, *ExternalActions*, *Enabled*, *Trans*, etc. These elements can be thought of as fields of an abstract tuple, and an automaton object can be thought of as an instance this tuple type. However, such a tuple type is not possible in PVS, because the fields of a tuple in PVS are not permitted to have type “type”. Further, unlike Isabelle/HOL [Pau93], PVS does not support parametric polymorphism. Another possible way to support the definition of simulation relations between two automata is to create simulation relation templates that import two automata theories (together with their associated invariants), and then require the user to tailor certain details of a definition of the simulation property to match the details of the automata. This approach is awkward for the user, who must carefully tailor complex definitions to specific cases following PVS naming conventions. It is also awkward for developing proof strategies, which would need to figure-out the details of the tailored definitions.

We circumvent these issues by making use of a relatively new feature of PVS called *theory instantiations* [OS01]. Our approach is both straightforward for users and clean from the point of view of strategy developers. This support relies on (a) a generic theory *SHIOA*, (b) a library of *property theories* defining relations, such as, forward simulation, refinement, and (c) a *template* for translating simulation relation statements to PVS theorems.

```
SHIOA:THEORY BEGIN
  States: TYPES+
  ExternalVals: TYPES
  LocalVals: TYPES
  Moves: TYPES+
  Actions(m:Moves):bool
  Trajectories(m:Moves):bool
  InternalActions(a:(Actions)):bool
  ExternalActions(a:(Actions)):bool
  Start(s:States):bool
  Enabled(m:Moves, v:Vals):bool
  Trans(m:Moves, v:Vals):Vals

  Trajs(t:nnreal): TYPES
  ExtTrajs(t:nnreal): TYPES
  states(v:(Vals)):States
  exttraj(m:(Trajectories)):ExtTrajs

  Reachable(s:States):bool
  InvisibleSeqReach(s,s1:States):bool
  VisibleContReach(s,s1:States,t:nnreal,
    trace:ExtTrajs):bool
END SHIOAutomaton
```

Figure 6-10: *SHIOA* PVS theory.

The SHIOA Theory

Figure 6-10 shows the *SHIOA* theory, which declares the elements that specify an SHIOA. Other than the pattern of type restrictions, the *SHIOA* theory makes no other restrictions on the elements. Note that the translation scheme we have described in Sections 3.2-3.6, precisely defines these elements for particular SHIOAs. That is, the names and types of elements in *SHIOA* theory match exactly the names and types of the definitions that appear in PVS translation of HIOA specifications. Henceforth, we shall say that a PVS theory obtained by translating a HIOA automaton specification, such as *Timeout_decls*, is an instance of the *SHIOA* theory. Instantiation of *SHIOA* provides concrete definitions of the *SHIOA*'s elements. The concrete definitions for the first set of these elements are written by the user when writing a HIOA specification and then this information is translated to the corresponding PVS theory. The remaining elements are the same for all SHIOAs automata and are defined in the *hybrid_machine* library library. The user does not have to redefine them for specific automata, because an appropriate instance of `hybrid_machine` is imported in the translated specification. We have defined all but the last two elements in this theory. The predicate $\mathit{InvisibleSeqReach}(s1, s2)$ is true if and only if there exists an execution fragment $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots \tau_n$ such that $\alpha.fstate = s1$, $\alpha.lstate = s2$, all the τ_i's are point trajectories and all the a_i's are internal actions. Finally, $\mathit{VisibleContReach}(s1, s2, t, trace)$ is *true* if and only if there exists an execution fragment $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots \tau_n$ such that $\alpha.fstate = s1$, $\alpha.lstate = s2$, $\alpha.ltime = t$, all the a_i's are internal actions, and the restriction of α to external variables and actions equals *trace*. Both these predicates is defined inductively in terms of the number of internal actions (and trajectories) constituting α.

Property Theories

The *Forward_simulation* theory shown in Figure 6-11 is a typical *property theory*; it defines what it means for a relation R to be a forward simulation relation between a pair of generic SHIOAs. In the same spirit we are developing property theories for homorphism, weak refinement, switching simulation, and backward simulation—all of which involve pairs of SHIOAs. The first two parameters *Forward_simulation* are of type THEORY *SHIOA*. This means that

IMPORTING *Forward_simulation*[*Theory1_decls*, *Theory2_decls*, . . . , . . .]

automatically instantiates the elements of *SHIOA* theory with two sets of definitions, corresponding to those appearing in *Theory1_decls* and *Theory2_decls*. For example, the type

A.States, is matched with the type *States* of *Theory1_decls*. The third parameter R is a relation on the states of A and B, which is the putative forward simulation relation. In the mathematical definition of a forward simulation (Definition 4.3), every move of A is matched by a sequence of moves of B *with the same trace* such that the final states are related. Since the type *ExternalActions* (and *ExternalVals*) of A is defined completely independently of the type *ExternalActions* (*ExternalVals*, resp.) of B, we cannot equate an external action (trajectory of an external variables) of A with that of B. For this reason, we have to map each external action (variable) of A to an external action (variable) of B. The last two parameters of *Forward_simulation*, namely, *ExtActMap* and *ExtValMap*, provide these maps. While often these functions map an action (variable) of A to an action (variable) of B with the same name, they do provide a mechanism for relating automata with completely different action (and variable) names.

Recall the definition of a forward simulation relation from Chapter 4. The first condition of Definition 4.3, that is, the base case is captured by the *FwdSimBase* predicate (line 10). The second and the third conditions of are combined into the *FwdSimInd* predicate (lines 12–24). *FwdSimInd* returns true if for all states $s_A, s1_A$ of A, state s_B of B and for all moves m of A the following three conditions hold: (1) s_A and s_B are related by R and are reachable states of A and B, respectively (line 12). (2) There exist appropriate valuations of the variables of A such that the move m takes the state s_A to $s1_A$ (line 14). (3) And Finally, based on the type of the move m one of the following conditions must hold (lines 15–24):

Case 1: m is an external action of A (lines 15–18). There exist intermediate valuations $v1_B, v2_B$, and a state $s1_B$ of B such that $s1_A$ and $s1_B$ are related by R, and (i) s_B can go to $states(v1_B)$ through a sequence of internal actions (no time passage), (ii) $states(v2_B)$ can go to $s1_B$ through a sequence of internal actions, (iii) $v1_B$ goes to $v2_B$ by performing the external action $ExtActMap(m)$.

Case 2: m is an internal action of A (line 20). There exists a state $s1_B$ of B such that $s1_A$ and $s1_B$ are related by R and s_B can go to $s1_B$ through a sequence of internal actions.

Case 3: m is a trajectory of A (line 23). There exists a state $s1_B$ of B and a nonnegative real number t, such that $s1_A$ and $s1_B$ are related by R and s_B can go to $s1_B$ through a sequence of alternating trajectories and internal actions such that the trace of the whole sequence is $ExtVarMap(tau(m))$ and has length t.

The *FwdSim* predicate combines *FwdSimBase FwdSimInd* for the given pair of SHIOAs, the relation R, and the external action and external variable maps.

Translation of Simulation Relations to PVS

With the *SHIOA* and the *Forward_simulation* theories in place, we are in a position to translate HIOA statements that define simulation relations between a pair of automata. The translation takes the form of a separate theory such as the one shown in Figure 6-12. The PVS theories generated by translating each of the HIOA automata specifications are imported, and then these theories are interpreted as instances of the *SHIOA* theory using the syntax

$$MA\colon \text{THEORY} = SHIOA \;:\to\quad Timeout_decls$$

```
1   Forward_simulation[A,B: THEORY SHIOA, R: [A.States, B.States → bool ]
      ExtActMap: [(A.ExternalActions) → (B.ExternalActions)],
3     ExtVarMap: [(A.ExternalVals) → (B.ExtrnalVals)] ]: THEORY BEGIN

5   m : VAR A.Moves
    s_A, s1_A: VAR A.States
7   s_B, s1_B: VAR B.States
    v1_B,v2_B: VAR B.Vals
9
    FwdSimBase: bool = FORALL s_A: A.Start(s_A) ⇒ EXISTS(s_B): B.Start(s_B) AND R(s_A,s_B)
11
    FwdSimInd: bool = FORALL s_A,s1_A,s_B,m: A.Reachable(s_A) AND B.Reachable(s_B) AND R(s_A,s_B)
13    AND EXISTS v_A,v1_A: states(v_A) = s_A AND states(v1_A) = s1_A
        AND A.Enabled(m,s_A) AND v1_A = A.Trans(m,v_A) ⇒
15    (A.ExternalActions(m) ⇒ EXISTS (v1_B,v2_B,s1_B):
        B.InvisibleSeqReach(s_B,states(v1_B)) AND B.InvisibleSeqReach(states(v2_B),s1_B) AND
17      B.Enabled(ExtActMap(m),v1_B) AND
        B.Trans(ExtActMap(m),v1_B) = v2_B AND R(s1_A, s1_B))
19    AND
      (A.InternalActions(m) AND A.Actions(m)) ⇒ EXISTS (s1_B):
21      B.InvisibleSeqReach(s_B,s1_B) AND R(s1_A, s1_B)
      AND
23    (A.Trajectories(m) ⇒ EXISTS (t:nnreal, s1_B):
        B.VisibleContReach(s_B,s1_B,t,ExtVarMap(t,exttraj(m))) AND R(s1_A, s1_B))
25
    FwdSim:bool = FwdSimBase AND FwdSimInd
27 END Forward_simulation
```

Figure 6-11: *Forward_simulation* theory.

This matches the elements of *SHIOA* with their corresponding definitions in *Timeout_decls* and makes them available as components of MA. The function *ExtActMap* maps the two external actions of *Timeout_decls*, namely, $fail$ and *timeout*, to the identically named external actions of *Spec_decls*. Since neither automata have external variables, the function *ExtVarMap* is declared but is left uninterpreted. It does not impose any constrains apart from the fact that a trajectory move m of MA should be matched with a sequence of trajectories and actions of MB with a total length $ltime(m)$. The relation R on the states of MA and MB is the putative forward simulation relation from Timeout to Spec. Importing *Forward_simulation* defines the *FwdSim* predicate of Figure 6-11 for the automaton theories *Timeout_decls* and *Spec_decls*. Finally, the *FwdSimThm* theorem simply asserts that the *FwdSim* predicate holds, that is, R satisfies all the conditions to be a forward simulation relation.

6.3 Strategies

In the previous section we have described how HIOA specifications of automata and their properties such as invariant assertions and simulation relations can be translated to PVS . Once translated, the proofs of theorems and lemmas, such as those in Figures 6-9 and 6-12, can be constructed interactively by invoking the PVS theorem prover. The PVS theorem prover provides a set of general proof commands for manipulating logical formulas. Basic proof commands include commands for simplification of expressions, expansion of definitions, and instantiation of existentially quantified variables. The PVS prover also provides more powerful commands for automated theorem proving such as general semi-decision procedures for first and higher order logics and an infinite bounded model-checker. Notwithstanding the power of the PVS prover, we have found that construction of invariant

```
Timeout_Spec: THEORY BEGIN
  IMPORTING Timeout_decls, Spec_decls
  MA: THEORY = SHIOA :→  Timeout
  MB: THEORY = SHIOA :→  Spec_decls

  ExtActMap(a:(MA.ExternalActions)):(MB.ExternalActions) =
    CASES a OF
      fail: fail,
      timeout: timeout
    ENDCASES

  ExtVarMap(t:nnreal,w:(MA.ExtTrajs(t))):(MB.ExtTrajs(t))

  R(s_A: MA.States, s_B: MB.States): bool =
    failed(s_A) = failed(s_B) AND
    suspected(s_A) = suspected(s_B) AND
    now(s_A) = now(s_B) AND
    now(s_A) = clock_p(s_B) AND
    now(s_A) = clock_d(s_B) AND
    IF NOT failed(s_B) THEN last_timeout(s_B) = infinity
      Elsif queue(s_A) = {} THEN last_timeout(s_B) ≥ deadline(latst(queue(s_A))) + u2
      ELSE last_timeout(s_B) ≥ last_recv(s_A)
    ENDIF

  IMPORTING Forward_simulation[MA, MB, ref, ExtActMap, ExtVarMap]
  FwdSimThm: THEOREM FwdSim

END Timeout_Spec
```

Figure 6-12: Translation of forward simulation relation of failure detector.

and simulation proofs for SHIOAs is a labor intensive, and often tedious process, with many repetitive steps. Our translated theories, on the one hand, are complex because the HIOA specifications often have complicated user-defined types, and higher-order expressions in preconditions, stopping conditions, and effects; but, on the other hand, the proofs of the lemmas are often very well structured and they follow from applications of Lemmas 4.2 and 4.4 (as examples of such structured proofs, see the case studies of Chapters 4 and 5). The work involved in constructing proofs in PVS can be significantly reduced by building custom-made proof command or *strategies* that exploit the structure in SHIOA proofs and automate many of the repetitive steps.

In this section, we present the principles underlying the development of such proof strategies. First, in Section 6.3.1 we discuss the strategies for proving invariants. These strategies were developed by Archer [Arc01] for constructing proofs of Timed Automata. We discuss them here because they are compatible with the SHIOA model and also because the experience gained from their design informed the later development of the strategies for proving refinement and simulation relations. The latter strategies are discussed in Section 6.3.2. It is worth mentioning that although others have used PVS to construct invariant and simulation proofs for I/O Automata [DGRV00], however, to our knowledge, no one has developed generic PVS strategies to automate construction of such proofs.

6.3.1 Strategies for Proving Invariants

SHIOA strategies for proving invariant properties are derived from earlier TAME strategies. The strategy essentially relies on the following PVS-version of Lemma 4.2, called theorem *Machine_induct*, which appears in the *hybrid_machine* library theory.

$Base(Inv)$:bool = FORALL (v): $Start(local(v)) \Rightarrow Inv(states(v))$
$Inductstep(Inv)$: bool = FORALL v, m: $Reachable(states(v))$ AND $Inv(states(v))$
  AND $Enabled(m, v) \Rightarrow Inv(states(Trans(m, v)))$;
$Inductthm(Inv)$: bool = $Base(Inv)$ & $Inductstep(Inv) \Rightarrow$
  FORALL s: $reachable(s) \Rightarrow Inv(s)$;
$Machine_induct$: THEOREM (FORALL Inv: $Inductthm(Inv)$);

Notice that the transition and the trajectory conditions of Lemma 4.2 are merged into a single *Inductstep*. Thus, to prove that a predicate on states Inv is an invariant, it suffices to check that $Base(Ind)$ and $InductStep(Ind)$. The `Auto_Induct` strategy, the primary strategy for proving invariant properties, works as follows: it breaks down the proof of invariance of a given predicate Inv into several subgoals: (a) the base case $Base(Inv)$, and (b) one subgoal for each constructor of *Moves* type. Some of subgoals of the second type are discharged automatically using PVS's simplification commands; for example, *Moves* that are not enabled or those that do not alter the variables involved in Inv. For the remaining subgoals, `auto_induct` invokes the `Apply_Precond` sub-strategy, which asserts the *Enabled* predicate of the move, and attempts to simplify the resulting expressions using the `Try_Simp` simplification strategy or by using PVS's decision procedures. Harder subgoals require more careful user interaction in the form of using previously proved invariants and instantiating formulas.

In branches involving moves that are *Trajectories*, the post-state is the last state of the trajectory $tau(m)$. In order to prove that the invariant holds at $tau(m)(ltime(m))$, the inductive hypothesis is used along with the enabling condition of m. Recall that the enabling condition states that (i) at every point t in $interval(ltime)$ the invariant condition for the state model $name(m)$ must hold, (ii) if the stopping condition holds at some t in $interval(ltime)$, then t equals $ltime$, and that (iii) $tau(t)$ is a solution of the differential equations described in the state model of $name(m)$. Recall that the invariant condition is the predicate associated the state model $name(m)$, and is *different* from the invariant Inv of the automaton that the user is attempting to prove. For SHIOAs with general dynamics, at this point, the user has to use the PVS prover and deduce that the invariant holds at $tau(m)(ltime(m))$ from the above facts. Many of the subgoals reduce to reasoning about inequalities over real numbers and for completing such proofs the strategies for manipulating real inequalities provided by the Field [MM01] and the Manip [Vit02] packages are useful.

For proving convex invariants of rectangular SHIOAs we have a special strategy called `Deadline_Check` which exploits the monotonicity of the solutions of the state models to further automate this proof step. This strategy describes the post-state, that is, the valuation of each continuous variable x at $tau(ltime)$, as $x(tau(ltime)) = x(v0) + k \times ltime(tau)$, and then performs the simplifications to check if $tau(ltime)$ satisfies the invariant. For example, a commonly occurring type of invariant asserts that a variable x, evolving according to differential equation $d(x) = k$, does not cross a deadline, say d (see e.g., Invariants 2 and 3 of Figure 6-9). The proof branch corresponding to a trajectory move m of such an invariant, is split into two cases by the `Deadline_Check`, namely, (a) $ltime(m) \leq \frac{d-x(v0)}{k}$ and (b) $ltime(m) > \frac{d-x(v0)}{k}$. For case (a) it is deduced automatically that at the post state, $x(tau(m)(ltime(m))) = x(v0) + ltime(m) \times k \leq x(v0) + d - x(v0) \leq d$, that is, it satisfies the invariant. For case (b), by deducing $false$ from $Enabled(m, v0)$, a contradiction is reached. Internally, `Deadline_Check` performs the following checks: $Enabled(m, v0)$ requires that, forall $t \in interval(ltime(m))$, if $x(tau(m)(t)) = k$, then $t = ltime(m)$. From (i) inductive hypothesis, $x(v0) \leq k$, (ii) instantiation of stopping

$x(tau(m)(ltime(m))) > x(v0) + k \times \frac{d-x(v0)}{k} = k$, and (iii) monotonicity of $x(tau(m))$, that is, $\forall t \in interval(ltime(m)), x(tau(t)) = x(v0) + k \times t$, it is deduced that there exists $t \in interval(ltime(m))$, such that $x(tau(t)) = d$ but $t \neq ltime(m)$—a contradiction with $Enabled(m, v0)$. Thus, in case (b) m is not a valid move of the automaton in question and therefore this branch of the induction proof is complete.

6.3.2 Strategies for Proving Forward Simulation

In this section, we discuss strategies we have developed for proving forward simulation relations for SHIOAs. These strategies were originally developed for Timed I/O Automata along with several other strategies for proving weak-refinement [MA03, MA05].

The main strategy for proving forward simulations, as defined by the *Forward_simulation* theory, is called `Prove_Fwd_Sim`. The generic nature of the definition of the *FwdSim* property allows us to define `Prove_Fwd_sim` in such a way that it can be applied to an arbitrary simulation proof between any given pair of SHIOAs. This strategy is designed to perform much of the work, for an arbitrary simulation theorem such as the one stated at the end of the *Timeout_Spec* theory. The structure of `Prove_Fwd_Sim` in terms of sub-strategies is shown in Figure 6-16. The branches terminating in blue circles signify that the if the preceding sub-strategy fails to complete the proof, the resulting subgoals are handed over to the user for constructing their proofs interactively. First, `Prove_Fwd_Sim` splits the predicate *FwdSim* into *FwdSimBase* and *FwdSimInd* and applies `Setup_Sim_Base` and `Setup_Sim_Induct_Cases` sub-strategies to the branches, respectively. `Setup_Sim_Base` performs the standard steps needed in the base case, including skolemization, expansion of the definitions *Start* and *R*, and simplifications. The base case sequent, which is handled by `Setup_Sim_Base`, is shown in Figure 6-13, in which `Timout_decls.Start` and `SPEC_decls.start` are the start predicates of Timeout and Spec, respectively. `Prove_Fwd_Sim`

```
;;;Base case
{-1} Timeout_decls.Start(s_A)
|-------
{1} Exists (s_B:Spec_decls.States): Spec_decls.Start(s_B) And R(s_A,s_B)
```

Figure 6-13: Base case sequent of simulation proof.

then probes the proof state to check if the base case can be discharged trivially, and then, it moves over to the next branch of the proof, namely, the application of `Setup_Sim_Induct_Cases` to *FwdSimInd*. If a proof subgoal is not completely resolved by the application of a strategy, then the final set of sequents produced by the strategy are presented to the user for interactive proving.

The sub-strategy `Setup_Sim_Induct_Cases` performs skolemization and expands the definitions of *InternalActions*, *ExternalActions*, and *Trajectories* and splits up the induction step into a set of subgoals—one for each constructor of the *Moves* datatype. Based on whether the constructor classifies as *ExternalActions*, *InternalActions*, or *Trajectories*, the `Sim_Induct_ExtAction`, the `Sim_Induct_IntAction`, or the `Sim_Induct_Traj` sub-strategies are applied. In what follows, we will discuss each of these sub-strategies, but first we discuss a strategy called `State_Trans` which is a building-block for all these strategies. The `State_Trans` strategy is used for constructing states by applying a sequence of moves to a given state. Thus, this strategy is useful for proving the *InvisibleSeqReach* predicate which appear in the definition of *FwdSimInd*. This strategy takes a sequence of action, say

$\sigma = a_0, a_1, \ldots, a_n$, a starting valuation v_1, and a target state s_2 as parameters and produces the following set of subgoals:

- $s_2 = state(\mathit{Trans}(a_n, \mathit{Trans}(a_{n-1}, \mathit{Trans}(a_{n-2}, \ldots, \mathit{Ttrans}(a_1, v_1) \ldots))))$,
- For each $i \in \{1, \ldots, n\}$, $\mathit{InternalActiona}(a_i)$, and
- For each $i \in \{1, \ldots, n\}$, $\mathit{Enabled}(a_i, \mathit{Trans}(a_{i-1}, \mathit{Trans}(a_{i-2}, \ldots, v_1) \ldots)$.

For an appropriate sequence of internal actions, the strategy then discharges the first and the second set of subgoals by expanding the definitions of *InternalActions*, *Enabled*, and *Trans*. The remaining subgoals are then presented to the user with properly labeled sequents. Instances of this strategy are used by the `Sim_Induct_ExtAction` and the `Sim_Induct_IntAction` sub-strategies as discussed below.

The `Sim_Induct_IntAction` strategy is designed for proving the second case (line 21) in the definition of *FwdSimInd* of Figure 6-11. This strategy prompts the user for a sequence

```
;;;Induction step: internal action Case "send"
[-1, (reachable A.prestate)] A.Reachable(s_A)
[-2, (reachable B.prestate)] B.Reachable(s_B)
[-3, (related A.prestate B.prestate)] R(s_A,s_B)
[-4, (send A.move)] send?(m)
[-5, (enabled A.action)] A.Enabled(m,s_A)
[-6] s1_A = states(v1_A)
[-7] v1_A = A.Trans(m,v_A)
|-------
{1} Exists (s1_B:Spec_decls.States): B.InvisibleSeqReach(s_B,s1_B) And R(s1_A, s1_B))
```

Figure 6-14: Inductive step sequent for internal actions.

of internal actions σ of automaton B. This is the sequence of actions that will be used to match the move m of automaton A and this sequence will take state s_B to $s1_B$. Formally, $s1_B$ is instantiated as $\mathit{Trans}(\sigma, s_B)$ and the branch generates the subgoals: (i) s_B reaches $s1_B$ through a sequence of internal actions, (ii) $s1_B$ and $s1_A$ are related by R.

For example, consider the sequent it Figure 6-14, which is an intermediate subgoal in the inductive branch corresponding to moves constructed using the *send* constructor of *Timeout*. This corresponds to the proof branch in a hand-proof for an the internal send action of Timeout. The first line of the sequent is a comment added by the strategy which indicates the current branch of the proof. All the variables appearing in the antecedent are constants resulting from skolemizing universally quantified expressions. Suppose in proving this branch, the user invokes the `Sim_Induct_IntAction` strategy with parameter " ", indicating that the send action of Timeout is emulated by Spec by an empty sequence of actions. Then, $s1_B$ is instantiated as `State_Trans(" ",`s_B`)` which of course is identical to s_B. In the resulting sequent, $s1_B$ is replaced by s_B; $B.\mathit{InvisibleSeqReach}(s_B, s_B)$ holds trivially and is proved automatically by the `State_Trans` strategy. Therefore, sequent reduces to proving $R(s1_A, s_B)$ which is deduced from the hypothesis $R(s_A, s_B)$ and from the definition of *Trans*. This last bit of simplification is performed by the `Do_Trans` sub-strategy which computes the post-state from a pre-state by applying a transition and repeatedly simplifies. In this case, $s1_A$ is computed by applying a *send* transition to s_A. In general, when `State_Trans` strategy is called with a sequence of actions of length n, then for each action an enablement-subgoeal is produced which is delegated to an `Enablement`

sub-strategy. The `Enablement` sub-strategy expands the definition of the *Enabled* predicate and simplifies to prove that the move in question is enabled at a given state.

The `Sim_Induct_ExtAction` sub-strategy is used to prove branches corresponding to moves that are *ExternalActions*. This strategy prompts the user for two sequences of internal actions, say σ_a and σ_2. These sequences are used to construct a sequence $\sigma_1 \mathit{ExtActMap}(m)\sigma_2$, which matches the move m of automaton A.

Consider a typical sequent of this type as shown in Figure 6-15 which is an inductive branch corresponding to moves constructed using the $fail$ constructor. As in the case of internal actions, the first line of the sequent is a comment added by the strategy and all the variables appearing skolem constants. The requirement for this branch is to show the existence of a valuations $v1_B, v2_B$, and state $s1_B$, such that (i) s_B reaches $v1_B$ through a sequence of internal actions, (ii) $v2_B$ reaches $s1_B$ through a sequence of internal actions, (iii) $v1_B$ reaches $v2_B$ through an action $\mathit{ExtActMap}(m)$, and (iv) $s1_B$ and $s1_A$ are related by R. `Sim_Induct_ExtAction` instantiates $v1_B, v2_B$, and $s1_B$ by applying `State_Trans` strategy as follows: $v1_B = \texttt{State_Trans}(\sigma_1, s_B)$, $v2_B = \mathit{Trans}(\mathit{ExtActMap}(m), v1_B)$, and $s1_B = \mathit{states}(\texttt{State_Trans}(\sigma_2, v2_B))$. Once

```
;;;Induction step: external action Case "fail"
[-1, (reachable A.prestate)] A.Reachable(s_A)
[-2, (reachable B.prestate)] B.Reachable(s_B)
[-3, (related A.prestate B.prestate)] R(s_A,s_B)
[-4, (fail A.move)] fail?(m)
[-5, (enabled A.action)] A.Enabled(m,s_A)
[-6] s1_A = states(v1_A)
[-7] v1_A = A.Trans(m,v_A)
|-------
{1} Exists (v1_B,v2_B:Spec_decls.Vals,s1_B:Spec_decls.States):
 B.InvisibleSeqReach(s_B,states(v1_B)) And B.InvisibleSeqReach(states(v2_B),s1_B) And
 B.Enabled(ExtActMap(m),v1_B) And B.Trans(ExtActMap(m),v1_B) = v2_B And R(s1_A, s1_B))
```

Figure 6-15: Inductive step sequent for external actions.

instantiated, the resulting sequent is split into the above four branches. Branches (i), (ii) and (iv) are handled the same way as in the case of internal actions. Branch (iii) requires an additional application of the `Enablement` and the `Do_Trans` strategies. All these applications are automated by the `Sim_Induct_ExtAction` strategy and any subgoals that are not completely proved are handed back to the user.

The `Sim_Induct_Traj` strategy is designed for proving the last case (line 23) in the definition of *FwdSimInd* of Figure 6-11. At the onset, the trajectory spits up the proof into several cases, one for each element in the type *SModelNames*, and applies the `Induct_Smodel` sub-strategy to each case. For example, the intermediate sequent in Figure 6-17, appears in the inductive branch corresponding to trajectory moves of *Timeout* at the point in which `Induct_Smodel` is invoked. In Timeout there is a single state model, that is, *SModelNames* is a singleton type and therefore this is the only branch corresponding to trajectory moves. Note that the fifth expression in the antecedent asserts the name of the state model branch. The `Induct_Smodel` allows the user to construct an execution fragment of B that emulates the trajectory $tau(m)$ of A. The strategy prompts the user for a four parameters: (a) a natural number n, (b) a sequence $s_0, \ldots, s_n$ of elements of type *SModelNames*, (c) a sequence $a_1, \ldots, a_{n-1}$ of internal actions, and (d) a sequence $t_0, \ldots, t_n$ of positive reals summing to $ltime(m)$. From these parameters it implicitly constructs an execution fragment $\tau_0 a_1 \tau_1 a_2 \ldots \tau_n$ of $n+1$ trajectories and n transitions. The first trajectory τ_0 is defined as

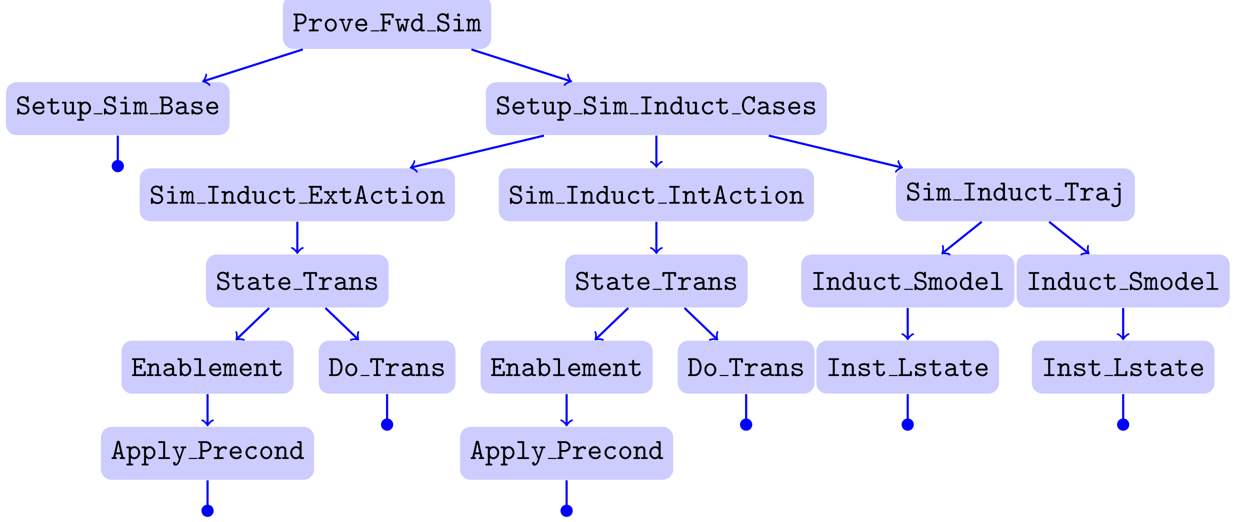

Figure 6-16: Structure of `Prove_Fwd_Sim` strategy.

follows: $\tau_0.fval = v_A$, $\tau_0.ltime = t_1$, and the intermediate states of τ_0 are defined by the evolve clause of the state model s_0. Then, `Induct_Smodel` instantiates the variables in the last expression of Figure 6-17, t with $t_0 + t_2 \ldots + t_n$ and $s1_B$ with $\tau_n.lstate$ and generates a set of subgoals for checking the above conditions and the following additional conditions: (i) the trace of α equals $ExtVarMap(t, exttraj(m))$, and (ii) $s1_B$ and $s1_A$ are related by R. Much of the simplifications and computations in each of these branches are preformed by

```
;;;Induction step: trajectory Case "normal"
[-1, (reachable A.prestate)] A.Reachable(s_A)
[-2, (reachable B.prestate)] B.Reachable(s_B)
[-3, (related A.prestate B.prestate)] R(s_A,s_B)
[-4, (smodel A.move)] smodel?(m)
[-5, (normal A.smodelname)] name(m) = normal
[-6, (enabled A.action)] A.Enabled(m,s_A)
[-7] s1_A = states(v1_A)
[-8] v1_A = A.Trans(m,v_A)
|-------
{1} Exists (t:nnreal, s1_B): B.VisibleContReach(s_B,s1_B,t,ExtVarMap(t,exttraj(m))) And R(s1_A, s1_B)
```

Figure 6-17: Inductive step sequent for trajectories.

the `Inst_Lstate` sub-strategy which checks *Enabled* predicate and computes the transition functions for each a_i and τ_i in the sequence.

For example, in proving this branch, the user invokes `Induct_Smodel` strategy with parameters “0”,“normal”,“_”, and “ltime(m)”, indicating that the trajectory $tau(m)$ constructed using state model normal of Timeout is emulated by a single trajectory, of state model normal and duration $ltime(m)$. In this case, to complete the proof, the user will have to consider two cases: NOT $failed(v_A)$ and $failed(v_A)$, and then in the first case the strategy parameters should be as indicated, and in the latter case the “normal” should be replaced by “failed”.

In the interaction of `Prove_Fwd_Sim` and its substrategies, significant use is made of formula labels, both for deciding which action to take based on the presence or absence of a formula with a given label, and to focus computation on formulae with specific labels. The

labels are designed to be informative: for example, the label `A.enabled` on line 4 of the proof in Figure 6-15 indicates that the enabling condition of the `fail` action of the automaton A (in this case, Timeout). This is so that when an unresolved subgoal is returned to the user, its content is as informative as possible. For the same reason, `Prove_Fwd_Sim` and its substrategies attach comments to any subgoals they create that denote their significance. The comment `;;;Induction step:  trajectory Case "normal"` that appears on line 2 in Figure 6-17 indicates that this subgoal is the subgoal for trajectories of the state model normal.

6.4 Discussion of Case Studies

Our approach to mechanizing proofs for the SHIOA framework is to study examples of SHIOA specifications and their properties and then identify a standard set of proof steps sufficient to mechanize proofs of those properties. To this end, we have targeted invariant properties and simulation relations and the outcome has been the strategies such as `Prove_Fwd_Sim`, `Auto_Induct`, and `State_Trans`. These strategies do not completely eliminate human intervention but they do eliminate many of the repetitive and tedious steps that are necessary for constructing proofs.

Although the translation scheme outlined in Section 6.2 is suitable for any HIOA specification conforming to the assumptions in Section 6.2.1, the current implementation of our translator tool is able to translate the restricted class of HIOA specifications with rectangular dynamics, that is, differential equations with constant right hand sides. As a result, the examples we have studied thus far all have state models with relatively simple continuous dynamics. Nevertheless, the case studies have interesting data structures and demonstrate non-trivial hybrid behavior. We discuss our experience in applying these strategies to two case studies.

6.4.1 Failure Detector

The HIOA specification of a simple failure detector system from [KLSV05] was presented in Figure 6-7. The details of the verification of this system appears in [MA05]. The composite specification consists of (1) a process that sends a message every u_1 time units until it fails, (2) a communication channel that delays messages by at most d time units, and (3) a detector process which times out and suspects the sending process to have failed when it does not receive a message for $u_2 = u_1 + d$ time units. We are interested in proving two properties for this system, namely, (i) a no false positive property, i.e., whenever the detector suspects, the sender is indeed failed, and (ii) a timely detection property, i.e., whenever the sender fails, the detector suspects it within $u_2 + b$ time units. Property (i) is a predicate on the states of the composed system and is stated as an invariant. For describing property (ii), an abstract specification Spec of Timeout is written (shown in Figure 3-7), and then it is proved, using forward simulation, that Timeout implements Spec. The translator generates PVS theory files for the automaton specifications, the invariants, and the simulation relations (see Figures 6-8, 6-9, 6-12, and 6-6).

We invoke the PVS-prover on these theories to interactively prove the translated lemmas and theorems. Invariants Inv_1 to Inv_4 of Figure 6-9, are relatively simple and are proved automatically by the `Auto_Induct` strategy. Proofs of invariants Inv_5, Inv_6, and Inv_7 require the application of the first four invariants, and consequently, in these cases `Auto_Induct` presents us with a set of subgoals that require manual intervention. At which

point, we use a separate strategy called `Apply_Inv_Lemma` which introduces invariant lemma of our choice and instantiates its state appropriately for the current proof context. In most cases, this completes the proof branch; for some of the branches corresponding to trajectory moves, further simplifications are necessary.

In the simulation proof of *FwdSimThm*, each move branch leads to six subgoals corresponding to the six high level conjuncts in the simulation relation, but the trivial subgoals (e.g., first two subgoals) are discharged automatically by the `Prove_Fwd_Sim` strategy. The remaining subgoals required the application of previously proved invariants and reasoning about inequalities involving real expressions for which we also use the strategies in the Manip [MM01] and the Field [Vit02] packages.

6.4.2 Two-Task Race

In this case study we applied our strategies to prove time bounds through a forward simulation in the two process race system described in [Lyn96b]. The automaton TwoTaskRace (see Figure 6-18) models two processes running in parallel. The main process triggers an action once every t time, where t is between a_1 and a_2. The second process produces a set action within the time interval $[b_1, b_2]$. The main process triggers increment actions, incrementing a *count*, until the set action occurs. After set, the main process triggers decrement actions, which decrements a *count*. When the *count* reaches 0 then the main process triggers a report action, which indicates the end of execution. The property that we investigate is the upper and lower time bounds on the occurrence of report action. The details of the translation and verification of the system appears in [LKLM05].

Here we focus to the upper time bound which is $b_2 + a_2 + \frac{b_2 a_2}{a_1}$. The intuition behind this bound as follows: in order to maximize the value of the counter until set occurs, the main process should increment *count* every a_1 time. Thus, the value of *count* when the set action occurs is b_2/a_1. Thereafter, the maximum time taken to decrement *count* down to zero is $a_2 b_2/a_1$. The latest time when the set action occurs is b_2, and therefore the upper time bound for report is $a_2 + b_2 + a_2 b_2/a_1$.

This time bounds on report are specified as a simple abstract automaton TwoTaskRaceSpec which triggers a report action within the above time bound. A forward simulation relation of Figure 6-19 is used to prove that TwoTaskRace implements TwoTaskRaceSpec. This simulation relation is more complex than that for the Timeout example because the relation between the states differs depending on whether or not the set action has occurred. The occurrence of the set action set the $flag$ variable to true. For example, lines 16–17 relate the deadline variable $first_report$ of TwoTaskRaceSpec to the deadline variable $first_set$ and the counter of TwoTaskRace, for the case where the $flag$ has been set. While a different relation is stated in lines 21–23 for the case where the $flag$ has not been set.

The structure of the proof is similar to that for Timeout and Spec and so `Prove_Fwd_Sim` strategy successfully breaks up the simulation proof into subgoals for the individual actions and trajectories. Applying `Sim_Induct_ExtAction`, `Sim_Induct_IntAction` , and `Sim_Induct_Traj` with proper arguments instantiates and simplifies the move branches.

In our case studies, the proofs translated theories constructed using our strategies require 70% or fewer proof steps than those constructed using raw PVS proof commands. This comparison, however, may not be completely fair because our translation is specifically geared toward making the strategies work efficiently, and it adds extra structures in the generated PVS theories that eliminate the generation of some type-correctness-related sub-

```
automaton TwoTaskRace(a1, a2, b1, b2 : Real)
  where a1 > 0 ∧ b1 ≥ 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
  signature
    internal increment, decrement, set
    output report

  variables
    internal count : Int := 0; flag : Bool := false;
      reported : Bool := false; now : Real := 0;
      first_main : Real := a1;
      last_main : AugmentedReal := a2;
      first_set : Real := b1;
      last_set : AugmentedReal := b2;

  transitions
    internal increment
    pre ¬ flag ∧ now ≥ first_main;
    eff count := count + 1;
      first_main := now + a1; last_main := now + a2;

  internal set
    pre ¬ flag ∧ now ≥ first_set;
    eff flag := true;
      first_set := 0; last_set := ∞;

  internal decrement
    pre flag ∧ count > 0 ∧ now ≥ first_main;
    eff count := count − 1;
      first_main := now + a1; last_main := now + a2;

  output report
    pre flag ∧ count = 0 ∧
      ¬ reported ∧ now ≥ first_main;
    eff reported := true;
      first_main := 0;
      last_main := ∞;

trajectories
  trajdef traj
    stop when now = last_main ∨ now = last_set;
    evolve d(now) = 1;
```

```
automaton TwoTaskRaceSpec(a1, a2, b1, b2 : Real)
  where a1 > 0 ∧ b1 ≥ 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
  signature
    output report

  variables
    internal reported : Bool := false; now : Real := 0;
      first_report : Real := if a2 < b1
        then min(b1, a1) + (b1 − a2) ∗ a1/a2
        else a1;
      last_report : AugmentedReal := b2 + a2 + (b2 ∗ a2/a1);

  transitions
    output report
      pre ¬ reported ∧ now ≥ first_report;
      eff reported := true;
        first_report := 0;
        last_report := ∞;

  trajectories
    trajdef pre_report
      invariant ¬ reported;
      stop when now = last_report;
      evolve d(now) = 1;

    trajdef post_report
      invariant reported;
      evolve d(now) = 1;
```

Figure 6-18: TwoTaskRace automaton and its abstract specification.

goals during the proof process. Further, if small changes to the specification are made after the completion of proofs, then the proofs can be re-run, completely automatically, to check if the proofs still hold or not. Another lesson we have learned is that the details of the specification template that a translator to PVS targets, if chosen carefully, can greatly facilitate the implementation of PVS strategies.

6.5 Summary

We presented "an SHIOA interface" for the PVS theorem prover system. In particular, we have discussed (a) how a large class of HIOA specifications can be translated to the language of the PVS theorem prover and (b) how SHIOA-specific proof strategies can be developed, exploiting the structure of typical SHIOA proofs, that automate proof construction in PVS.

The translation scheme relies on two components, namely, (i) a library of theories defining the semantics of SHIOAs in general, and (ii) a theory template with specific definitions of states, actions, transitions, and trajectories, which is populated by the definitions ap-

```
   TwoTaskRace_TwoTaskRaceSpec: THEORY BEGIN
2    IMPORTING TwoTaskRace_decls, TwoTaskRaceSpec_decls
     MA: THEORY = SHIOA timed_auto_libtimed_automaton :→ TwoTaskRace_decls
4    MB: THEORY = timed_auto_libtimed_automaton :→ TwoTaskRaceSpec_decls

6    ExtActMap(a:(MA.ExternalActions)):(MB.ExternalActions) =
       CASES a_A OF
8        report: report
       ENDCASES
10   ...
     R(s_A: MA.States, s_B: MB.States): bool =
12     reported(s_A) = reported(s_B) AND
       now(s_A) = now(s_B) AND
14     NOT flag(s_A) AND last_main(s_A) < first_set(s_A)
         ⇒ first_report(s_B) ≤ min(first_set(s_A), first_main(s_A))
16               + fintime(count(s_A) + (first_set(s_A) − last_main(s_A))× (a1 / a2)) AND
       (flag(s_A) OR last_main(s_A) ≥ first_set(s_A))
18       ⇒ first_report(s_B) ≤ first_main(s_A) + fintime(count(s_A) × a1) AND
       NOT flag(s_A) AND first_main(s_A) ≤ last_set(s_A)
20       ⇒ last_report(s_B) ≥ last_set(s_A)
                 + fintime(count(s_A) + 2 + (last_set − first_main) × (a2 / a1)) AND
22     NOT reported(s_A) AND (flag(s_A) OR first_main(s_A) > last_set(s_A))
         ⇒ last_report(s_B) ≥ last_main(s_A) + fintime(count(s_A) × a2)
24   ...
   END TwoTaskRace_TwoTaskRaceSpec
```

Figure 6-19: Forward simulation relation for TwoTaskRace.

pearing the HIOA input. The current translation scheme does not allow input variables. Overcoming this restriction would require the development of a real-analysis theory in PVS with definitions and the usual results related to the notions of limits, continuity, differentiability. On the other hand, in practice, most SHIOA specifications can be converted to closed SHIOAs with appropriate assumptions about the input variables. The current implementation of the translator can only handle differential equations with constant right hand sides. The translator does allow us to translate more complex differential equations by manually providing the solutions. In the future, we plan on eliminating this manual step by invoking software tools for solving DAEs.

Our approach to developing strategies for abstraction proofs is geared toward theorem provers that support tactic-style interactive proving. In theorem proving systems that allow a definition of an automaton type, such as Isabelle [Pau93], an approach to developing such strategies that is not based on templates may be possible. Because we cannot expect to develop strategies that will do arbitrary abstraction proofs fully automatically, a major goal for us is to design our strategies to support user-friendly interactive proving. A PVS feature that facilitates making both interaction with the prover and understanding the significance of saved proofs easier is support for comments and formula labels. Thus, a challenge for other theorem proving systems is to find ways to support ease of understanding during and after the proof process equivalent to what we provide using PVS. To become practical for real-world applications, our strategies will require more testing, tuning, optimization, and better support for manipulating real-valued expressions.

A longer term goal is to integrate theorem provers, computer algebra systems (e.g., Maple), mathematical program solvers, under a single interface for SHIOAs. For instance, the safety and stability verification techniques of Chapters 4 and 5, can be checked by computer algebra systems and optimization tools, whereupon these conditions can be used by theorem prover strategies in automatically deducing the properties.

Part II

Probabilistic Hybrid Systems

Chapter 7

Probabilistic State Machines

Structured Hybrid I/O Automata (SHIOAs) of Part I model system uncertainties as nondeterministic choices. Nondeterminism describes uncertainty as a set of *possible* choices, but does not capture the *probability* of individual choices. Incorporating probabilities in the hybrid system framework gives us a richer language to construct models with. In addition to the types of properties we studied in Part I, with probabilistic models we can begin to state and prove quantitative properties, such as probability of hitting unsafe states and expected time of stabilization. Developing semantics and verification techniques for models that combine nondeterminism, probabilities, and continuous evolution poses two key challenges, namely, (i) reconciling the interaction between probabilistic and nondeterministic choices, and (ii) ensuring *measurability* of all reasonable sets of executions. We refer the reader back to Section 1.3.1 for an informal discussion of these two issues. In this chapter we make certain design decisions to solve these problems and present them in the form of the *Probabilistic Timed I/O Automata (PTIOA)* framework. A transition of a PTIOA can be nondeterministic and probabilistic; continuous evolution is non-probabilistic; and concurrently executing automata communicate through shared actions. We develop the trace-based semantics for PTIOAs which involves measure-theoretic constructions on the space of executions of the automata. We introduce a new notion of external behavior and implementation for PTIOAs and show that PTIOAs have simple substitutivity properties. In Chapter 8, we present techniques for verifying approximate implementation relations for a restricted class of PTIOAs.

7.1 An Overview

Automata with probabilistic transitions are useful for modeling systems that interact with environments with stochastic description of uncertainties. Examples include Gaussian channel delays, stochastic noise in sensors, processor failure probabilities, and also randomization in algorithms. Pure nondeterminism, on the other hand, is necessary for constructing (a) implementation-free, abstract models that are underspecified, and (b) models with arbitrary interleaving of concurrently executing processes. For a detailed discussion on the need for nondeterminism in probabilistic automata we refer the reader to Chapter 4 of [Her02] and the introduction of [dA97]. The interplay between probability and nondeterminism makes it challenging to develop semantics for automaton models that have both [Seg95b, MOW04, Che06, CCK+06b]. Introduction of continuous state spaces and distributions adds another layer of complexity to the problem [CSKN05, vBMOW05, DDLP05].

The PTIOA framework presented here addresses some these challenges. A PTIOA can make nondeterministic and probabilistic choices, its continuous evolution is not stochastic, and sets of PTIOAs communicate through shared actions. PTIOAs generalize several existing models, including Timed I/O Automata [KLSV05], Probabilistic I/O Automata [Seg95b, CCK+06b] and its timed extension presented in [Seg95b], and discrete state Markov Decision Processes [dA97, BCG06, KNSW04, HKNP06].

In Section 7.2, we recall some standard notions from measure theory. In Section 7.3 we present the non-probabilistic semantics for PTIOAs which closely parallels the semantics of SHIOAs with one major difference: PTIOAs do not have input/output variables. This implies that a trace of a PTIOA contains information about the occurrence of input/output actions and the duration of time that elapses between them, but not about evolution of variables over that time.

In Section 7.4 we develop the probabilistic semantics for PTIOAs. In order to ensure that all reasonable sets of executions are measurable we impose the following measurability conditions on a PTIOA $\mathcal{A}$ which has measurable space $(Q, \mathcal{F}_Q)$ as its state space: **M1** for any action, the set of states in which the action is enabled a $\mathcal{F}_Q$-measurable, and **M2** for measurable subsets $R \subseteq \mathbb{R}_{\geq 0}, Y \subseteq Q$, the set of states from which there exists a trajectory with length in R and final state in Y, is $\mathcal{F}_Q$-measurable. In order to define a probability distribution over the set of executions of a PTIOA $\mathcal{A}$, the nondeterministic choices in $\mathcal{A}$ have to be resolved first. Nondeterminism comes from two sources: (a) *external nondeterminism*: choice of one automaton which makes the next move from a set of interacting automata, and (b) *internal nondeterminism*: choice of one move (transition or trajectory) from a set of possible moves of an automaton, We proceed by working with a restricted model, namely, *Task-Structured Deterministic PTIOA (task-DPTIOA)*, that has limited internal nondeterminism, and then develop the semantics for general PTIOAs in Section 7.6. For resolving external nondeterminism we rely on the task structure and task schedules, similar to those introduced in [CCK+06b]. The task-structure ensures that a task uniquely determines an task-DPTIOA which gets to make the next move. The sequence of tasks is specified by a task schedule. Combining a task schedule with a task-DPTIOA $\mathcal{A}$ gives rise to a *probabilistic execution*—a probability measure over the set of executions of $\mathcal{A}$.

The construction of probability measure over executions of task-DPTIOAs is similar in spirit to that in the case of Stochastic Transition Systems (STS) of Cattani et al. [CSKN05]. However, a STS does not have notions of time or trajectories and this leads to very different notions of observable behavior or traces (see Section 7.4.3 for details). In particular, this issue surfaces as we define a distribution over traces corresponding to a probabilistic execution.

In Section 7.5, we use a simple, but intuitive notion of *external behavior* for task-DPTIOAs: for a given automaton $\mathcal{A}$, its external behavior is a function that maps each closing environment $\mathcal{E}$ of $\mathcal{A}$ to the set of all possible trace distributions of the composition of $\mathcal{A}$ and $\mathcal{E}$. We show that task-DPTIOAs are substitutive with respect to the implementation relation defined in terms of the above notion of external behavior. Indeed, considering closed automata and using this functional definition of external behavior lets us circumvent some of the difficulties that underlie compositionality in the probabilistic setting. However, viewing external behavior as a mapping from environments as opposed to a set of trace distributions is natural in many applications, including analysis of security protocols [CCK+06c]. Section 7.7 presents an extension of the HIOA language for specifying PTIOAs and examples illustrating the key notions introduced in the chapter.

7.2 Preliminaries

Deterministic Sets of Trajectories. The notations for sets, functions, variables, and trajectories from Section 2.1 of Part I have the same meanings unless stated otherwise. We associate with the time axis $\mathbb{T} = \mathbb{R}_{\geq 0} \cup \{\infty\}$ the Euclidean topology with a point at ∞. Let X be a set of variables. The set of all valuations for the variables in X is denoted by $val(X)$. Given a set of trajectories $\mathcal{T}$ for X, we denote the subset of $\mathcal{T}$ starting from $\mathbf{x} \in val(X)$, by $\mathcal{T}(\mathbf{x})$. A set of trajectories $\mathcal{T}$ for X is *deterministic* if for all $\mathbf{x} \in val(X)$, for any two trajectories $\tau_1, \tau_2 \in \mathcal{T}(\mathbf{x})$ either τ_1 is a prefix of τ_2 or τ_2 is a prefix of τ_1. If $\mathcal{T}$ is deterministic then for any $\mathbf{x} \in val(X)$, $(\mathcal{T}(\mathbf{x}), \leq)$ is a total order.

As $max(\mathbf{x})$ is required to be a closed trajectory,$max(\mathbf{x}).ltime$ and $max(\mathbf{x}).lval$ are well defined.

Measurability and Measures. We follow the standard notations as found in any text book on measure theory, as for instance [Dud89]. A measurable space is denoted by $(S, \mathcal{F}_S)$, where S is a set and $\mathcal{F}_S$ is a σ-algebra over S. $\mathcal{F}_S$ is closed under countable union and complementation and its members are called *measurable sets*. Given a topological space $(S, \mathscr{T})$, there exists a smallest σ-algebra containing $\mathscr{T}$ and it is called the *Borel σ-algebra*. Whenever the sets $\mathbb{R}, \mathbb{R}_{\geq 0}$ and $\mathbb{T}$ are viewed as measurable spaces, it is assumed that they are equipped with their usual Borel σ-algebras.

The *product* of two measurable spaces $(S_1, \mathcal{F}_{S_1})$ and $(S_2, \mathcal{F}_{S_2})$ is defined as the measurable space $(S_1 \times S_2, \mathcal{F}_{S_1} \otimes \mathcal{F}_{S_2})$, where $\mathcal{F}_{S_1} \otimes \mathcal{F}_{S_2}$ is the smallest σ-algebra generated by sets of the form $A \times B = \{(s_1, s_2) \mid s_1 \in A, y \in B\}$, for all $A \in \mathcal{F}_{S_1}, B \in \mathcal{F}_{S_2}$.

A *measure* over $(S, \mathcal{F}_S)$ is a function $\mu : \mathcal{F}_S \to \mathbb{R}_{\geq 0}$, such that $\mu(\emptyset) = 0$ and for every countable collection of disjoint sets $\{S_i\}_{i \in I}$ in $\mathcal{F}_S$, $\mu(\biguplus_{i \in I} S_i) = \sum_{i \in I} \mu(S_i)$. Recall that $\biguplus_{i \in I} S_i$ denotes the union of a collection $\{S_i\}_{i \in I}$ of pairwise disjoint sets. A *probability measure* (resp. *sub-probability measure*) over $(S, \mathcal{F}_S)$ is a measure μ such that $\mu(S) = 1$ $(\mu(S) \leq 1)$. The set of probability measures and the set of sub-probability measures over $(S, \mathcal{F}_S)$ is denoted by $\mathcal{P}(S, \mathcal{F}_S)$ and $\mathcal{SP}(S, \mathcal{F}_S)$. A particular family of probability measures is the *Dirac measures*: given a measurable space $(S, \mathcal{F}_S)$ where singleton sets are measurable and $s \in S$, the Dirac measure at s, denoted by δ_s, is defined as $\delta_s(A) = 1$ if $s \in A$, $\delta_s(A) = 0$ otherwise.

A function $f : (S_1, \mathcal{F}_{S_1}) \to (S_2, \mathcal{F}_{S_2})$ is said to be *measurable* if $f^{-1}(E) \in \mathcal{F}_{S_1}$ for every $E \in \mathcal{F}_{S_2}$. The *indicator function* of a set $A \subseteq S_1$, is defined as $I_A(s) = 1$ if $s \in A$, and 0 otherwise. If A is a measurable set, it is easy to check that I_A is a measurable function. If $f : (S_1, \mathcal{F}_{S_1}) \to (S_2, \mathcal{F}_{S_2})$ is a measurable function, and μ is a measure on S_1, then the *image measure of μ under f* is a measure φ on Y defined as $\varphi(E) = \mu(f^{-1}(E))$, for each $E \in \mathcal{F}_{S_2}$.

Integration. A *simple function* $\hat{f}$ on a measurable space $(S, \mathcal{F}_S)$ is a function whose range consists of only finitely many points in $[0, \infty)$. If $c_1, \ldots, c_n$ are distinct values of a simple function $\hat{f}$, and if we set $C_i = \{s \mid \hat{f}(s) = c_i\}$, then $\hat{f}(s) = \sum_{i=1}^{n} c_i I_{C_i}(s)$, where I_{C_i} is the indicator function of C_i. It is clear that $\hat{f}$ is measurable if and only if $C_i \in \mathcal{F}_S$, for each $i \in \{1, \ldots, n\}$. If $\hat{f} : S \to \mathbb{R}_{\geq 0}$ is a simple measurable function of the form $\hat{f} = \sum_{i=1}^{n} c_i I_{C_i}(s)$, where $c_1, \ldots, c_n$ are distinct values of $\hat{f}$, and if $E \in \mathcal{F}_S$, we define

$$\int_E \hat{f} d\mu = \sum_{i=1}^{n} c_i \mu(C_i \cap E).$$

If $f : S \to \mathbb{R}_{\geq 0}$ is measurable and $E \in \mathcal{F}_S$, the Lebesgue integral of f over E is defined as

$$\int_E f d\mu \triangleq \sup_{0 \leq \hat{f} \leq f} \int_E \hat{f} d\mu,$$

the supremum being taken over all simple measurable functions $\hat{f}$ that are less than f.

Semi-Rings. A collection $\mathscr{C}$ of subsets of S, is a *semi-ring* if (1) the sets S and $\emptyset$ are in $\mathscr{C}$, (2) for any $A, B \in \mathscr{C}$, $A \cap B \in \mathscr{C}$, and (3) for any $A, B \in \mathscr{C}$, there exists a finite collection of disjoint sets $\{C_i\}_{i=1}^n$ in $\mathscr{C}$ such that $A \setminus B = \bigcup_{i=1}^n C_i$. It is well known (see, e.g. [Dud89]) that a measure μ defined over a semi-ring $\mathscr{C}$ can be uniquely extended to a measure over the σ-algebra generated by $\mathscr{C}$ by defining $\mu(\cup_{i=1}^n C_i) = \sum_{i=1}^n \mu(C_i)$. We will use the following theorem to constructing measures of the space of executions of automata:

Theorem 7.1. *A probability measure defined over a semi-ring $\mathscr{C}$ can be uniquely extended to a probability measure over the σ-algebra generated by $\mathscr{C}$.*

In constructing measures over the space of executions of a PTIOA, we have to integrate over the space of probability distributions over sets, namely the state space Q, and therefore, we need to define a σ-algebra over $\mathcal{P}(Q, \mathcal{F}_Q)$. For this, we use the following construction due to Giry [Gir81]: for each $A \in \mathcal{F}_Q$, let the function $p_A : \mathcal{P}(Q, \mathcal{F}_Q) \to [0, 1]$ be defined as $p_A(\mu) = \mu(A)$. The σ-algebra on $\mathcal{P}(Q, \mathcal{F}_Q)$, then is the smallest σ-algebra such that all p_A's are measurable.

7.3 Task-Deterministic Probabilistic Timed I/O Automata

In this section we introduce *Task-Deterministic Probabilistic Timed Input/Output Automata (Task DPTIOAs)* and present definitions and results regarding the non-probabilistic aspects of their behavior.

7.3.1 Definition of Task DPTIOAs

We present the definition of Task DPTIOAs in two parts. First we define *Probabilistic Timed I/O Automata (PTIOAs)* and then we impose several assumptions that restrict the nondeterminism in a PTIOA to obtain the task DPTIOA model.

Definition 7.1. A *Probabilistic Timed I/O Automaton* $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T})$, where:

(a) X is a set of *internal* or *state* variables.

(b) $Q \subseteq val(X)$ is a set of states, $(Q, \mathcal{F}_Q)$ is a measurable space called the *state space*, and $\bar{\mathbf{x}} \in Q$ is the *start state*.

(c) A is a countable set of *actions*, partitioned into *internal* H, *input* I and *output* O actions. $L = O \cup H$ is the set of *local actions* and $E = O \cup I$ is the set of *external actions*.

(d) $\mathcal{D} \subseteq Q \times A \times \mathcal{P}(Q, \mathcal{F}_Q)$ is the set of *probabilistic transitions*. If $(\mathbf{x}, a, \mu)$ is an element of $\mathcal{D}$, we write $\mathbf{x} \xrightarrow{a} \mu$ and action a is said to be *enabled* at $\mathbf{x}$. The set of states in which action a is enabled is denoted by E_a. For $B \subseteq A$, we define E_B to be $\cup_{a \in B} E_a$. The set of actions enabled at $\mathbf{x}$ is denoted by *enabled*$(\mathbf{x})$. If a single action $a \in B \subseteq$

is enabled at $\mathbf{x}$ and $\mathbf{x} \xrightarrow{a} \mu$, then this μ is denoted by $\mu_{\mathbf{x},B}$. If B is a singleton set $\{a\}$ then we drop the set notation and write $\mu_{\mathbf{x},a}$.

(e) $\mathcal{T}$ is a set of trajectories for Q that is (i) deterministic, (ii) closed under prefix, suffix, concatenation, and (iii) contains $\wp(\mathbf{x})$ for every $\mathbf{x} \in Q$. For $R \subseteq \mathbb{R}_{\geq 0}$ and $P \subseteq Q$, we define $E_{R,P}$ to be the set of states $\{\mathbf{x} \in Q \mid \exists \text{ closed } \tau \in \mathcal{T}(x), \tau.\mathit{ltime} \in R \wedge \tau.\mathit{lstate} \in P\}$.

In addition $\mathcal{A}$ must satisfy the following conditions:

E1 *(Input action enabled)* For every input action $a \in I$ and for every state $\mathbf{x} \in Q$, a is enabled at $\mathbf{x}$.

E2 *(Time passage enabled)* For every state $\mathbf{x} \in Q$, there exists $\tau \in \mathcal{T}(\mathbf{x})$ such that either $\tau.\mathit{ltime} = \infty$ or τ is closed and some $a \in L$ is enabled in $\tau.\mathit{lstate}$.

M1 *(Transitions measurability)* For every $a \in L$, E_a is measurable.

M2 *(Trajectory measurability)* For measurable sets $R \subseteq \mathbb{R}_{\geq 0}$, $Y \in \mathcal{F}_Q$, $E_{R,Y}$ is measurable.

P1 *(Progressive)* $\mathcal{A}$ never generates infinitely many locally controlled actions within a finite time interval.

Notations. A PTIOA which does not necessarily satisfy axioms **M1** and **M2** is called a *pre-PTIOA*. A pre-PTIOA is *closed* if its set of input actions is empty. We denote the components of a pre-PTIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Q_{\mathcal{A}}, \bar{\mathbf{x}}_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \mathcal{T}_A$ etc., and the components of a pre-PTIOA $\mathcal{A}_i$ by $X_i, Q_i, \mathbf{x}_i, \mathcal{D}_i$, etc. The special sets of states of pre-PTIOA $\mathcal{A}$ that we defined above, are denoted by $E_{a,\mathcal{A}}, E_{R,Y,\mathcal{A}}$, etc., and those of pre-PTIOA $\mathcal{A}_i$ by $E_{a,i}, E_{R,Y,i}$, etc.

The axioms **E1** and **E2** are identical to the standard non-blocking axiom of SHIOAs. Since PTIOAs do not have input variables, here the input trajectory enabled axiom takes the form of the time-passage-enabled axiom **E2**. Axioms **M1** and **M2** are used later to construct a measurable space over the set of executions of $\mathcal{A}$ and to assign probabilities measures to certain measurable sets of executions. **M1** states that for any local action a, the set of states of $\mathcal{A}$ in which a is enabled is a measurable set, and **M2** states that for measurable subsets $R \subseteq \mathbb{R}_{\geq 0}, Y \subseteq Q$, the set of states from which there exits a closed trajectory with length in R and final state in Y, is a measurable set. **E1** and **M1** together imply that E_B is a measurable set for any set of actions $B \subseteq A$. The progressive axiom **P1** is a natural assumption for timed automata (see, Page 75 of [KLSV05]). The formal meaning of **P1** is given shortly after we define executions of $\mathcal{A}$.

A few remarks comparing this definition with that of a Structured Hybrid I/O Automaton of Part I:

(1) The measurability requirement on Q is quite weak; most state spaces that we typically encounter in applications are measurable.

(2) Unlike SHIOAs, PTIOAs do not have input or output variables, because we assume that PTIOAs communicate through shared actions only and not through shared variables. Extending the results presented here to a model with external variables is a direction of future research.

(3) For clarity of exposition, it is assumed that PTIOAs start from a single start state $\bar{\mathbf{x}}$ and that the set of actions A is countable; correctness of our results does not depend

crucially on these assumptions. Wherever necessary, we assume that every subset of A is measurable.

Definition 7.2. A *Task-Deterministic PTIOA (Task-DPTIOA)* $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T}, \mathcal{R})$, where

(a) $(X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T})$ is a PTIOA, and

(b) $\mathcal{R}$ is an equivalence relation on the set of local actions L; the equivalence classes of $\mathcal{R}$ are called *tasks*. A task T is called an *output task* if $T \subseteq O$.

In addition, $\mathcal{A}$ must satisfy the following axioms:

D1 *(Transition determinism)* For every $\mathbf{x} \in Q$ and $a \in A$, there is at most one $\mu \in \mathcal{P}(Q, \mathcal{F}_Q)$ such that $(\mathbf{x}, a, \mu) \in D$.

D2 *(Action determinism)* For every $\mathbf{x} \in Q$ and $T \in \mathcal{R}$, at most one $a \in T$ is enabled in $\mathbf{x}$.

D3 *(Time-action determinism)* For any state x at most one of the following may exist: (1) a local action a such that $x \in E_a$, (2) a non-point trajectory $\tau \in \mathcal{T}(x)$.

Axioms **D1-3** allow resolution of nondeterminism in a structured manner. Axioms **D1** and **D2** together imply that from any state $\mathbf{x}$, given a task T, there can be at most one action $a \in T$ that is enabled at $\mathbf{x}$. Further, if there exists such an action a, then there exists a unique distribution $\mu \in \mathcal{P}(Q, \mathcal{F}_Q)$, which specifies the probabilistic transition $\mathbf{x} \xrightarrow{a} \mu$, from $\mathbf{x}$. According to **D3**, from any state $\mathbf{x}$, a local action may be enabled, or some non-zero amount of time can elapse, but not both. It prevents an action from remaining enabled while time elapses and is similar to the maximal progress assumption found in real-time process algebras (see, for example [HR95]). If local actions are enabled at $\mathbf{x}$ then time cannot elapse and the automaton nondeterministically chooses one action a from the set of enabled actions. This nondeterministic choice is resolved by a *task schedule*, which we shall define shortly. If a task T is specified then **D2** implies that at $\mathbf{x}$ there can be at most one enabled action in T, and hence, by **D1**, at most one probabilistic transition corresponding to that action. This transition tells us the probability of particular state $\mathbf{x}'$ to be the post-state of $\mathbf{x}$. If, on the other hand, time can elapse from $\mathbf{x}$, then according to **E2** there are two further possibilities, namely, either time advances to infinity (with no local action ever being enabled) or there exists a closed trajectory $\tau \in \mathcal{T}(\mathbf{x})$ such that some local action is enabled at $\tau.lstate$. In the second case, axiom **D3** and determinism of $\mathcal{T}$ imply that, the state evolves according to the maximal deterministic trajectory $\tau \in \mathcal{T}(\mathbf{x})$.

We conclude this section by proving the following theorem relating the PTIOAs with the HIOA model of Chapter 2: Suppose $\mathcal{A}$ is a pre-PTIOA for which all probabilistic transitions are of the form $\mathbf{x} \xrightarrow{a} \delta_{\mathbf{x}'}$ and $\hat{\mathcal{A}}$ is obtained by replacing these probabilistic transitions with corresponding deterministic transition $\mathbf{x} \xrightarrow{a} \mathbf{x}'$. Then, $\hat{A}$ is HIOA.

Theorem 7.2. *Suppose $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T})$ is a pre-PTIOA such that for every transition $x \xrightarrow{a} \mu$, μ is a Dirac measure $\delta_{\mathbf{x}'}$, for some $\mathbf{x}' \in Q$. Then, $\hat{\mathcal{A}} = (X, \emptyset, \emptyset, Q, \{\bar{\mathbf{x}}\}, I, O, H, \hat{\mathcal{D}}, \mathcal{T})$ is a hybrid I/O automaton, where $\hat{\mathcal{D}} \triangleq \{(\mathbf{x}, a, \mathbf{x}') \mid \mathbf{x} \xrightarrow{a}_{\mathcal{A}} \delta_{\mathbf{x}'}\}$.*

Proof. All the components of $\hat{\mathcal{A}}$ satisfy the type requirements in the definition of a HIOA. In particular, $\hat{\mathcal{D}}$ is flattened to be a subset of $Q \times A \times Q$. The set of trajectories of $\hat{\mathcal{A}}$ satisfies **T1-3** from the definition of $\mathcal{T}$. This establishes that $\hat{\mathcal{A}}$ is a pre-HIOA. Since $\hat{\mathcal{A}}$ does not have any input variables, checking that it is input-trajectory-enabled is tantamount to checking that it is time-passage-enabled, which is implied by bf E2 of $\mathcal{A}$. ∎

7.3.2 Executions and Traces

Execution fragments and executions of a PTIOA $\mathcal{A}$ are defined in the same way as in the case of HA (see, Section 2.2.2). We remind the reader that an *execution fragment* of is an alternating sequence of actions and trajectories $\alpha = \tau_0 a_1 \tau_1 a_2 \ldots$, where each $\tau_i \in \mathcal{T}$, $a_i \in A$ and a_i is enabled at $\tau_{i-1}.lstate$. The *first state* of an execution fragment α, $\alpha.fstate$, is $\tau_0.fstate$. An execution fragment α is an *execution* of $\mathcal{A}$ if $\alpha.fstate = \bar{\mathbf{x}}$. The *length* of a finite execution fragment α is the number of trajectories in α. An execution fragment is *closed* if it is a finite sequence and the last trajectory is closed. Proposition 7.3 follows from axiom **D3**.

Proposition 7.3. In any execution fragment of a closed PTIOA all trajectories, except possibly the last trajectory (of a finite fragment) are maximal.

Proof. Suppose, for the sake of contradiction, the proposition is false. That is, there exists a non-final, non-maximal trajectory τ in the execution fragment α. Since τ is not maximal, there exists a non point trajectory τ', such that τ' is enabled at $\tau.lstate$. Since τ is non-final, there exists an action $a_{i+1} \in A$, that is enabled at $\tau.lstate$. This contradicts **D3** at $\tau.lstate$. ∎

The *trace* of an execution α represents its externally visible part. Unlike SHIOAs, a PTIOA does not have external variables. Hence, the externally visible part of any trajectory τ of a PTIOA is simply a mapping from $dom(\tau)$ to the empty set of variables. Formally, the trace of an execution α, denoted by $trace(\alpha)$, is the $(E, \emptyset)$-restriction of α. The only information that is conveyed by the externally visible part of a trajectory τ is $dom(\tau)$, and hence, the information in the trace of an execution fragment is the external actions and duration of the intervening time intervals. For the development of probabilistic semantics for task-DPTIOAs it will be necessary for the *trace* function to be measurable; we shall prove this result in Section 7.4.3.

It is worth comparing this definition of a trace of a PTIOA with that in other probabilistic transition systems that do not have special semantics for modeling time and trajectories. For example, in Stochastic Transition System (STS) of [CSKN05], a trace is obtained by simply removing the internal (invisible) actions and states from an execution. In such models, one obvious approach for modeling time passage is to treat a transition labeled by a real number r as a time passage action of duration r. So, an execution is a sequence for the form $q_0\ r_0\ q_1\ a_1\ q_2\ r_1\ q_3\ a_2\ q_4$, where the $q_i's$ are states, a_1 is an internal action, a_2 is an external actions, and $r_0, r_1 \in \mathbb{R}_{\geq 0}$ are internal actions which model the passage of time. The trace of an STS contains information about the point of occurrence of internal actions over an interval of time. For example, the trace of the above execution is $r_0 r_1 a_2$, which tells us that some internal action occurred between r_0 and r_1. And this makes it relatively straightforward to show that the trace function is measurable. In PTIOAs, the trace of an execution $\tau_0 a_1 \tau_1 a_2 \tau_2$, where a_1 is an internal action and a_2 is an external action,

equals $((\tau_0 \frown \tau_1) \downarrow \emptyset)a_2(\tau_2 \downarrow \emptyset)$. Concatenating τ_0 and τ_1 hides information about the time of occurrence of internal actions a_1. Consequently, this "information loss" makes the proof of measurability of the *trace* function more complicated in PTIOAs, compared to the corresponding proof in the STS setting.

We denote the set of execution fragments, the set of executions, and the set of traces of PTIOA $\mathcal{A}$ by $\mathsf{Frags}_{\mathcal{A}}$, $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Traces}_{\mathcal{A}}$. The set of finite fragments, finite executions and finite traces are denoted by $\mathsf{Frags}^*_{\mathcal{A}}$, $\mathsf{Execs}^*_{\mathcal{A}}$ and $\mathsf{Traces}^*_{\mathcal{A}}$. Having defined traces and executions, we can now state the meaning of axiom **P1** formally. This axiom asserts that over any execution fragment α, if $\alpha.ltime$ is finite then α can have only a finite number of locally controlled actions.

7.3.3 Composition of Task-DPTIOAs

The composition operation enables us to construct a PTIOAs representing a complex system from two interacting PTIOAs by identifying their external actions that have the same names. PTIOAs do not have external (input/output) variables and component automata communicate through external actions only.

Definition 7.3. Pre-PTIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if $X_1 \cap X_2 = H_1 \cap A_2 = H_2 \cap A_1 = O_1 \cap O_2 = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible pre-PTIOAs then their *composition* $\mathcal{A}_1||\mathcal{A}_2$, is defined to be $\mathcal{A} \triangleq (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T})$, where:

(a) $X = X_1 \cup X_2$,

(b) $(Q, \mathcal{F}_Q) = (Q_1 \times Q_2, \mathcal{F}_{Q_1} \otimes \mathcal{F}_{Q_2})$, and $\bar{\mathbf{x}} = (\bar{\mathbf{x}_1}, \bar{\mathbf{x}_2})$,

(c) $A = A_1 \cup A_2, I = (I_1 \cup I_2) \setminus (O_1 \cup O_2), O = O_1 \cup O_2$, and $H = H_1 \cup H_2$,

(d) $\mathcal{D} \subseteq Q \times A \times \mathcal{P}(Q, \mathcal{F}_Q)$ is the set of triples $((\mathbf{x}_1, \mathbf{x}_2), a, \mu_1 \otimes \mu_2)$ such that for $i \in \{1, 2\}$ if $a \in A_i$ then $(\mathbf{x}_i, a, \mu_i) \in \mathcal{D}_i$, otherwise $\mu_i = \delta_{\mathbf{x}_i}$,

(e) $\mathcal{T} = \{\tau \in trajs(X) \mid \tau \downarrow X_i \in \mathcal{T}_i, i \in \{1, 2\}\}$, and

We show that the class of pre-PTIOAs are closed under the composition operator.

Theorem 7.4. *If $\mathcal{A}_1, \mathcal{A}_2$ are compatible pre-PTIOAs then $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ is a pre-PTIOA.*

Proof. Parts (a), (b), (c), (d), and (f) of Definition 7.3 match with the corresponding parts of Definition 7.1. We check that the set $\mathcal{T}$ of trajectories of $\mathcal{A}$ satisfy the necessary properties stated in Part (e) of Definition 7.1. The fact that $\mathcal{T}$ is closed under prefix, suffix, concatenation follow from Definition 7.3. It is also easy to check that $\wp(\mathbf{x}) \in \mathcal{T}$, for every $\mathbf{x} \in Q$. Next, we show that $\mathcal{T}$ is deterministic. Consider two distinct trajectories $\tau, \tau' \in \mathcal{T}(\mathbf{x})$, for some $\mathbf{x} \in Q$ with $\tau \subseteq \tau'$. Since $(\tau \downarrow X_i), (\tau' \downarrow X_i) \in \mathcal{T}_i(\mathbf{x} \lceil X_i)$, for $i \in \{1, 2\}$ and $\mathcal{T}_i$ is a deterministic set of trajectories for X_i, it follows that $(\tau \downarrow X_i) \leq (\tau' \downarrow X_i)$. Combining this result for 1 and 2, we get $\tau \leq \tau'$.

The fact that $\mathcal{A}$ satisfies axioms **E1** and **E2** follow from Theorem 7.2 of [KLSV05], and that it satisfies **P1** follows from Theorem 7.12 of [KLSV05]. ∎

Lemma 7.5. *If $\mathcal{A}_1, \mathcal{A}_2$ are compatible pre-PTIOAs and they individually satisfy* **M1** *then $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ is a pre-PTIOA satisfying* **M1**.

Proof. From theorem 7.4, we know that $\mathcal{A}$ is a pre-PTIOA. We have to show that for any local action $a \in L$, the set of states E_a is $\mathcal{F}_{X_1} \otimes \mathcal{F}_{X_2}$-measurable. Suppose a is a local action of $\mathcal{A}_1$, and let $E_a^1 \subseteq X_1$ be the set of states of $\mathcal{A}_1$ where a is enabled. The set of states of $\mathcal{A}$ where a is enabled is $E_a = E_a^1 \times X_2$. $\mathcal{A}_1$ satisfies **M1**, therefore $E_a^1 \in \mathcal{F}_{X_1}$ and $E_a = E_a^1 \times X_2 \in \mathcal{F}_{X_1} \otimes \mathcal{F}_{X_2}$. The case for $a \in L$, where a is a local action of $\mathcal{A}_2$ is symmetric. ■

Theorem 7.6. *If $\mathcal{A}_1, \mathcal{A}_2$ are compatible PTIOAs and $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$ satisfies* **M2** *then $\mathcal{A}$ is a PTIOA.*

Proof. Follows from Lemma 7.5 and the assumption that $\mathcal{A}$ satisfies **M2**. ■

Definition 7.4. Task-DPTIOAs $\mathcal{A}_1 = (\mathcal{H}_1, \mathcal{R}_1)$ and $\mathcal{A}_2 = (\mathcal{H}_2, \mathcal{R}_2)$ are *compatible* if the underlying PTIOAs $\mathcal{H}_1$ and $\mathcal{H}_2$ are compatible. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible task-DPTIOAs then their *composition* $\mathcal{A}_1 || \mathcal{A}_2$, is defined to be $\mathcal{A} \triangleq (\mathcal{H}, \mathcal{R})$, where: (a) $\mathcal{H} = \mathcal{H}_1 || \mathcal{H}_2$ and (b) $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$.

Theorem 7.7. *If $\mathcal{A}_1, \mathcal{A}_2$ are compatible task-DPTIOAs and $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$ satisfies* **M2** *then $\mathcal{A}$ is a task-DPTIOA.*

Proof. Follows from Theorem 7.6 that $\mathcal{A}$ is a PTIOA. Note that $\mathcal{R}_1 \cup \mathcal{R}_2$ is an equivalence relation because compatibility of $\mathcal{A}_1$ and $\mathcal{A}_2$ ensures disjoint sets of locally controlled actions.

We check that $\mathcal{A}$ satisfies axioms **D1-3**. Transition determinism **D1** follows immediately because the sets of states are disjoint, and $\mathcal{A}_1, \mathcal{A}_2$ satisfy **D1** individually. It is also easy to check that action determinism **D2** is preserved under composition. Finally, we check that $\mathcal{A}$ satisfies time-action determinism **D3**. Suppose some local action $a \in L$ is enabled at a given state x. Let us assume without loss of generality that $a \in L_1$ and $a \notin L_2$. Then a is enabled at $\mathbf{x} \lceil X_1$ and since $\mathcal{A}$ satisfies **D3**, there does not exist any non-point trajectory in $\mathcal{T}_1$ (and therefore in $\mathcal{T}$) that starts from $\mathbf{x} \lceil X_1$. Likewise, if there exists a non-point trajectory starting from $\mathbf{x}$, then no local action is enabled at $x \lceil X_1$ or at $x \lceil X_2$. ■

We conclude this section with a projection lemma for composed PTIOAs analogous to Lemma 2.2 for non-probabilistic hybrid automata.

Theorem 7.8. *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are pre-PTIOAs and $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2$. If α is an execution of $\mathcal{A}_1 || \mathcal{A}_2$, then for $i \in \{1, 2\}$, $\alpha \lceil (X_i, A_i)$ is an execution for $\mathcal{A}_i$.*

Proof. Let α be any execution of $\mathcal{A}$. The proof is by induction on the length of α. The interesting case arises when $\alpha = \alpha' a_1 \tau$, where τ is a trajectory $\mathcal{T}$, and $a_1 \in A_1 \setminus A_2$, and we have to show that $\alpha \lceil (A_2, X_2)$ is an execution of $\mathcal{A}_2$. Since $a_1 \in A_1 \setminus A_2$, $\alpha \lceil (A_2, X_2) = \alpha' \lceil (A_2, X_2) \frown (\tau \downarrow X_2)$. From Definition 7.3 we have $\alpha'.lstate \lceil X_2 = (\tau \downarrow X_2).fstate$, therefore, indeed $(\alpha' \frown \tau) \lceil (A_2, X_2)$ is an execution of $\mathcal{A}_2$. ■

7.4 Probabilistic Semantics for Task-DPTIOAs

In the previous section, we presented the basic definitions and results concerning non-probabilistic behavior of PTIOAs. The set of executions and traces of a PTIOA $\mathcal{A}$ correspond to *all possible behaviors and visible behaviors* of $\mathcal{A}$. As we saw, these non-probabilistic notions are closely related to the corresponding notions for SHIOAs presented in Chapter 2.

The PTIOA framework, of course, is distinct because here in addition to specifying the set of *possible transitions*, the models also specify the *probability of those transitions.* Thus, in the PTIOA framework, we can describe and reason about not just the set of possible behaviors, but also the *set of probability distributions over all (visible) behaviors*. In this sections, we develop the mathematical machinery for such probabilistic reasoning.

In order to construct a probability measure over the set of executions of a given PTIOA $\mathcal{A}$, we have to first define the measurable sets in Execs_A. In the standard approach for probabilistic automata with discrete state spaces [Seg95b, CCK+06b, KNSW04] one defines the σ-algebra as the collection of sets of the form $C_\alpha := \{\alpha' \mid \alpha \text{ is a prefix of } \alpha'\}$. One then defines the probability of each C_α as the product of the probabilities of the transition sequence in α. It is well known (see, e.g., generalization of Markov processes in [Doo53]) that this approach does not work when the transitions give continuous probability distributions, as in the case of PTIOAs. This is because the probability of occurrence of any particular finite sequence of transitions is typically 0. Instead of considering a set of executions that extend a *single* prefix, we consider a set containing executions that extend any prefix from a "cylinder" or *base* of prefixes. A similar technique of constructing the measurable space of executions using cylinders has been previously employed in the setting of STS with general state spaces and actions [CSKN05].

7.4.1 Semi-ring on Executions and Traces

Definition 7.5. A *base* is a finite sequence of the form

$$\Lambda = Q_0\, R_0\, Q'_0\, A_1\, Q_1\, R_1\, Q'_1\, A_2\, Q_2\, R_2 \ldots Q'_{m-1}\, A_m\, Q_m\, R_m\, Q'_m,$$

where for every $i \in \{0, \ldots, m\}$, $Q_i, Q'_i \in \mathcal{F}_Q$, R_i is a measurable set in $\mathbb{R}_{\geq 0}$, and for every $i \in \{1, \ldots, m\}$, $A_i \subseteq A$. The *length* of the above base is defined to be m.

Informally, an execution fragment $\alpha = \tau_0 a_1 \tau_1 \ldots a_m \tau_m$ "matches" the base Λ defined above, if τ_0 starts from a state in Q_0, its length is a number in R_0, it finishes at a state in Q'_0, a_1 is an action in A_1 which takes $\tau_0.lstate$ to some state in Q_1 τ_1 starts at the post-state of the transition in Q_1, its length is a number in R_1, and so on. Given a base Λ, the set of execution fragments that "match" the pattern laid out by Λ is called the basic set or *cone* of Λ and is denoted by C_Λ.

Definition 7.6. The *basic set* corresponding to a base

$$\Lambda = Q_0\, R_0\, Q'_0\, A_1\, Q_1\, R_1\, Q'_1 \ldots Q'_{m-1}\, A_m\, Q_m\, R_m\, Q'_m$$

is a set of execution fragments of $\mathcal{A}$,

$$C_\Lambda \triangleq \{ \quad \tau_0\, a_1\, \tau_1 \ldots \tau_m\, \alpha \in \mathsf{Frags}_{\mathcal{A}} \mid \forall i \in \{0, \ldots, m\}, \tau.fstate \in Q_i, \tau_i.ltime \in R_i,$$
$$\tau.lstate \in Q'_i,\ \forall i \in \{1, \ldots, m\},\quad a_i \in A_i\}.$$

If $\Lambda = Q_0\, R_0\, Q'_0 \ldots Q'_{m-1}\, A_m\, Q_m\, R_m\, Q'_m$, $\Lambda_1 = Q_0\, R_0\, Q'_0 \ldots Q'_{m-1} A_m Q_m$, and $\Lambda_2 = Q_0\, R_0\, Q_1 \ldots Q'_{m-1}$, we will abbreviate Λ as $\Lambda_1\, R_m Q'_m$ or as $\Lambda_2\, A_m\, Q_m\, R_m\, Q'_m$. The next lemma is our starting point toward defining a probability measure over $\mathsf{Execs}_{\mathcal{A}}$. It asserts that the collection of all basic sets of $\mathcal{A}$ forms a semi-ring in $\mathsf{Frags}_{\mathcal{A}}$.

Lemma 7.9. *The collection $\mathscr{C}$ of all basic sets of $\mathcal{A}$ is a semi-ring.*

Proof. We check that $\mathscr{C}$ satisfies the three properties of a semi-ring:

1. $\mathscr{C}$ contains the empty set $\emptyset$ and $\mathsf{Frags}_{\mathcal{A}}$. If $\Lambda = Q_0\, R_0\, Q'_0\, A_1\, Q_1 \ldots Q_m\, R_m\, Q'_m$ with at least one $Q_i = \emptyset$, then $C_\Lambda = \emptyset$. If $\Lambda = Q$, then $C_\Lambda = \mathsf{Frags}_{\mathcal{A}}$.

2. If $C_\Lambda, C_\Gamma \in \mathscr{C}$ then there exists a base Δ, such that $C_\Delta = C_\Lambda \cap C_\Gamma$. Assume without loss of generality that Λ and Γ are of equal length. (If unequal then append an appropriately long sequence of $(A\; Q\; \mathbb{R}_{\geq 0}\; Q)$ to the shorter base. The intersection of the two bases after appending is the same as the original intersection.) Let $\Lambda = Q_0 R_0 Q'_0 A_1 Q_1 R_1 Q'_1 \ldots Q_m R_m Q'_m$ and $\Gamma = P_0 S_0 P'_0 B_1 P_1 S_1 P'_1 \ldots P_m S_m P'_m$. Define Δ to be the sequence $(Q_0 \cap P_0)(R_0 \cap S_0)(Q'_0 \cap P'_0)\ (A_1 \cap B_1)(Q_1 \cap P_1)(R_1 \cap S_1)(Q'_1 \cap P'_1) \ldots (Q_m \cap P_m)(R_m \cap S_m)(Q'_m \cap P'_m)$. Now, any execution fragment α is in C_Δ if and only if it is in C_Λ and C_Γ.

3. If $C_\Lambda, C_\Gamma \in \mathscr{C}$ and $C_\Lambda \subseteq C_\Gamma$, then we show that there exists a finite family of basic sets $\{C_i\}_{i \in I}$ such that $C_\Gamma \setminus C_\Lambda = \cup_{i \in I} C_i$ and for all $i, j \in I, C_i \cap C_j = \emptyset$ if $i \neq j$. As in the previous case, assume without loss of generality that Λ and Γ are of equal length. Let $\Lambda = Q_0 R_0 Q'_0 A_1 Q_1 R_1 Q'_1 \ldots Q_m R_m Q'_m$ and $\Gamma = P_0 S_0 P'_0 B_1 P_1 S_1 P'_1 \ldots P_m S_m P'_m$. The index sets for the sets of states and actions in the above bases are $\{0, \ldots, m\}$ and $\{1, \ldots, m\}$, respectively. We use subsets of these index sets to construct the disjoint collection of basic sets whose union equals $C_\Gamma \setminus C_\Lambda$. Since there are countably many possible such subsets, we get the required countable collection of disjoint basic sets.

   Let $I, J, K \subseteq \{0, \ldots, m\}$ and $L \subseteq \{1, \ldots, m\}$ be nonempty sets of indices. Define a collection of bases parameterized by three indices

   $$\Lambda_{I,J,K,L} = Z_0\, T_0\, Z'_0\, D_1\, Z_1\, T_1\, Z'_1\, D_1\, Z_2 \ldots Z_m\, T_m\, Z_m$$

   as follows:

   $$Z_i = \begin{cases} P_i \setminus Q_i & \text{if } i \in I \\ Q_i & \text{otherwise,} \end{cases} \qquad Z'_j = \begin{cases} P'_j \setminus Q'_j & \text{if } j \in J \\ Q'_j & \text{otherwise,} \end{cases}$$

   $$D_k = \begin{cases} B_k \setminus A_k & \text{if } k \in K \\ A_k & \text{otherwise,} \end{cases} \qquad T_l = \begin{cases} S_l \setminus R_l & \text{if } l \in L \\ R_l & \text{otherwise.} \end{cases}$$

   First we show that different values of I, J, K and L give disjoint basic sets and then we check that the union of all such basic sets equal $C_\Gamma \setminus C_\Lambda$. Consider two bases $\Lambda_{I,J,K,L}$ and $\Lambda_{I',J',K',L'}$. Of the three pairs of index sets, at least one must be a pair of different sets. Say, $J \subset J'$, that is, there exists an index $j \notin J$ such that $j \in J'$. Then, the jth set Z'_j in $\Lambda_{I,J,K,L}$ equals $P'_j \setminus Q'_j$ and the set Z'_j in $\Lambda_{I',J',K'}$ equals Q'_j, and hence are disjoint. It follows that the $C_{\Lambda_{I,J,K,L}} \cap C_{\Lambda_{I',J',K',L'}} = \emptyset$. Thus bases defined above are pairwise disjoint.

   It remains to check that any $\alpha \in C_\Lambda \setminus C_\Gamma$ is in one of the basic sets constructed above. Let $\alpha = \tau_0\, a_1\, \tau_1 \ldots \tau_m\, \beta$. Then, $\tau_0.fstate \in P_0 \setminus Q_0$, $\tau_0.ltime \in S_0 \setminus R_0$, $\tau_0.lstate \in P'_0 \setminus Q'_0$, $a_1 \in B_1 \setminus A_1$, $\ldots$, $\tau_m.lstate \in P'_m \setminus Q'_m$. Then there must exist index sets $I_\alpha, J_\alpha, K_\alpha \subseteq \{0, \ldots, m\}$, and $L_\alpha \subseteq \{1, \ldots, m\}$ such that $\tau_i.fstate \in P_i \setminus Q_i$ if $i \in I_\alpha$, $\tau_j.lstate \in P'_j \setminus Q'_i$ for all $j \in J_\alpha$, $a_l \in B_l \setminus A_l$ for all $l \in L_\alpha$, and $\tau_k.ltime \in S_k \setminus R_k$ for all $k \in K_\alpha$. It follows that $\alpha \in C_{\Lambda_{I_\alpha, J_\alpha, K_\alpha, L_\alpha}}$.

■

The σ-algebra generated by $\mathscr{C}$ is denoted by $\mathcal{F}_{\mathsf{Frags}_\mathcal{A}}$. The collection of sets obtained by taking the intersection of each element in $\mathscr{C}$ with $\mathsf{Execs}_\mathcal{A}$ is a semi-ring in $\mathsf{Execs}_\mathcal{A}$. We denote the σ-algebra generated by this semi-ring by $\mathcal{F}_{\mathsf{Execs}_\mathcal{A}}$. We define the measurable space of executions of $\mathcal{A}$ to be $(\mathsf{Execs}_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}_\mathcal{A}})$.

By restricting basic sets to finite execution fragments, we define *finite basic sets*. The collection of all finite basic sets forms a semi-ring over set $\mathsf{Frags}^*_\mathcal{A}$ of finite execution fragments of $\mathcal{A}$. The σ-algebra on $\mathsf{Frags}^*_\mathcal{A}$, and the measurable space $(\mathsf{Execs}^*_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}^*_\mathcal{A}})$ of finite executions are defined in a manner identical to the above constructions.

Definition 7.7. A *trace base* is a finite sequence of the form $\Lambda = R_0\, E_1\, R_1\, E_2 \ldots r_{n-1}\, E_n$ where $\forall i \in \{0, \ldots, n-1\}$, R_i is a measurable set in $\mathbb{R}_{\geq 0}$ and and $\forall j \in \{1, \ldots, n\}$, $E_j \subseteq E$. The *length* of such a trace base is defined to be n The *trace basic set* corresponding to the base Λ is a set of traces of $\mathcal{A}$ defined as:

$$C_\Lambda = \{\tau_0\, a_1\, \tau_1 \ldots a_n\, \beta \in \mathsf{Traces}_\mathcal{A} \mid \forall i \in \{0, \ldots, n\}, \tau_i.ltime \in R_i, \forall i \in \{1, \ldots, n\}, a_i \in E_i\}.$$

The collection $\mathscr{D}$ of all trace basic sets of $\mathcal{A}$ is a semi-ring. The σ-algebra $\mathcal{F}_{\mathsf{Traces}}$ on the set of traces of $\mathcal{A}$ is defined as the σ-algebra generated by the collection of trace basic sets; the measurable space of traces is denoted by $(\mathsf{Traces}_\mathcal{A}, \mathcal{F}_{\mathsf{Traces}_\mathcal{A}})$.

7.4.2 Probability Measure Over Executions

In this section, we proceed to define the value of μ at each basic set of $\mathcal{A}$. Then, using Theorem 7.1 we obtain a probability measure over $\mathsf{Execs}_\mathcal{A}$. The evolution of a PTIOA $\mathcal{A}$ involves both nondeterministic and probabilistic choices. From a given state $\mathbf{x}$ multiple actions may be enabled, and one of these enabled actions has to be chosen nondeterministically. The nondeterministic choice of a task, say T, uniquely determines an enabled action $a \in T$ (axiom **D2**), which in turn uniquely determines (axiom **D1**) the probabilistic transition $\mathbf{x} \xrightarrow{a} \mu$, corresponding to action a that occurs at $\mathbf{x}$. This transition then probabilistically chooses the next state $\mathbf{x}'$ according to the distribution μ.

In order to obtain a probability distribution over the set of executions of $\mathcal{A}$, one has to resolve the nondeterministic choices. This is done through an entity called a*scheduler*. Depending on memory usage and the choice mechanism employed by the scheduler, the following different kinds of schedulers are possible. Suppose α is an execution fragment of $\mathcal{A}$, and the scheduler has to choose one task T from the set of enabled tasks, at $\alpha.lstate$.

Memory usage. The scheduler may be (i) *oblivious*, that is, completely independent of the history of the execution α, (ii) *Markovian*—dependent only on the current state $\alpha.lstate$, or (iii) *history dependent*, in which case the scheduler may use all the information available in α.

Choice mechanism. The task T may be chosen as a deterministic function of the history or as a deterministic function of the current state. Alternatively, the function may give rise to a probability distribution over enabled tasks at $\alpha.lstate$, according to which the actual task T is chosen.

In this thesis, we use *oblivious task scheduler* [CCK+06c, CCK+06b]. An oblivious scheduler chooses the next action deterministically and independently of the information produced during an execution.

Definition 7.8. A *task schedule* for a closed task-DPTIOA $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T}, \mathcal{R})$ is a finite sequence $\rho = T_1 T_2 \ldots T_n$ of tasks in $\mathcal{R}$.

In this thesis, we restrict our attention to finite task schedules because we study probability distributions supported on the set of finite executions of the given PTIOA, and therefore we consider only finite task schedules.

A task schedule resolves nondeterministic choices by repeatedly scheduling tasks. Each task determines at most one transition for the PTIOA. A task schedule for $\mathcal{A}$ determines a probability measure over $(\mathsf{Execs}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}})$. We define an operation that "applies" a task schedule to a PTIOA . Given any task schedule ρ the corresponding probability distribution $\mathsf{apply}(\delta_{\bar{\mathbf{x}}}, \rho)$ over $\mathsf{Exec}_{\mathcal{A}}$, is called a *probabilistic execution* of $\mathcal{A}$.

For each base Λ and each $B \subseteq A$, we define an identity function $g_{\Lambda,B} : \mathsf{Execs}^*_{\mathcal{A}} \rightarrow \{0, 1\}$. Informally, for a closed execution α, $g_{\Lambda,B}(\alpha)$ returns 1 if and only if α matches the pattern of actions and trajectories defined by the base Λ and some action in B is enabled at its last state.

$$g_{\Lambda,B} = \begin{cases} 1 & \text{if } \alpha.lstate \in E_B \text{ and } \alpha \in C_\Lambda \\ 0 & \text{otherwise.} \end{cases} \quad (7.1)$$

Notice that $g_{\Lambda,B}$ is a $\mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}}$-measurable function. There are three cases to consider: if $\Lambda = P$, for some $P \in \mathcal{F}_Q$, then $g^{-1}_{\Lambda,B}(1) = C_{P \cap E_B}$. Secondly, if $\Lambda = \Lambda'\ R\ P$, for some $P \in \mathcal{F}_Q$, $R \in \mathcal{F}_{\mathbb{R}_{\geq 0}}$, then $g^{-1}_{\Lambda,B}(1) = C_{\Lambda'\ R\ (P \cap E_B)}$. Finally, if $\Lambda = \Lambda'\ B'\ P$, for some $B' \subseteq A$, $P \in \mathcal{F}_Q$, then $g^{-1}_{\Lambda,B}(1) = C_{\Lambda'\ B'\ (P \cap E_B)}$. Measurability of $g_{\Lambda,B}$ is ensures the integrability of a function which appears in the following recursive definition of $\mathsf{apply}(,)$.

Definition 7.9. Let $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T}, \mathcal{R})$ be a PTIOA. Given a task schedule ρ for $\mathcal{A}$ and a probability measure $\mu \in \mathcal{P}(\mathsf{Execs}^*_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}^*_{\mathcal{A}}})$, $\mu' = \mathsf{apply}(\mu, \rho)$ is a probability measure in $\mathcal{P}(\mathsf{Execs}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}})$, defined recursively as follows:

1. $\mathsf{apply}(\mu, \lambda) \triangleq \mu$, where λ denotes the empty sequence of tasks.

2. $\mathsf{apply}(\mu, T) \triangleq \mu'$, where T is a task in $\mathcal{R}$ and μ' is defined as follows:

$$\mu'(C_P) = \begin{cases} 1 & \text{if } \bar{\mathbf{x}} \in P \\ 0 & \text{otherwise,} \end{cases} \quad (7.2)$$

$$\mu'(C_{\Lambda B P}) = \mu(C_{\Lambda B (P \cap E^c_T)}) + \int_{\alpha \in \Lambda} g_{\Lambda, B \cap T}(\alpha) \mu_{\alpha.lstate, B \cap T}(P) \mu(d\alpha), \quad (7.3)$$

$$\mu'(C_{\Lambda R P}) = \int_{\alpha \in \Lambda} I_{E_{R,P}}(\alpha.lstate) \mu(d\alpha), \quad (7.4)$$

where $P \in \mathcal{F}_Q$, $B \subseteq A$, and R is a Borel set of $\mathbb{R}_{\geq 0}$. Recall that $\mu_{\mathbf{x},B'}$ denotes a probability distribution $\eta \in \mathcal{P}(Q, \mathcal{F}_Q)$ where $\mathbf{x} \xrightarrow{a} \eta$, and $a \in B'$ is the only action enabled at $\mathbf{x}$.

3. $\mathsf{apply}(\mu, \rho) \triangleq \mathsf{apply}(\mathsf{apply}(\mu, \rho'), T)$, where $\rho = \rho' T$.

The first part of the recursive definition states that the application of an empty task schedule λ does not alter the original distribution of $\mathsf{Execs}_{\mathcal{A}}$. The third part of the definition

states that the measure resulting from applying a task schedule $\rho = \rho' T$, is recursively obtained by applying T to $\mathsf{apply}(\mu, \rho')$.

The second part of the definition describes the measure μ' that results from applying a single task T to μ. Equations (7.2) states the base case of this inductive definition; it defines the value of μ' for basic sets of the form C_P, $P \in \mathcal{F}_Q$. For any task schedule, any basic set of the form C_P has measure 1 if the initial state $\bar{\mathbf{x}}$ is in P, otherwise it has measure 0.

Equations (7.3) and (7.4) inductively defines the value of μ' for basic sets of the form $C_{\Lambda BP}$ and $C_{\Lambda BP}$, by integrating over all executions in C_Λ. Let us consider Equation (7.4) first. The probability measure of an arbitrary basic set $C_{\Lambda RP}$ is simply the probability of the the set of executions in Λ from which there exist a closed trajectory τ such that $\tau.ltime \in R$ and $\tau.lstate \in P$. In other words, it is the probability measure of the executions in Λ whose last states are in $E_{R,P}$. Thus, $\mu'(C_{\Lambda RP})$ defined by integrating over Λ, the indicator function for $E_{R,P}$, with respect to the measure μ.

Next, we consider Equation (7.3). The probability measure of an arbitrary basic set $C_{\Lambda BP}$ can be written as the sum of $\mu(C_{\Lambda B(P\cap E_T^c)})$ and the "sum of probabilities" of all executions $\alpha \in \Lambda$ from the last state of which an action $a \in B \cap T$ and which leads to a state in P. This sum becomes an integral over Λ with respect to the measure μ. The probability distribution $\mu_{\alpha.lstate, B\cap T} \in \mathcal{P}(Q, \mathcal{F}_Q)$ denotes the distribution η of some probabilistic transition $\alpha.lstate \xrightarrow{a} \eta$, for *some* action $a \in B \cap T$. But, from axioms **D2** we know that there can be at most one action $a \in T$ that is enabled at $\alpha.lstate$. In fact, if $g_{\Lambda, B\cup T}(\alpha) \neq 0$ then there exists a unique action a. Moreover, from **D1** we know that there exists a unique transition $\alpha.lstate \xrightarrow{a} \eta$. Hence, $\mu_{\alpha.lstate, B\cap T}()$ is well defined in the integrand. Measurability of $\mu_{\alpha.lstate, B\cap T}$ follows from the Giry construction described in 7.2, and $\mathcal{F}_Q$-measurability of $E_{R,P}$ (axiom **M2**). We have already proved the measurability of $g_{\Lambda, B\cap T}$. The integrand is measurable as it is the product of two measurable functions, and therefore the integral is well defined.

Theorem 7.10. *Let μ be a probability measure on $(\mathsf{Execs}^*_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}^*_{\mathcal{A}}})$ and ρ be a task schedule for $\mathcal{A}$. Then $\mu' = \mathsf{apply}(\mu, \rho)$ is a probability measure on $(\mathsf{Execs}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}})$.*

Proof. For any base Λ, $\mu'(C_\Lambda) \geq 0$, $\mu'(C_{\mathsf{Execs}_{\mathcal{A}}}) = \mu'(C_{\bar{\mathbf{x}}}) = 1$, and if $C_\Lambda = \emptyset$ then $\mu'(C_\Lambda) = 0$. Next, consider a countable disjoint collection of bases $\{\Lambda_i\}_{i\in I}$. For any $i, j \in I$, at least one of the sets in the sequence Λ_i must be disjoint with the corresponding set in Λ_j. (If Λ_i and Λ_j are of different lengths, say Λ_i is shorter than Λ_j, then there must exist a disjoint set in the prefix of the Λ_i that equals in length to Λ_j.) Therefore, $\mu'(\biguplus_{i\in I} C_{\Lambda_i}) = \sum_{i\in I} \mu'(C_{\Lambda_i})$. Thus μ' is a probability measure over the semi-ring $\mathscr{C}$ defined by the collection of all basic sets. From Theorem 7.1 it follows that μ' is a probability measure on $(\mathsf{Execs}^*_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}^*_{\mathcal{A}}})$. ■

In summary, each task schedule for $\mathcal{A}$ gives rise to a probabilistic execution, which is a probability measure on the space $(\mathsf{Execs}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}})$. A set of task schedules give a set of probabilistic executions. Drawing the analogy between a non-probabilistic SHIOA and a task-DPTIOA, the set of all behaviors of the former is its set of executions while for the latter it is the set of all probabilistic executions.

7.4.3 Probability Measure Over Traces

For non-probabilistic automaton models like SHIOAs we are often interested in analyzing their observable behavior or traces as opposed to their actual behavior (executions). In

the PTIOA framework, the role of observable behavior is played by *trace distributions* or probability measures over the set of traces. In this section, we show that the *trace* function is a measurable function from $(\mathsf{Execs}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}})$ to $(\mathsf{Traces}_{\mathcal{A}}, \mathcal{F}_{\mathsf{Traces}_{\mathcal{A}}})$, which enables us to derive trace distributions from probabilistic executions.

For any probabilistic execution μ of a PTIOA, we want there to be a unique corresponding measure on the space of traces. Formally, the image measure of μ with respect to the *trace* function should be well defined. Therefore we require the function $trace : (\mathsf{Execs}, \mathcal{F}_{\mathsf{Execs}}) \to (\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}})$ to be measurable. The following example illustrates the difficulty in proving the measurability of *trace*.

Example 7.1. Consider a trace base $[0, r]\{\mathsf{a}\}$ of a task-DPTIOA $\mathcal{A}$, where r is a positive real and a is an external action. In order to prove measurability of the trace function, we will have to express the set of executions $trace^{-1}(C_{[0,r]\{\mathsf{a}\}})$ as a countable union of basic sets of $\mathcal{A}$. What is the set of executions for which the traces are in $C_{[0,r]\{\mathsf{a}\}}$? For any $n \in \mathbb{N}$, any execution α of the form $\tau_0\, h_1\, \tau_1\, h_2 \ldots \tau_n a$, has $trace(\alpha) \in C_{[0,r]\{\mathsf{a}\}}$, as long as h_i's are internal actions and the sum of the length of the trajectories is at most r. For the *trace* function to be measurable, the above set of executions must be in $\mathcal{F}_{\mathsf{Execs}}$, that is, $\mathcal{E}$ should be expressible as a countable union of basic sets. Showing this requires some work because the condition on the sum of the τ_i's makes them interdependent.

In what follows, we prove a sequence of lemmas that go into proving measurability of the *trace* function for PTIOAs.

Definition 7.10. A trace base Γ of the form $[0, b_0)E_1[0, b_1)E_2 \ldots E_n$, where each $b_i \in \mathbb{R}_{\geq 0}$ and each $E_i \subseteq E$, is said to be a *canonical trace base*.

Lemma 7.11 states that for proving measurability of *trace*, it suffices to show that for any canonical trace base Γ, $trace^{-1}(C_\Gamma)$ is in the σ-algebra of executions.

Lemma 7.11. *Consider a function* $f : (\mathsf{Execs}, \mathcal{F}_{\mathsf{Execs}}) \to (\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}})$*. If* $f^{-1}(C_\Gamma) \in \mathcal{F}_{\mathsf{Execs}}$ *for every canonical trace base* Γ *then* f *is measurable.*

Proof. We define a collection of sets in Traces

$$\mathscr{C} = \{C \subseteq \mathsf{Traces} \mid f^{-1}(C) \in \mathcal{F}_{\mathsf{Execs}}\},$$

and check that $\mathscr{C}$ is in fact a σ-algebra on Traces.

1. $f^{-1}(\mathsf{Traces}) = \mathsf{Execs} \in \mathcal{F}_{\mathsf{Execs}}$, therefore $\mathsf{Traces} \in \mathscr{C}$.

2. For any $C \in \mathscr{C}$, $f^{-1}(\mathsf{Traces} \setminus C) = \mathsf{Execs} \setminus f^{-1}(C) \in \mathcal{F}_{\mathsf{Execs}}$.

3. For any $C_1, C_2 \ldots \in \mathscr{C}$, $f^{-1}(C_1 \cup C_2 \ldots) = f^{-1}(C_1) \cup f^{-1}(C_2) \ldots \in \mathcal{F}_{\mathsf{Execs}}$.

Next we show that if $f^{-1}(C_{\bar{\Gamma}}) \in \mathcal{F}_{\mathsf{Execs}}$ for every canonical trace base $\bar{\Gamma}$, then for every trace base Γ, $f^{-1}(C_\Gamma) \in \mathcal{F}_{\mathsf{Execs}}$. We prove this by induction on the length of trace basic sets. First we assume that for each $b_1 \in \mathbb{R}_{\geq 0}$, $E_1 \subseteq E$, the canonical trace base $\bar{\Gamma}_1 = [0, b_1)E_1$ of length one, $C_{\bar{\Gamma}_1} \in \mathscr{C}$. And we show that for every trace base Γ_1 of unit length (not necessarily canonical), $C_{\Gamma_1} \in \mathscr{C}$. If Γ_1 is of the form $[b, \infty)E_2$, then $C_{[b,\infty)E_2} = C_{[0,\infty)E_2} \cap C^c_{[0,b)E}$, and since $\mathscr{C}$ is a σ-algebra, $C_{[b,\infty)E_2} \in \mathscr{C}$. To see that a trace base of the form $C_{[0,b]E_2}$, $E_2 \subseteq E$, is also in $\mathscr{C}$. Choose a sequence of real numbers $\{b_n\}_{n=1}^{\infty}$ such that

$b_{n+1} > b_n$ and $b_n \to b$ as $n \to \infty$. Since for each n, $C_{[0,b_n)E_1} \in \mathscr{C}$ and $\mathscr{C}$ is a σ-algebra we have:

$$C_{[0,b]E_1} = \bigcap_{n=1}^{\infty} C_{[0,b_n)E_1} \in \mathscr{C}.$$

The same holds true for basic sets of the form $C_{(a,b)E_2} = C_{[0,b)E_2} \cap C_{(a,\infty]E_2}$. Since every measurable set in $\mathbb{R}_{\geq 0}$ is a countable union of segments of the types $[0,b), (b,\infty)$, and (a,b), we have proved that for any trace base Γ_1 of unit length, $C_{\Gamma_1} \in \mathscr{C}$ which implies that $f^{-1}(C_{\Gamma_1}) \in \mathcal{F}_{\mathsf{Execs}}$.

In the induction step we assume that for any $n \in \mathbb{N}$, if all canonical trace basic sets of length n are in $\mathscr{C}$, then all trace basic sets of length n are in $\mathscr{C}$. Based on this inductive hypothesis and the additional assumption that every canonical trace base of length $n+1$, $\bar{\Gamma} = \bar{\Gamma}_n[0, b_{n+1})E_{n+1} \in \mathscr{C}$, where Γ_n is a canonical base of length n, $b_{n+1} \in \mathbb{R}_{\geq 0}$, and $E_{n+1} \subseteq E$, we show that any trace basic set of length $n+1$ is in $\mathscr{C}$. The details of this argument are similar to those in the base case.

Every set in $\mathcal{F}_{\mathsf{Traces}}$ can be expressed as a countable union of the trace basic sets, and by the hypothesis of the lemma, $f^{-1}(C_{\bar{\Gamma}})$ is measurable for every canonical trace base $\bar{\Gamma}$, and therefore f is measurable. ■

Our next step is to show that $trace^{-1}(C_{\bar{\Gamma}})$ is a measurable set for every canonical trace base $\bar{\Gamma}$. In the light of lemma 7.11, this will be sufficient to conclude the measurability of the *trace* function. The proof of the above statement follows from repeated application of the construction described by Definition 7.11. For clear exposition, we describe the construction for canonical trace bases of length one; the same construction is repeated for canonical bases of any finite length. The key idea is to express $trace^{-1}(C_{[0,r)E_1})$ as the countable union of a disjoint basic sets that partition $[0,r)$ into several intervals with rational endpoints. A base is then constructed by inserting internal actions in between the successive sub-intervals.

Definition 7.11. Let $\bar{\Gamma}_1 = [0,r)E_1$ be a canonical trace base of unit length. For $n \in \mathbb{N}$, an *n-partition base* of $\bar{\Gamma}_1$ is a trace base of length n of the form:

$$\Lambda^{n,r}_{q_1,q'_1,\ldots,q'_n} \triangleq Q[q_1,q'_1]Q\ H\ Q[q_2,q'_2]Q\ H\ Q \ldots [q_n,q'_n]Q\ E_1 Q$$

where $q_1, q'_1, q_2, q'_2, \ldots, q_n, q'_n \in \mathbb{Q}_{\geq 0}$, $q_1 < q'_1$, $q_2 < q'_2$, $\ldots$, $q_n < q'_n$, and $q'_1 + q'_2 + \ldots + q'_n < r$. We denote the union over all possible n-partition basic sets of $\bar{\Gamma}_1$ by $\mathcal{Q}_{n,r}$, that is,

$$\mathcal{Q}_{n,r} \triangleq \bigcup_{q_1,q'_1,\ldots,q'_n \in \mathbb{Q}_{\geq 0}} C_{\Lambda^{n,r}_{q_1,q'_1,\ldots,q'_n}},$$

where the union is over nonnegative rationals $q_1, q'_1, \ldots, q'_n$ satisfying the above constraints.

In order to understand the above definitions and the proof of the following Lemma, let us consider an execution α of task-PDTIOA $\mathcal{A}$, such that $trace(\alpha) \in C_{\bar{\Gamma}_1}$. We know that α is of the form $\alpha_1\ a\ \alpha_2$, where $a \in E_1 \subseteq E$, α_2 is any execution fragment of $\mathcal{A}$, and within α_1 some number (possibly 0) of internal actions occur and $\alpha_1.ltime \in [0,r)$. From axiom **P1** we know that the number of internal actions that occur in α_1 is finite. Let us assume that this number is one. Then, α_1 is of the form $\tau_0 h_1 \tau_1$, where $h_1 \in H$ and $\tau_0.ltime + \tau_1.ltime < r$. If we choose $q_1, q'_1, q_2, q'_2 \in \mathbb{Q}_{\geq 0}$, such that $\tau_0.ltime \in [q_1, q'_1]$,

$\tau_1.ltime \in [q_2, q_2']$, and $q_1' + q_2' < r$, then, α is in the basic set of $\Lambda^{2,r}_{q_1,q_1',q_2,q_2'}$. Such choices are possible, as long as $q_1' + q_2' < r$, because $\mathbb{Q}_{\geq 0}$ is dense in $\mathbb{R}_{\geq 0}$,

Informally, each point in the set $\mathbb{S}_2 = \{(x_0, x_1) \in \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \mid x_0 + x_1 < r\}$ corresponds to a possible execution fragment of the form $\tau_0\, h_1\, \tau_1$, with $x_0 = \tau_0.ltime$ and $x_1 = \tau_1.ltime$. And all such executions are included in $\mathcal{Q}_{2,r}$. To see this, notice that (a) each basic set in $\mathcal{Q}_{2,r}$, such as the $C_{\Lambda^{2,r}_{q_1,q_1',q_2,q_2'}}$ we chose above, covers a rectangle $[q_1, q_1'] \times [q_2, q_2'] \subseteq \mathbb{S}_2$, and (b) the union of all such rectangles, defined by all possible choices of $q_1, q_1', q_2, q_2' \in \mathbb{Q}_{\geq 0}$ satisfying $q_1' + q_2 < r$, covers all of $\mathbb{S}_2$.

For the general case, say, $\alpha_1 = \tau_0\ h_1\ \tau_1 \ldots h_n\ \tau_n$, with $n \in \mathbb{N}$. Then the set $\mathbb{S}_n = \{(x_0, \ldots, x_{n-1}) \in \mathbb{R}_{\geq 0}{}^n \mid \sum_{i=0}^{n-1} x_i < r\}$ corresponds to all possible execution fragments with n internal actions with with $x_i = \tau_i.ltime, i \in \{0, \ldots, n\}$. The union over all n-partitions for $\bar{\Gamma}$, namely $\mathcal{Q}_{n,r}$ covers all such possible executions.

Lemma 7.12. *Suppose $\bar{\Gamma}_1 = [0, r)E_1$ is a canonical trace base of a PTIOA $\mathcal{A}$. Then,*

$$trace^{-1}(C_{\bar{\Gamma}_1}) = \bigcup_{n \in \mathbb{N}} \mathcal{Q}_{n,r}.$$

Proof. First we show that $\bigcup_{n \in \mathbb{N}} \mathcal{Q}_{n,r} \subseteq trace^{-1}(C_{\bar{\Gamma}_1})$. Let α be an execution in $\mathcal{Q}_{n,r}$, for some $n \in \mathbb{N}$. Since the first n actions of α are internal actions, $trace(\alpha)$ is of the form $\tau_0 a \alpha'$, where α' is a trace of $\mathcal{A}$, $a \in E_1$ and $\tau.ltime \leq r$. By definition of trace basic set it follows that $trace(\alpha) \in C_{[0,1)E_1}$.

Next, we show that $trace^{-1}(C_{\bar{\Gamma}_1}) \subseteq \bigcup_{n \in \mathbb{N}} \mathcal{Q}_{n,r}$. Consider any execution α of $\mathcal{A}$, such that $trace(\alpha) \in C_{[0,r)E_1}$. Then, α must be of the form $\alpha_1\, a\, \alpha_2$, where $a \in E_1$ and $\alpha_1.ltime < r$. It suffices to show that $\alpha_1 \in \mathcal{Q}_{n,r}$, for some $n \in \mathbb{N}$.

By axiom **P1**, the number of internal actions in α' is finite. Let us assume that the number of internal actions in α' is k, for some $k \in \mathbb{N}$. We write α_1 as $\tau_0\ h_1\ \tau_1\ h_2 \ldots \tau_k$, where each $h_i \in H$, and $\sum_{i=0}^{k} \tau_i.ltime < r$. Since the $\mathbb{Q}_{\geq 0}$ is dense in $\mathbb{R}_{\geq 0}$, there exist $q_0, q_0', q_1, q_1', \ldots, q_k, q_k' \in \mathbb{Q}_{\geq 0}$, such that, for all $i \in \{0, \ldots, k\}$, $\tau_i.ltime \in [q_i, q_i']$ and $\sum_{i=0}^{k} q_i' < r$. By definition 7.11, it follows that α_1 is contained in the basic set of the k-partition base $\Lambda = \Lambda^{k,r}_{q_1,q_1',\ldots,q_k,q_k'}$, that is, $\alpha \in \mathcal{Q}_{k,r}$. ∎

Next we prove the generalization of Lemma 7.12 for finite length canonical trace bases.

Lemma 7.13. *If $\bar{\Gamma}$ is a canonical trace base of $\mathcal{A}$, then $trace^{-1}(C_{\bar{\Gamma}}) \in \mathcal{F}_{\mathsf{Execs}}$.*

Proof. The proof follows from a straightforward induction on the length of the canonical trace base $\bar{\Gamma}$. Suppose $\bar{\Gamma} = [0, r_1)\, E_1\, [0, r_2)\, E_2 \ldots [0, r_n)\, E_n$, where r_i's are nonnegative reals and E_i's are sets of external actions, for some $n \in \mathbb{N}$. For a canonical trace base of length one, by Lemma 7.12 $trace^{-1}(C_{\bar{\Gamma}_1})$ is expressed as a countable union of bases, and therefore is a measurable set.

Suppose the lemma holds for all canonical trace basic sets of length n, for some $n \in \mathbb{N}$. Consider a canonical trace base $\bar{\Gamma} = \bar{\Gamma}_n[0, r)\, E_1$ of length $n+1$, where $\bar{\Gamma}_{n+1}$ is a canonical trace base of length n, $r \in \mathbb{R}_{\geq 0}$, and $E_1 \subseteq E$. From the induction hypothesis we know that $trace^{-1}(C_{\bar{\Gamma}_n}) = \cup_{i \in \mathcal{I}} C_{\Delta_i}$, for some countable collection of basic sets indexed by $\mathcal{I}$. Here the Δ_i's are concatenation of partition bases for the sets in $\bar{\Gamma}_n$. We extend the Δ_i's with a k-partition base $\Lambda^{k,r}$ for the trace base $[0, r)E_1$. Then, using essentially the same arguments

as in Lemma 7.12 it can be shown that:

$$\begin{aligned} trace^{-1}(C_{\bar{\Gamma}}) &= \bigcup_{i \in \mathcal{I}} \bigcup_{k \in \mathbb{N}} C_{\Delta_i \Lambda^{k,r}} \text{ (}2k\text{ rational parameters of } \Lambda^{k,r} \text{ are suppressed.)} \\ &= \bigcup_{i \in \mathcal{I}} \bigcup_{k \in \mathbb{N}} \bigcup_{q_1, q'_1, \ldots, q_k, q'_k \in \mathbb{Q}_{\geq 0}} C_{\Delta_i \Lambda^{k,r}_{q_1, q'_1, \ldots, q_k, q'_k}}. \end{aligned}$$

Here the union over $q_1, q'_1, \ldots, q_k, q'_k \in \mathbb{Q}_{\geq 0}$ is taken for those rationals that satisfy the usual restrictions, that is, $q_1 < q'_1, \ldots, q_k < q'_k$, and $\sum q'_i < r$. Since $trace^{-1}(C_{\bar{\Gamma}})$ is a countable union of basic sets it is contained in the σ-algebra $\mathcal{F}_{\mathsf{Execs}}$. ■

Theorem 7.14. *$trace : (\mathsf{Execs}, \mathcal{F}_{\mathsf{Execs}}) \rightarrow (\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}})$ is a measurable function.*

Proof. It follows from Lemmas 7.13 and 7.11 that for any trace base Γ, $trace^{-1}(\Gamma)$ is a countable union of disjoint basic sets. This suffices to establish measurability of the *trace* function. ■

Theorem 7.14 implies that corresponding to each probabilistic execution μ of a given task-DPTIOA $\mathcal{A}$ there exists a unique probability measure on the set of traces of $\mathcal{A}$, namely, the image measure of μ under the *trace* function. This probability measure on $\mathsf{Traces}_{\mathcal{A}}$ is called the *trace distribution* corresponding to μ. From Section 7.4.2 we saw that each task schedule ρ of a task-DPTIOA gives rise to a probabilistic execution μ_ρ. Thus, each task schedule also gives a corresponding trace distribution which we denote by $tdist(\mu_\rho)$, or in short as $tdist(\rho)$. More formally, $tdist(\rho) : \mathsf{Traces}_{\mathcal{A}} \rightarrow [0,1]$, is defined as $tdist(\rho)(E) = \mu_\rho(trace^{-1}(E))$, for any measurable set $E \in \mathcal{F}_{\mathsf{Traces}_{\mathcal{A}}}$. The set of trace distributions of $\mathcal{A}$, $\mathsf{Tdists}_{\mathcal{A}}$, is the set of $tdist(\rho)$'s for any task schedule ρ of $\mathcal{A}$.

7.5 Implementation and Compositionality

In the previous section we defined the set of trace distributions of a task-DPTIOA $\mathcal{A}$. A natural definition for implementation of task-DPTIOAs would be to say $\mathcal{A}_1$ implements $\mathcal{A}_2$, if and only if $\mathsf{Tdists}_{\mathcal{A}_1} \subseteq \mathsf{Tdists}_{\mathcal{A}_2}$. This definition, however, does not give the desirable substitutivity property of the type stated by Theorem 2.5, for example. Instead, based on the idea presented in [CCK+06a], we define a notion of *external behavior* and show that the implementation relation based on this external behavior is compositional. We formulate the external behavior of a $\mathcal{A}$ as a mapping from possible "environments" for $\mathcal{A}$ to sets of trace distributions that can arise when $\mathcal{A}$ is composed with the given environment.

Definition 7.12. An *environment* for task-DPTIOA $\mathcal{A}$ is a PTIOA $\mathcal{E}$ such that $\mathcal{A}$ and $\mathcal{E}$ are compatible and their composition $\mathcal{A}||\mathcal{E}$ is closed. The *external behavior* of a task-DPTIOA $\mathcal{A}$, written as $\mathsf{ExtBeh}_{\mathcal{A}}$, is defined as a function that maps each environment PTIOA $\mathcal{E}$ for $\mathcal{A}$ to the set of trace distributions $\mathsf{Tdists}_{\mathcal{A}||\mathcal{E}}$.

Definition 7.13. Two task-DPTIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if $E_1 = E_2$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable then $\mathcal{A}_1$ is said to *implement* $\mathcal{A}_2$, written as $A_1 \leq A_2$ if, for every environment PTIOA $\mathcal{E}$ for both $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathsf{ExtBeh}_{\mathcal{A}_1}(\mathcal{E}) \subseteq \mathsf{ExtBeh}_{\mathcal{A}_2}(\mathcal{E})$.

This definition of implementation as a functional map from environment automata gives us the desired compositionality result for task-DPTIOAs.

Theorem 7.15. *Suppose $\mathcal{A}_1, A_2$ and $\mathcal{B}$ are task-DPTIOAs, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable and $\mathcal{A}_1 \leq \mathcal{A}_2$. If $\mathcal{B}$ is compatible with $\mathcal{A}_1$ and $\mathcal{A}_2$ then $\mathcal{A}_1||\mathcal{B} \leq \mathcal{A}_2||\mathcal{B}$.*

Proof. Let $\mathcal{E}$ be an environment task-DPTIOA for both $\mathcal{A}_1||\mathcal{B}$ and $\mathcal{A}_2||\mathcal{B}$. Consider a task schedule ρ_1 for the composed task-DPTIOA $(\mathcal{A}_1||\mathcal{B})||\mathcal{E}$. Let $\eta = tdist(\rho_1)$ be the trace distribution of $(\mathcal{A}_1||\mathcal{B})||\mathcal{E}$ generated by ρ_1. It suffices to show that η is also a trace distribution of $(\mathcal{A}_2||\mathcal{B})||\mathcal{E}$, generated by some task schedule.

As ρ_1 is a task schedule for $\mathcal{A}_1||(\mathcal{B}||\mathcal{E})$ it generates the same trace distribution η for PTIOA $\mathcal{A}$ composed with the environment $\mathcal{B}||\mathcal{E}$. Further $\mathcal{B}||\mathcal{E}$ is also a closing environment for $\mathcal{A}_2$ because $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible. As $A_1 \leq A_2$, there exists a task schedule ρ_2 for $\mathcal{A}_2||(\mathcal{B}||\mathcal{E})$ that generates the trace distribution η. It follows that ρ_2 is a task schedule for $(A_2||\mathcal{B})||\mathcal{E}$ that produces the trace distribution η. ■

7.6 PTIOAs and Local Schedulers

In this section, we develop the probabilistic semantics for PTIOAs that do not necessarily satisfy the determinism axioms **D1**, **D2**, and **D3**. This development relies on *local schedulers* which are task-DPTIOAs.

Definition 7.14. A *Generalized PTIOA* $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T}, \mathcal{R})$, where all the components except $\mathcal{T}$ and $\mathcal{R}$ are defined as in Definition 7.1.

(a) $\mathcal{T}$ is a set of trajectories for Q that is closed under prefix, suffix, and concatenation. Further, for any $\mathbf{x} \in Q, \wp(\mathbf{x}) \in \mathcal{T}$.

(b) $\mathcal{R}$ is an equivalence relation on the local actions. The equivalence classes of $\mathcal{R}$ are called *tasks*.

Indeed, a generalized PTIOA $\mathcal{A}$ is similar to a task-PTIOA except that its trajectories are not necessarily deterministic and it may not necessarily satisfy **D1**, **D2**, and **D3**. State of a $\mathcal{A}$ may change through nondeterministic choice of actions and also through choice of distinct, non-point trajectories starting.

Definition 7.15. A *local scheduler* for generalized PTIOA $\mathcal{A} = (X, (Q, \mathcal{F}_Q), \bar{\mathbf{x}}, A, \mathcal{D}, \mathcal{T}, \mathcal{R})$, is a task-DPTIOA $\mathcal{S} = (X_\mathcal{A}, (Q_\mathcal{A}, \mathcal{F}_{Q_\mathcal{A}}), \bar{\mathbf{x}}_\mathcal{A}, A_\mathcal{A}, \mathcal{D}', \mathcal{T}', \mathcal{R}_\mathcal{A})$ that is identical to $\mathcal{A}$ except that $\mathcal{D}' \subseteq D_\mathcal{A}$ and $\mathcal{T}' \subseteq \mathcal{T}_\mathcal{A}$, such that $\mathcal{T}'$ is deterministic and $\mathcal{S}$ satisfies **D1-3**.

A probabilistic system captures the notion of possible ways of resolving internal nondeterminism in a generalized PTIOA. Formally, a *probabilistic-system* is a pair $\mathcal{M} = (\mathcal{A}, \mathscr{L})$, where $\mathcal{A}$ is a generalized PTIOA and $\mathscr{L}$ is a set of local schedulers for $\mathcal{A}$. The notions of probabilistic execution and trace distribution defined earlier for task-DPTIOAs, carry over naturally to generalized PTIOAs. A probabilistic execution for $\mathcal{M}$ is defined to be any probabilistic execution of $\mathcal{S}$, for any $\mathcal{S} \in \mathscr{L}$.

Compatibility and composition of generalized PTIOAs is defined in the same way as in the case of task-DPTIOAs (see, Definition 7.4). We remind the reader that the composition $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ of compatible generalized PTIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, is also a generalized PTIOA only if $\mathcal{A}$ satisfies **M2**. In order to avoid restating this condition in all our definitions and

results, we assume that whenever two compatible generalized PTIOAs are composed, their composition satisfies **M2**, and hence is a generalized PTIOA.

An *environment* for $\mathcal{M}$ is any generalized PTIOA $\mathcal{E}$ such that $\mathcal{A}||\mathcal{E}$ is closed. For probabilistic system $\mathcal{M} = (\mathcal{A}, \mathscr{L})$, we define the *external behavior* of $\mathcal{M}$ to be the total function $\mathsf{ExtBeh}_{\mathcal{M}}$ that maps each environment PTIOA $\mathcal{E}$ for $\mathcal{M}$ to the set $\cup_{\mathcal{S}' \in \mathscr{L}} \mathsf{Tdists}_{\mathcal{S}'||\mathcal{E}}$. Thus, for each environment, we consider the set of trace distributions that arise from the choices of the local scheduler of $\mathcal{M}$ and the task scheduler ρ. This leads to a notion of implementation of probabilistic systems, similar to that of PTIOAs.

Definition 7.16. Let $\mathcal{M}_1 = (\mathcal{A}_1, \mathscr{L}_1)$ and $\mathcal{M}_2 = (\mathcal{A}_2, \mathscr{L}_2)$ be probabilistic systems such that $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable generalized PTIOAs. Then, $\mathcal{M}_1$ is said to *implement* $\mathcal{M}_2$ if for every environment $\mathcal{E}$ of $\mathcal{M}_1$ and $\mathcal{M}_2$, $\mathsf{ExtBeh}_{\mathcal{M}_1}(\mathcal{E}) \subseteq \mathsf{ExtBeh}_{\mathcal{M}_2}(\mathcal{E})$.

Two probabilistic systems $\mathcal{M}_1 = (\mathcal{A}_1, \mathscr{L}_1)$ and $\mathcal{M}_2 = (\mathcal{A}_2, \mathscr{L}_2)$ are compatible if $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible, and their composition $\mathcal{M}_1||\mathcal{M}_2$ is defined as $(\mathcal{A}_1||\mathcal{A}_2, \mathscr{L})$, where $\mathscr{L}$ is the set of local schedulers $\{\mathcal{S}_1||\mathcal{S}_2 \mid \mathcal{S}_1 \in \mathscr{L}_1 \text{ and } \mathcal{S}_2 \in \mathscr{L}_2\}$. Theorem 7.16 gives the following sufficient condition for implementation of probabilistic systems: each local scheduler for the concrete probabilistic system must always correspond to the same local scheduler for the abstract.

Theorem 7.16. *If $\mathcal{M}_1 = (\mathcal{A}_1, \mathscr{L}_1)$, $\mathcal{M}_2 = (\mathcal{A}_2, \mathscr{L}_2)$ are comparable and there exists $f : \mathscr{L}_1 \to \mathscr{L}_2$, such that for all $\mathcal{S}_1 \in \mathscr{L}_1$, $\mathcal{S}_1$ implements $f(\mathcal{S}_1)$, then $\mathcal{M}_1$ implements M_2.*

Proof. Fix an environment PTIOA $\mathcal{E}$ of probabilistic system $\mathcal{M}_1$ and $\mathcal{M}_2$. The external behavior of $\mathcal{M}_1$ with environment $\mathcal{E}$, is defined as $\mathsf{ExtBeh}_{\mathcal{M}_1}(\mathcal{E}) = \cup_{\mathcal{S}_1 \in \mathscr{L}_1} \mathsf{Tdists}_{\mathcal{S}_1||\mathcal{E}}$. Consider any $\mathcal{S}_1 \in \mathscr{L}_1$. For every $\mathcal{S}_1 \in \mathscr{L}_1$, S_1 implements $f(\mathcal{S}_1)$, that is, $\mathsf{Tdists}_{\mathcal{S}_1||\mathcal{E}} \subseteq \mathsf{Tdists}_{f(\mathcal{S}_1)||\mathcal{E}}$. Taking the union over all $\mathcal{S}_1 \in \mathscr{L}_1$,

$$\bigcup_{\mathcal{S}_1 \in \mathscr{L}_1} \mathsf{Tdists}_{\mathcal{S}_1||\mathcal{E}} \subseteq \bigcup_{\mathcal{S}_1 \in \mathscr{L}_1} \mathsf{Tdists}_{f(\mathcal{S}_1)||\mathcal{E}} \subseteq \bigcup_{\mathcal{S}_2 \in \mathscr{L}_2} \mathsf{Tdists}_{\mathcal{S}_2||\mathcal{E}} = \mathsf{ExtBeh}_{\mathcal{M}_2}(\mathcal{E}).$$

It follows that $\mathsf{ExtBeh}_{\mathcal{M}_1}(\mathcal{E}) \subseteq \mathsf{ExtBeh}_{\mathcal{M}_2}(\mathcal{E})$, that is, $\mathcal{M}_1$ implements $\mathcal{M}_2$. ∎

7.7 A Language for Specifying PTIOAs

In this section, we present a for specifying task-DPTIOAs. This language is a modification of the `HIOA` language we introduced in Chapter 3 for specifying SHIOAs. The syntax and semantics for this modified language is identical to those of `HIOA` with the following key distinctions.

Variables. Automaton specifications have only **internal** or state variables and no input/output variables. A variable v of type S gives rise to a topological space $(S, \mathscr{T})$; $\mathscr{T}$ is the Euclidean topology on S if S is a compact subset of $\mathbb{R}^n$ and v is continuous variables, otherwise $\mathscr{T}$ is the discrete topology on S. The Borel σ-algebra generated by $\mathscr{T}$ defines the measurable space $(S, \mathcal{F}_S)$ associated with the variable v. The state space of the automaton is defined as the measurable space obtained by taking the product of the individual variable spaces.

For example, the variables *now* and *flag* in Figure 7-1 are associated with the measurable spaces $(\mathbb{R}, \mathscr{B})$ and $(\{0, 1\}, \mathscr{D}\}$, where $\mathscr{B}$ is the Borel σ-algebra on $\mathbb{R}$ and $\mathscr{D}$ is the collection of sets $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.

Initial state. Each state variable is initialized to a single value. These initial valuations together define the unique starting state of the automaton.

Transitions. The precondition-effect style of HIOA is used to define the transitions. However, the effects of transitions are either purely deterministic or probabilistic. That is, nondeterministic **choose** statements are not allowed. Probabilistic transitions are defined by a new type of **choose** statement that specifies the *distribution according to which this choice is made*.

For example, the statement $x_s :=$ **choose** $\texttt{Normal}(x, \sigma)$ of in Figure 7-1 means that x_s is chosen according to a normal probability distribution with mean x and standard deviation σ. The statement $toss :=$ **choose** $\{1, 0\}$ $\texttt{Disc}[\frac{1}{2}, \frac{1}{2}]$ means that $toss$ is assigned a value 0 or 1 with probability $\frac{1}{2}$ each.

Trajectories. State models are used to specify the set of trajectories of a PTIOA. An additional requirement is that the set of trajectories should be deterministic, i.e., the state models should have differential equations with unique solutions.

The state model normal of Example 7.2 specifies that along any trajectory the continuous variable *now* increases at the same rate as that of real time, until the stopping condition is satisfied. Hence, normal specifies a deterministic set of trajectories.

Tasks. A new language construct is introduced to HIOA for specifying the set of tasks of the PTIOA. The **tasks** section of the code specifies the tasks as a sequence of sets of actions. In the PTIOA of Example 7.2, sample and send actions constitute the one and only task.

In the remaining part of this section we use this language for specifying two typical task-PTIOAs and then, we illustrate some of the concepts we defined earlier in the context of these examples.

Example 7.2. The automaton NoisySensor(d, σ) of Figure 7-1 specifies a sensor that periodically reports the sampled value x_s of a variable x that is evolving according to the differential equation $d(x) = f(x)$. The value is reported through the occurrence of the send(x_s) action which immediately follows the sample action. The sample action occurs periodically, every d time units, and assigns a value to x_s. The state variables, actions, transitions, and trajectories are specified in the same way as in HIOA specifications with the following differences. It is routine to check that NoisySensor satisfies the axioms of Definition 7.2. This automaton has several similarities with the Sensor automaton of Figure 4-4. Here, the sampled value x_s is chosen according to a normal probability distribution with mean x and standard deviation σ whereas in Sensor, the choice of the sampled value x_0 was made nondeterministically over the interval $[\theta_s^0 - e_0, \theta_s^0 + e_0]$. In many applications, such as sensors, probabilistic information about uncertainties is available, and in those cases PTIOA models can capture this extra information.

Example 7.3. The Ben-Or consensus protocol [BO83] is a randomized algorithm for n fault-prone processors to agree on a valid value by communicating over an asynchronous network. The algorithm proceeds in a sequence of stages. In each stage, nonfaulty processes send and receive messages based on coin-flips and comparison of values. With perfectly unbiased coins, a stage ends successfully with probability $\frac{1}{2^n}$, and all nonfaulty processes agree on a value. After one communication round of a successful stage the consensus value is disseminated. An unsuccessful stage is followed by the beginning of the next stage.

```
automaton NoisySensor(d, σ : Real)
signature
  internal sample
  output send(x_s : Real)

variables
  internal now : Real := 0;
    next_sample : AugmentedReal := 0;
    x : Real := 0; x_s : Real := 0;
    flag : Bool := false;

transitions
  internal sample
    pre now = next_send;
    eff x_s := choose Normal(x, σ);
      flag := true; next_send := now + d;

  output send(m)
    pre flag ∧ (x_s = m);
    eff flag := false;

trajectories
  trajdef normal
    stop when now = next_send ∨ flag;
    evolve d(x) = f(x);

tasks
  {send, sample}
```

Figure 7-1: Noisy sensor.

The Consensus PTIOA of Figure 7-2 abstracts the actual computation performed by the processes as well as the messages exchanged, and specifies the termination behavior of the Ben-Or protocol in a terms of number of stages and time elapsed. The *stage* variable represents the current stage of the protocol. The *phase* variable is 0 at the beginning of a stage, 1 when a stage completes successfully, and 2 when the protocol terminates at all nonfaulty processes.

The try action models the computation and communication within a stage. With probability $1 - \frac{1}{2^n}$ it leads to the next stage and with probability $\frac{1}{2^n}$ it leads to phase 1 of the current stage. This is specified by the probabilistic **choose** statement which assigns to *phase* the value 1 with probability $\frac{1}{2^n}$ and 0 with probability $1 - \frac{1}{2^n}$. The decide action marks the termination of the protocol.

The **trajectories** section specifies that along any trajectory, *timer* increases at the same rate as real time, and that all other variables remain constant. The amount of time that elapses in phase 0 owing to message delays is modeled by an exponential distribution with parameter λ_0. Specifically, the *delay* variable is assigned a value chosen from this distribution and **stop when** condition together with the **pre** condition of try forces the action to occur when *timer* equals *delay*. Likewise, the amount of time that elapses in phase 1 is modeled by an exponential distribution with parameter λ_1. The **tasks** section of the code specifies the two tasks of the automaton.

A typical execution of the Consensus automaton is a sequence

$$\alpha = \tau_0 \text{ try } \tau_1 \text{ try } \tau_2 \text{ decide } \tau_3,$$

were each τ_i, $i \in \{0, \ldots, 3\}$, is a trajectory over which *timer* increases monotonically at the same rate as real time and all other variables remain constant. The length of the trajectory τ_i, $i \in \{0, 1, 2\}$ in each stage is determined by the value of the variable *delay*, which is chosen according to the exponential distribution with parameter λ_0. The corresponding trace is:

$$trace(\alpha) = (\tau_0 \frown \tau_1 \frown \tau_2 \downarrow \emptyset) \text{ decide } (\tau_3 \downarrow \emptyset).$$

Notice that the trace contains information about the total time elapsed before decide occurs but not the amount of time that is spent in individual stages.

Let us examine how Definition 8.3 assigns probability measures to the basic sets of the

```
Consensus(n : Nat, λ0, λ1, p : Real)
  where λ0, λ1 > 0, 0 < p < 1
signature
  internal try
  output decide

variables
  internal stage : Nat := 1; phase : {0,1,2} := 0;
    timer : Real := 0;
    delay : AugmentedReal := t0;

transitions
  output decide
    pre  timer = delay ∧ phase = 1;
    eff timer := 0; phase:= 2; delay:= ∞;

  output try
    pre  timer = delay ∧ phase = 0;
    eff timer := 0;
      phase:= choose {1,0} Disc{1/2^n, 1 − 1/2^n};
      if phase = 0
        then stage:= stage + 1; delay:= choose Exp(λ0);
        else delay:= choose Exp(λ1) fi

trajectories
  trajdef normal
    stop when timer = delay;
    evolve d(timer) = 1;

tasks
  {try}{decide}
```

Figure 7-2: Randomized consensus with exponential message delays.

Consensus automaton. Recall that a state $\mathbf{x}$ of Consensus is an ordered 4-tuple specifying the valuations of the 4 variables $stage, phase, timer$, and $delay$, respectively. From the specification in Figure 7-2, $\bar{\mathbf{x}} = (1,0,0,t_0)$ and we define $\bar{\mathbf{x}}' = (1,0,t_0,t_0)$. Suppose $\mu_1 =$ apply$(\delta_{\bar{\mathbf{x}}}, \lambda)$, where λ is the empty task schedule, Let $\Lambda_1 = Q_0 R_0 Q_1$, such that $\bar{\mathbf{x}} \in Q_0 \in \mathcal{F}_Q$, $\mathbf{x}' \in Q_1 \in \mathcal{F}_Q$, and $t_0 \in R_0$. Then, by Definition 8.3, $\mu_1(C_{Q_0}) = 1$ and $\mu_1(C_{\Lambda_1}) = 1$. In fact, for any base $\Lambda_1' = Q_0\, R_0'\, Q_1'$, $\mu_1(C_{\Lambda_1'}) = 1$, if there exists $t < t_0$, such that $t \in R_0'$ and $(1,0,t,t_0) \in Q_1'$. For any base $\Lambda_1\, A_1\, Q_2$ of $\Lambda_1\, R_2\, Q_2'$ that extends Λ_1, the measure assigned is 0.

Next, suppose $\mu_2 =$ apply$(\mu_1, \{\text{try}\})$. Let $\Lambda_2 = \Lambda_1\, \{\text{try}\}\, Q_2$, $Q_2 = \{\mathbf{x} \mid \mathbf{x} \lceil stage = 2, \mathbf{x} \lceil phase = 0, \mathbf{x} \lceil delay \le r_1\}$, for some $r_1 \in \mathbb{R}_{\ge 0}$. Then,

$$\begin{aligned} \mu_2(C_{\Lambda_2}) &= \int_{s\in G} \mu_s(Q_2)\mu_1(d\alpha), \quad \text{where } G = \{\mathbf{x} \mid g_{\Lambda_1,\{\text{try}\}}(\mathbf{x})\}, \\ &= \left(1 - \frac{1}{2^n}\right)(1 - e^{\lambda_0 r_1}). \end{aligned}$$

Since $\mu_1(\alpha) = 1$ for a single execution which is a trajectory starting from $\bar{\mathbf{x}}$ and ending at $\mathbf{x}'$, in the above case the integral of Equation (7.3) reduces to $\mu_{\mathbf{x}'}(Q_2)$. Suppose $\Lambda_3 = \Lambda_2\, R_1\, Q_3$ where $R_1 = [0, r_1']$, where $r_1' \le r_1$, and $Q_3 = \{\mathbf{x} \mid \mathbf{x} \lceil stage = 2, \mathbf{x} \lceil phase = 0, \mathbf{x} \lceil timer \le \mathbf{x} \lceil delay \le r_1\}$. Then Equation (7.4) gives: $\mu_2(C_{\Lambda_3}) = \frac{r_1'}{r_1}\frac{1}{2^n}(1 - e^{\lambda_0 r_1})$. Suppose $Q_2' = \{\mathbf{x} \mid \mathbf{x} \lceil stage = 0, \mathbf{x} \lceil phase = 1, \mathbf{x} \lceil delay \le r_2\}$, $\Lambda_2' = \Lambda_1\, \{\text{try}\}\, Q_2'$, then

$$\begin{aligned} \mu_2(C_{\Lambda_2'}) &= \int_{s\in G} \mu_s(Q_2)\mu_1(d\alpha), \quad \text{where } G = \{\mathbf{x} \mid g_{\Lambda_1,\{\text{try}\}}(\mathbf{x})\}, \\ &= \frac{1}{2^n}(1 - e^{\lambda_0 r_2}). \end{aligned}$$

Taking another step, suppose $\mu_3 =$ apply$(\mu_2, \{\text{decide}\})$. Let $Q_4 = \{\mathbf{x} \mid \mathbf{x} \lceil stage = 0, \mathbf{x} \lceil phase = 2, \mathbf{x} \lceil delay \le r_2\}$, where $r_2 \in \mathbb{R}_{\ge 0}$. From Equation (7.3) we obtain:

$$\begin{aligned} \mu_3(C_{\Lambda_2'\{\text{decide}\}\, Q_4}) &= \frac{1}{2^n}(1 - e^{\lambda_0 r_1})(1 - e^{\lambda_1 r_2}), \\ \mu_3(C_{\Lambda_2}) &= \mu_2(C_{\Lambda_2}) = \left(1 - \frac{1}{2^n}\right)(1 - e^{\lambda_0 r_1}). \end{aligned}$$

The distributions μ_1, μ_2, and μ_3 are the probabilistic executions of $\mathsf{Consensus}$ corresponding to the task schedules λ, $\{\mathsf{try}\}$, and $\{\mathsf{try}\}\{\mathsf{decide}\}$.

7.8 Summary

We have introduced PTIOAs for modelling and discretely communicating probabilistic hybrid systems. The non-probabilistic aspects of the behavior of a PTIOA, namely, its executions and its traces are defined in the same way as in the case of SHIOAs. In order to associate probability measures with sets of executions of a PTIOA, first, we had to ensure measurability of all reasonable sets of executions. For this purpose, we found it natural to work with a measurable state space Q, that is, Q associated with the additional structure of a σ-algebra $\mathcal{F}_Q$. Introducing two new measurability axioms, namely, **M1** and **M2** we constructed the σ-algebra $\mathcal{F}_{\mathsf{Execs}_\mathcal{A}}$ over the space of executions of task-DPTIOA $\mathcal{A}$. We showed that a PTIOA $\mathcal{A}$ combined with a local scheduler (for resolving internal nondeterminism), and an oblivious task scheduler (for resolving external nondeterminism) gives rise to a probabilistic execution—a probability distribution over the measurable space of executions $(\mathsf{Execs}_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}_\mathcal{A}})$. The probabilistic behavior of PTIOAs is defined in terms of sets of probabilistic executions—one for each possible local and task schedule. Based on probabilistic execution we define trace distributions and implementation relations for PTIOAs.

We show that the composition of two PTIOAs is also a PTIOA only if the composite automaton satisfies an additional measurability requirement, namely, axiom **M2**. This could be improved by imposing restrictions on the trajectories or with additional measurability requirements, so that a subclass of PTIOAs is closed under composition. In the future we will extend PTIOAs to support external variables. For including input variables one has to remove the the time-action determinism axiom **D3** because the stopping points of a trajectory would also depend on inputs. We also intend to develop a suite of verification techniques for quantitative properties such as, probabilistic safety and stochastic stability [CL06]. In Chapter 8 we present techniques for verifying approximate implementation relations for a class of "discrete" PTIOAs. Such techniques are particularly relevant in context of PTIOAs because inaccuracies related to timing information can be captured by approximate abstractions. A longer term research direction is to incorporate continuous stochastic behavior over trajectories.

Chapter 8

Verifying Approximate Implementation Relations

In this chapter we introduce the notion of approximate implementation for Probabilistic I/O Automata (PIOA) and develop methods for proving such relationships. A PIOA is essentially a "discrete" pre-PTIOA (of Chapter 7); it does not have trajectories and only discrete probability distributions appear in transition definitions. The nondeterminism in a PIOA is resolved by task schedulers which gives rise to probabilistic executions and trace distributions. A PIOA $\mathcal{A}$ is said to implement PIOA $\mathcal{B}$, in the traditional sense, if each trace distribution of $\mathcal{A}$ is also a trace distribution of $\mathcal{B}$. But small perturbations to the parameters of $\mathcal{A}$ produces traces distributions with slightly different probabilities and this breaks the implementation relation. We would like to define and verify implementation relations that not only capture the binary fact of whether or not $\mathcal{A}$ implements $\mathcal{B}$, but also the degree to which $\mathcal{A}$ implements $\mathcal{B}$. To this end, we develop new notions of approximate implementation for PIOAs and generalizing existing simulation relations propose real-valued simulation functions for proving approximate implementations.

In Section 8.2.2 we briefly describe the semantics of PIOAs. We relax the notion of implementation by taking into consideration the "similarity" of trace distributions that are not exactly equal. In Sections 8.3 and 8.4 we introduce two different metrics for measuring similarity of trace distributions; each of these give rise to corresponding notions of approximate implementation and we develop simulation-based techniques for verifying them. In Sections 8.3.5 and 8.4.3, we discuss applications of approximate implementations to verification of probabilistic safety and termination.

8.1 An Overview

Implementation relations play a fundamental role in the study of complex interacting systems because they allow us to prove that a given concrete system implements an abstract specification. An automaton $\mathcal{A}$ is said to implement another automaton $\mathcal{B}$ if the observable behavior of the first is subsumed by that of the latter. In the non-probabilistic setting, we saw applications of implementation relations in verifying safety (Chapter 4) and stability (Chapter 5). These notions of implementation rely on equality of observable behavior. That is, every observable behavior of the concrete system must be exactly equal to some observable behavior of the abstract specification. It has been well-known for some time now that such strict equality based implementation relations are not robust (see,

e.g., [JS90, DJGP02, GJP04]). Small perturbations to the parameters of the system produces traces with slightly different numbers (representing say, timing or probability information), and thus breaks the equality between traces. One way to overcome this problem is to relax the notion of implementation by taking into consideration the "similarity" of traces using a metric. Apart from providing robust implementation relations, notions of approximate implementation also enable us to create abstract models without introducing extra nondeterminism. For instance, a complex trajectory τ of a hybrid system can be abstracted by a simpler set of trajectories which envelop τ. Such an abstraction achieves simplicity, and perhaps tractability, but sacrifices determinism. On the other hand, τ could be approximately abstracted by a single simpler trajectory τ', but now, the abstract model is not implemented by the concrete model, at least not in the traditional sense. We can, however, quantify the closeness of the abstract and the concrete systems using some appropriate notion of approximate implementation based some metric over trajectories.

Probabilistic I/O Automata (PIOA) were introduced by Segala in [Seg95b, Seg95a] for the purpose of modeling and verifying untimed, randomized, distributed algorithms. Task-structured PIOAs (task-PIOAs) [CCK+06c, CCK+06a] introduce an additional *task structure* to PIOAs which makes it possible to resolve nondeterminism by a simple sequence of tasks. In this chapter we introduce the notion of approximate implementations for task-PIOAs. Parts of this work previously appeared in [ML06] and [ML07]. A task-PIOA is a nondeterministic automaton with a countable state space. A task-PIOA interacts with a *task scheduler* to give rise to a trace distribution—a probability distributions over its traces. A task-PIOA is said to (exactly) implement another task-PIOA if the set of trace distributions of the first is a subset of the trace distributions of the latter. Implementations, simulation relations for proving implementations, and compositionality results for task-PIOAs are presented in [CCK+06b]. A special kind of approximate implementation relation that tolerates small differences in the probability of occurrence of a particular action is used in [CCK+06d] to verify a security protocol. In contrast, the notions of approximation introduced here can be used to quantify discrepancies in the distributions of any (measurable) function, that is, any random variable on the space of trace distributions.

In Section 8.3 we define uniform approximate implementations for task-PIOAs based on the uniform metric on trace distributions. A PIOA $\mathcal{A}$ is a δ-approximate implementation of another PIOA $\mathcal{B}$, for a positive δ, if the for any trace distribution of $\mathcal{A}$, there exists a trace distribution of $\mathcal{B}$ such that their discrepancy, with respect to the uniform metric, over any measurable set of traces is at most δ. We present *Expanded Approximate Simulations (EAS)* for proving uniform approximate implementations. EAS is a natural generalization of the simulation relation presented in [CCK+06d]. Let μ_1 and μ_2 be probability distributions over executions of task-PIOAs $\mathcal{A}$ and $\mathcal{B}$. An EAS from $\mathcal{A}$ to $\mathcal{B}$ is a function ϕ mapping each μ_1, μ_2 pair to a nonnegative real. The number $\phi(\mu_1, \mu_2)$, is a measure of how similar μ_1 and μ_2 are in terms of producing similar trace distributions. Informally, if $\phi(\mu_1, \mu_2) \leq \epsilon$, for some $\epsilon \geq 0$, then it is possible to closely (with respect to the uniform metric on trace distributions) simulate from μ_2 anything that can happen from μ_1, and further, the resulting distributions, say μ_1' and μ_2', are also close in the following sense. There exists a joint distribution ψ supported on the set $\{(\eta_1, \eta_2) \mid \phi(\eta_1, \eta_2) \leq \epsilon\}$ such that the first and the second marginal distributions of ψ have means μ_1' and μ_2'. Informally, this means that μ_1' and μ_2' can be decomposed into a set of measures that are close in the sense of ϕ.

In Section 8.4, we define discounted uniform approximate implementations of task-PIOAs based on the discounted uniform metric on trace distributions, respectively. Uniform approximate implementations are useful for deducing probabilistic safety properties. How-

ever, since they gives absolute bounds on the discrepancy over any set of traces, they do not give us useful information when the probability of the set itself is smaller than the approximation factor δ. In order to obtain useful bounds on the discrepancies over a sequence of sets of traces that have monotonically decreasing probabilities, the notion of discounted uniform approximate implementation is useful. PIOA $\mathcal{A}$ to be a δ_k-discounted approximate implementation of $\mathcal{B}$, if the for any trace distribution of $\mathcal{A}$, there exists a trace distribution of $\mathcal{B}$ such that their discrepancy *over any trace of length* k is at most δ_k; here $\{\delta_k\}_{k\in\mathbb{N}}$ is a collection of *discount factors*. We define *Discounted Approximate Simulations (DAS)* in a similar way as we defined EAS and prove that they are sound for proving discounted approximate implementations. We demonstrate the utility of discounted approximate implementations and DASs by proving that the probability of termination of an ideal randomized consensus protocol (after a certain number of rounds) is close to the same probability for a protocol that uses biased coins.

Throughout the chapter we assume that the task-PIOAs in question are closed and in Section 8.5 we outline how our results extend to general (not necessarily closed) task-PIOAs. We start the chapter by giving the basic definitions and results for task-PIOAs; we refer the reader to [CCK+05] for a detailed treatment.

8.2 Task-structured PIOA

In this section we present the basic definitions and results for task-PIOAs. The notations for sets, functions, and variables from Section 2.1 of Part I and those for σ-algebras, measurable spaces, and measures from Section 7.2, have the same meanings in this chapter. Given a set Q, we denote a σ-algebra over Q by $\mathcal{F}_Q$, the set of discrete (sub-) probability measures on X by $\mathsf{Disc}(Q)$ (resp. $\mathsf{SubDisc}(Q)$). If μ is a discrete probability or sub-probability measure on X, the *support* of μ, written as $\mathsf{supp}(\mu)$, is the set of elements of X that have non-zero measure.

8.2.1 Definition of Task Structured PIOA

The task-PIOA model used in this chapter is slightly more general than the one in [CCK+06b] because we allow the starting configuration of an automaton to be any distribution over states and not just a Dirac mass.

Definition 8.1. A *task-structured Probabilistic I/O Automaton (task PIOA)* $\mathcal{A} = (X, Q, \bar{\nu}, A, D, \mathcal{R})$, where:

(a) X is a set of *internal* or *state variables*,

(b) $Q \subseteq val(X)$ is a countable set of *states*, $\bar{\nu} \in \mathsf{Disc}(Q)$ is the *starting distribution* on states;

(c) A is a countable set of *actions*, partitioned into *internal* H, *input* I, and *output* O actions. $L = O \cup H$ is the set of *local actions* and $E = O \cup I$ is the set of *external actions*. The set of *external* actions of $\mathcal{A}$ is $E := I \cup O$ and the set of *locally controlled* actions is $L := O \cup H$.

(d) $D \subseteq Q \times A \times \mathsf{Disc}(Q)$ is set of *discrete transitions*. An action a is *enabled* in a state $\mathbf{x}$ if $(\mathbf{x}, a, \mu) \in D$ for some μ. In this case, we write $\mathbf{x} \stackrel{a}{\rightarrow} \mu$.

(e) $\mathcal{R}$ is an equivalence relation on the locally controlled actions. The equivalence classes of $\mathcal{R}$ are called *tasks*. A task T is *enabled* in a state $\mathbf{x}$ if some action $a \in T$ is enabled in $\mathbf{x}$.

In addition, $\mathcal{A}$ satisfies:

E1 (Input enabling) For every $\mathbf{x} \in Q$ and $a \in I$, $\mathbf{x} \in E_a$.

D1 (Transition determinism) For every $\mathbf{x} \in Q$ and $a \in A$, there is at most one $\mu \in \mathsf{Disc}(Q)$ such that $(\mathbf{x}, a, \mu) \in D$.

D2 (Action determinism) For every $\mathbf{x} \in Q$ and $T \in R$, at most one $a \in T$ is enabled in $\mathbf{x}$.

Notations. A task PIOA is *closed* if its set of input actions is empty. We denote the components of a task PIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Q_{\mathcal{A}}, \bar{\mathbf{x}}_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}$ etc., and the components of a task PIOA $\mathcal{A}_i$ by $X_i, Q_i, \mathbf{x}_i, \mathcal{D}_i$, etc.

The input enabling axiom **E1** is identical to the standard non-blocking axiom for non-probabilistic and probabilistic hybrid system models (see, Definitions 2.7 and 7.1. Of course, the other non-blocking axiom, namely, **E2** input trajectory enabling, is not relevant for PIOAs. Also, the measurability axioms, such as **M1** and **M2** of PTIOAs, are unnecessary because: (a) the state space Q is countable and it is equipped with the σ-algebra generated by the discrete topology on Q. That is, every subset of Q is measurable. And (b) the transitions give rise to discrete probability distributions. Axioms **D1** and **D2** are the same as in the case of task-DPTIOAs (see, Definition 7.2). These axioms together with the task structure on local actions enable us to systematically resolve the nondeterminism in a PIOA.

8.2.2 Executions and Traces

An *execution fragment* of $\mathcal{A}$ is a finite or infinite sequence $\alpha = q_0\, a_1\, q_1\, a_2\, \ldots$ of alternating states and actions, such that (i) if α is finite, then it ends with a state; and (ii) for every non-final i, there is a transition $(q_i, a_{i+1}, \mu) \in D$ with $q_{i+1} \in \mathsf{supp}(\mu)$. We write $\alpha.fstate$ for q_0, and, if α is finite, we write $\alpha.lstate$ for its last state. We use $\mathsf{Frags}_{\mathcal{A}}$ (resp., $\mathsf{Frags}^*_{\mathcal{A}}$) to denote the set of all (resp., all finite) execution fragments of $\mathcal{A}$. An *execution* of $\mathcal{A}$ is an execution fragment beginning from some state in $\mathsf{supp}(\bar{\nu})$. $\mathsf{Execs}_{\mathcal{A}}$ (resp., $\mathsf{Execs}^*_{\mathcal{A}}$) denotes the set of all (resp., finite) executions of $\mathcal{A}$. The *trace* of an execution fragment α, written $trace(\alpha)$, is the restriction of α to the set of external actions of $\mathcal{A}$. We say that β is a *trace* of $\mathcal{A}$ if there is an execution α of $\mathcal{A}$ with $trace(\alpha) = \beta$. $\mathsf{Traces}_{\mathcal{A}}$ (resp., $\mathsf{Traces}^*_{\mathcal{A}}$) denotes the set of all (resp., finite) traces of $\mathcal{A}$.

8.2.3 Composition of Task-PIOAs

The composition operation on task-PIOAs enable us to construct a PIOAs representing a complex system from two interacting PIOAs by identifying their external actions. PIOAs do not have external (input/output) variables and component automata communicate through external actions only.

Composition of a pair of PIOAs is defined as follows:

Definition 8.2. Task-PIOA $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if $X_1 \cap X_2 = H_1 \cap A_2 = H_2 \cap A_1 = O_1 \cap O_2 = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible task-PIOAs then their *composition* $\mathcal{A}_1 || \mathcal{A}_2$ is defined to be $\mathcal{A} = (X, Q, \bar{\nu}, A, \mathcal{D}, \mathcal{R})$, where:

(a) $X = X_1 \cup X_2$,

(b) $Q = Q_1 \times Q_2$, and the initial distribution $\bar{\nu}$ is defined as $\bar{\nu}(\mathbf{x}_1, \mathbf{x}_2) \triangleq \bar{\nu}_1(\mathbf{x}_1) \times \bar{\nu}_2(\mathbf{x})$, for $\mathbf{x}_i \in Q_i, i \in \{1, 2\}$.

(c) $A = A_1 \cup A_2$, $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = O_1 \cup O_2$, $H_1 \cup H_2$,

(d) $D \subseteq Q \times A \times \mathsf{Disc}(Q)$ is the set of triples $((\mathbf{x}_1, \mathbf{x}_2), a, \mu_1 \times \mu_2)$ such that for $i \in \{1, 2\}$, if $a \in A_i$ then $(q_i, a, \mu_i) \in D_i$, otherwise $\mu_i = \delta_{\mathbf{x}_i}$,

(e) $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$.

It can be checked easily that the composite structure $\mathcal{A}$ is also a task-PIOA.

8.2.4 Probabilistic Executions and Trace Distributions

So far we have presented the concepts concerning non-probabilistic behavior of PIOAs. In this section, we present definitions and results from [CCK+06a] which describe the probabilistic semantics of PIOAs.

For any finite execution fragment α of $\mathcal{A}$, the *cone* of α, denoted by C_α, is the set of execution fragments that have α as prefix. The σ-algebra generated by cones of finite execution fragments of $\mathcal{A}$ is denoted by $\mathcal{F}_{\mathsf{Execs}_\mathcal{A}}$. This defines a measurable space $(\mathsf{Execs}_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}_\mathcal{A}})$ in which any set of finite execution fragments is measurable. A probability measure on $(\mathsf{Execs}_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}_\mathcal{A}})$ is called a *probabilistic execution*.

Nondeterministic choices in $\mathcal{A}$ are resolved using a *task scheduler*—an entity which chooses a task—which in turn probabilistically determines the next state of the automaton. One could define various kinds task schedulers by specifying what dynamic information may be used in choosing the next task. We refer the reader to Section 7.4.2 for a discussion on the different kinds of schedulers. The probabilistic semantics of task-PIOAs with a general class of history dependent schedulers has been developed in [CCK+06b]. In this chapter, we restrict our attention to *oblivious* schedulers that do not depend on dynamic information generated during execution. A *task schedule* for $\mathcal{A}$ is any finite or infinite sequence $\rho = T_1 T_2 \ldots$ of tasks in R. Although restrictive this class of schedulers arise naturally in many applications, including in analysis of security protocols [CCK+06c]. A scheduler ρ and a finite execution fragment α *generate* a measure $\mu_{\rho,\alpha}$ on $(\mathsf{Execs}_\mathcal{A}, \mathcal{F}_{\mathsf{Execs}_\mathcal{A}})$.

A task schedule generates a probabilistic execution of the task-PIOA $\mathcal{A}$ by repeatedly scheduling tasks, each of which determines at most one transition of $\mathcal{A}$. Formally, we define an operation that "applies" a task schedule to a task-PIOA:

Definition 8.3. Let $\mathcal{A}$ be an action-deterministic task-PIOA. Given $\mu \in \mathsf{Disc}(\mathsf{Frags}^*_\mathcal{A})$ and a task schedule ρ, $\mathsf{apply}(\mu, \rho)$ is the probability measure on $\mathsf{Frags}_\mathcal{A}$ defined recursively by:

1. $\mathsf{apply}(\mu, \lambda) \triangleq \mu$, where λ denotes the empty sequence of tasks.

2. $\mathsf{apply}(\mu, T) \triangleq p_1 + p_2$, where T is a task and p_1, p_2 are defined as follows: For every

$\alpha \in \mathsf{Frags}^*_{\mathcal{A}}$:

$$p_1(\alpha) \triangleq \begin{cases} \mu(\alpha')\eta(q) & \text{if } \alpha = \alpha'\, a\, q, \text{ such that } a \in T \text{ and } \alpha'.lstate \xrightarrow{a} \eta, \\ 0 & \text{otherwise.} \end{cases}$$

$$p_2(\alpha) \triangleq \begin{cases} \mu(\alpha) & \text{if } T \text{ is not enabled in } \alpha.lstate, \\ 0 & \text{otherwise.} \end{cases}$$

3. $\mathsf{apply}(\mu, \rho) \triangleq \mathsf{apply}(\mathsf{apply}(\mu, \rho'), T)$, where $\rho = \rho'\, T$, $T \in R$.

4. $\mathsf{apply}(\mu, \rho) \triangleq \lim_{i\to\infty}(\mu_i)$, where ρ is infinite ρ_i is the length-i prefix of ρ, and $\mu_i = \mathsf{apply}(\mu, \rho_i)$.

In Case (2) above, p_1 represents the probability that α is executed when applying task T at the end of α'. Because of transition-determinism and action-determinism, the transition $\alpha'.lstate \xrightarrow{a} \eta$ is unique, and so p_1 is well-defined. The term p_2 represents the original probability $\mu(\alpha)$, which is relevant if T is not enabled after α. It is routine to check that the limit in Case (4) is well-defined. The other two cases are straightforward. Given any task schedule ρ, $\mathsf{apply}(\bar{\nu}, \rho)$ is a probability distribution over $\mathsf{Exec}_{\mathcal{A}}$. Several useful properties of the $\mathsf{apply}(,)$ function relating sequences of probability distributions on executions and traces are given in Appendix 8.8.

The *lstate* function is a measurable function from the discrete σ-algebra on finite execution fragments of $\mathcal{A}$ to the discrete σ-algebra of states of $\mathcal{A}$. If μ is a probability measure on execution fragments of $\mathcal{A}$, then we define the lstate distribution of μ, $lstate(\mu)$, to be the image measure of μ under the *lstate* function.

We note that the *trace* function is a measurable function from $\mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}}$ to the σ-algebra generated by cones of traces. Thus, given a probability measure μ on $\mathcal{F}_{\mathsf{Execs}_{\mathcal{A}}}$ we define the *trace distribution* of μ, denoted $tdist(\mu)$, to be the image measure of μ under the *trace* function. We extend the $tdist()$ notation to arbitrary measures on execution fragments of $\mathcal{A}$. We write $tdist(\mu, \rho)$ as shorthand for $tdist(\mathsf{apply}(\mu, \rho))$, the trace distribution obtained by applying task schedule ρ starting from the measure μ on execution fragments. We write $tdist(\rho)$ for $tdist(\mathsf{apply}(\bar{\nu}, \rho))$. A *trace distribution* of $\mathcal{A}$ is any $tdist(\rho)$. We use $\mathsf{Tdists}_{\mathcal{A}}$ to denote the set $\{tdist(\rho) : \rho \text{ is a task schedule for } \mathcal{A}\}$ of all trace distributions of $\mathcal{A}$.

8.2.5 Exact implementations and Simulations

Two task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if they have the same set of external actions. The implementation relation for comparable task-PIOAs is defined in terms of inclusion of sets of trace distributions.

Definition 8.4. Given comparable closed task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1$ is said to *implement* $\mathcal{A}_2$ if $\mathsf{Tdists}_{\mathcal{A}_1} \subseteq \mathsf{Tdists}_{\mathcal{A}_2}$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ implement each other then they are said to be *equivalent*.

In [CCK+06b] a simulation relation for closed, task-PIOAs is defined and it is shown to be sound for proving the above implementation relation. This definition is based on three operations involving probability measures: flattening, lifting, and expansion. First we define these operations.

The *flattening* operation takes a discrete probability measure over probability measures and "flattens" it into a single probability measure. Let Q and P be a sets. If $\eta \in \mathsf{Disc}(\mathsf{Disc}(Q))$, then the *flattening* of η, denoted by $\mathsf{flatten}(\eta) \in \mathsf{Disc}(Q)$, is defined by $\mathsf{flatten}(\eta) = \sum_{\mu \in \mathsf{Disc}(Q)} \eta(\mu)\mu$.

The *lifting* operation takes a relation $R \subseteq Q \times P$ and "lifts" it to a relation $\mathcal{L}(R) \subseteq \mathsf{Disc}(Q) \times \mathsf{Disc}(P)$ Informally, a measure μ_1 on Q is related to a measure μ_2 on P if μ_2 can be obtained by redistributing the probabilities assigned by μ_1, in such a way that the relation R is respected. The *lifting* of R, denoted by $\mathcal{L}(R)$, is a relation from $\mathsf{Disc}(Q)$ to $\mathsf{Disc}(P)$ defined by: $\mu_1 \; \mathcal{L}(R) \; \mu_2$ iff there exists a *weighting function* $w : Q \times P \to \mathbb{R}_{\geq 0}$ such that: (i) for each $q \in Q$ and $p \in P$, $w(q,p) > 0$ implies $q \; R \; p$, (ii) for each $q \in X$, $\sum_y w(q,p) = \mu_1(q)$, and (iii) for each $p \in Y$, $\sum_p w(q,p) = \mu_2(p)$.

Finally, the *expansion* operation takes a $R \subseteq \mathsf{Disc}(P) \times \mathsf{Disc}(Q)$, and returns a relation $\mathcal{E}(R) \subseteq \mathsf{Disc}(Q) \times \mathsf{Disc}(P)$ such that $\mu_1 \; \mathcal{E}(R) \; \mu_2$ whenever they can be decomposed into two $\mathcal{L}(R)$-related measures. Formally, $\mathcal{E}(R)$, is defined by: $\mu_1 \; \mathcal{E}(R) \; \mu_2$ iff there exist two discrete measures η_1 and η_2 on $\mathsf{Disc}(X)$ and $\mathsf{Disc}(Y)$, respectively, such that $\mu_1 = \mathsf{flatten}(\eta_1)$, $\mu_2 = \mathsf{flatten}(\eta_2)$, and $\eta_1 \; \mathcal{L}(R) \; \eta_2$.

The next definition expresses consistency between a probability measure over finite executions and a task schedule. Informally, a measure μ over finite executions is said to be consistent with a task schedule ρ if it assigns non-zero probability only to those executions that are possible under the task schedule ρ. This condition is used to avoid useless proof obligations in the definition of both exact and approximate simulations.

Definition 8.5. Suppose $\mathcal{A}$ is a closed, task-PIOA and ρ is a finite task schedule for $\mathcal{T}$. $\mu \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$ is *consistent with* ρ if $\mathsf{supp}(\mu) \subseteq \mathsf{supp}(\mathsf{apply}(\bar{\nu}, \rho))$.

Suppose we have a mapping c that, given a finite task schedule ρ and a task T of a task-PIOA $\mathcal{A}_1$, yields a task schedule of another task-PIOA $\mathcal{A}_2$. The idea is that $\mathsf{c}(\rho, T)$ describes how $\mathcal{A}_2$ matches task T, given that it has already matched the task schedule ρ. Using c, we define a new function $\mathsf{full}(\mathsf{c})$ that, given a task schedule ρ, iterates c on all the elements of ρ, thus producing a "full" task schedule of $\mathcal{A}_2$ that matches all of ρ.

Definition 8.6. Let $\mathcal{A}_1, \mathcal{A}_2$ be task-PIOAs, and let $\mathsf{c} : ({R_1}^* \times R_1) \to {R_2}^*$ be a function that assigns a finite task schedule of $\mathcal{A}_2$ to each finite task schedule of $\mathcal{A}_1$ and task of $\mathcal{A}_1$. The function $\mathsf{full}(\mathsf{c}) : {R_1}^* \to {R_2}^*$ is recursively defined as:

- $\mathsf{full}(\mathsf{c})(\lambda) \triangleq \lambda$, and
- $\mathsf{full}(\mathsf{c})(\rho T) \triangleq \mathsf{full}(\mathsf{c})(\rho) \, \mathsf{c}(\rho, T)$ (i.e., concatenation of $\mathsf{full}(\mathsf{c})(\rho)$ and $\mathsf{c}(\rho, T)$).

Now we give the definition of exact simulation relation for task-PIOAs. Note that the simulation relations do not just relate states to states, but rather, probability measures on executions to probability measures on executions. The use of measures on executions here rather than just executions is motivated by certain cases that arise in proofs where related random choices are made at different points in the low-level and high-level models (see, e.g., proof of OT protocol in [CCK+06d]).

Definition 8.7. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two comparable closed task-PIOAs. A relation R from $\mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1})$ to $\mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2})$ is a *simulation* from $\mathcal{A}_1$ to $\mathcal{A}_2$ if there exists $\mathsf{c} : ({R_1}^* \times R_1) \to {R_2}^*$ such that following properties hold:

1. *(Start condition)* $\bar{\nu}_1 \; R \; \bar{\nu}_2$.

2. *(Step condition)* If $\mu_1 \; R \; \mu_2$, $\rho \in {R_1}^*$, μ_1 is consistent with ρ, μ_2 is consistent with $\mathsf{full}(\mathsf{c})(\rho)$, and $T \in R_1$, then $\mu_1' \; \mathcal{E}(R) \; \mu_2'$ where $\mu_1' = \mathsf{apply}(\mu_1, T)$ and $\mu_2' = \mathsf{apply}(\mu_2, \mathsf{c}(\rho, T))$.

3. *(Trace condition)* If $\mu_1 \; R \; \mu_2$, then $tdist(\mu_1) = tdist(\mu_2)$.

We close this section with the statement of the soundness theorem for the above simulation relation which has been proved in [CCK+06b].

Theorem 8.1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be comparable closed action-deterministic task-PIOAs. If there exists a simulation relation from $\mathcal{A}_1$ to $\mathcal{A}_2$, then* $\mathsf{Tdists}_{\mathcal{A}_1} \subseteq \mathsf{Tdists}_{\mathcal{A}_2}$.

Theorem 8.1 shows soundness of simulation relations for deducing trace distribution inclusion. This theorem is a discrete-probabilistic analogue of Theorem 4.4 which gave a sound way of proving trace inclusion in the hybrid-nonprobabilistic setting.

8.3 Uniform Approximate Implementation

In this section we define approximate implementations for task-PIOAs based on the uniform metric on trace distributions and propose *Expanded Approximate Simulations (EAS)* as a sound method for proving uniform approximate implementations. Informally, a task-PIOA $\mathcal{A}_1$ should be regarded to be close to a comparable task-PIOA $\mathcal{A}_2$ if they give rise to "similar" trace distributions, where similarity is measured by some metric on the space of trace distributions. Henceforth, we denote by Traces the set of all possible traces of $\mathcal{A}_1$ and $\mathcal{A}_2$. $\mathsf{Disc}(\mathsf{Traces})$ denotes the set of all possible discrete probability measures over Traces.

8.3.1 Uniform Metric on Traces

We say that a task-PIOA $\mathcal{A}_1$ uniformly approximately implements a task-PIOA $\mathcal{A}_2$, if every trace distribution of $\mathcal{A}_1$ is close to some trace distribution of $\mathcal{A}_2$, closeness being defined by the uniform metric on trace distributions. A *metric* $\mathbf{d}$ on a set $\mathcal{X}$ is a function $\mathbf{d}\mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_{\geq 0}$, which satisfies 1. *(identity)* $d(x_1, x_2) = 0 \iff x_1 = x_2$, 2. *(symmetry)* $d(x_1, x_2) = d(x_2, x_1)$, and 3. *(triangle inequality)* $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$ If $\mathbf{d}$ does not satisfy the identity property then it is called a *psuedometric*.

Definition 8.8. Let $\mathcal{A}$ be a closed task-PIOA. The *uniform metric* over trace distributions of $\mathcal{A}$ is the function $\mathbf{d}_u : \mathsf{Disc}(\mathsf{Traces}_{\mathcal{A}}) \times \mathsf{Disc}(\mathsf{Traces}_{\mathcal{A}}) \rightarrow \mathbb{R}_{\geq 0}$ defined by:

$$\mathbf{d}_u(\mu_1, \mu_2) \triangleq \sup_{C \in \mathcal{F}_{\mathsf{Traces}_{\mathcal{A}}}} |\mu_1(C) - \mu_2(C)| \, .$$

The next lemma shows that $\mathbf{d}_u$ has the standard convergence property for limits of sequences of trace distributions.

Lemma 8.2. *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are closed task-PIOAs. For $i \in \{1, 2\}$, let $\{\mu_{ij}\}_{j \in \mathbb{N}}$ be a chain of discrete probability distributions on the traces of $\mathcal{A}_i$ and let $\lim_{j \to \infty} \mu_{ij} = \mu_i$. Then $\lim_{j \to \infty} \mathbf{d}_u(\mu_{1j}, \mu_{2j}) = \mathbf{d}_u(\mu_1, \mu_2)$.*

Proof. We have to show that for every $\epsilon > 0$, there exists $N \in \mathbb{N}$, such that for all $k > N$, $\mathbf{d}_u(\mu_{1k}, \mu_{2k}) - \mathbf{d}_u(\mu_1, \mu_2) < \epsilon$. From triangle inequality, we get that for any k, $\mathbf{d}_u(\mu_{1k}, \mu_{2k}) \leq \mathbf{d}_u(\mu_{1k}, \mu_1) + \mathbf{d}_u(\mu_1, \mu_2) + \mathbf{d}_u(\mu_2, \mu_{2k})$. Therefore, it suffices to show that exists $N \in \mathbb{N}$,

such that for all $k > N$, $\mathbf{d}_u(\mu_{1k}, \mu_1) + \mathbf{d}_u(\mu_2, \mu_{2k}) \leq \epsilon$. Now since $\lim_{j\to\infty} \mu_{1j} = \mu_1$, $\lim_{j\to\infty} \mu_{2j} = \mu_2$, we know that there exists $N' \in \mathbb{N}$, such that for all $k > N'$, for every $C \in \mathcal{F}_{\mathsf{Traces}_{\mathcal{A}_i}}$, $|\mu_{ij}(C) - \mu_i(C)| \leq \frac{\epsilon}{2}$. If we choose $N = N'$, we have for all $k > N$, $\mathbf{d}_u(\mu_{1k}, \mu_1) + \mathbf{d}_u(\mu_2, \mu_{2k}) \leq \epsilon$, as required. ■

For comparable task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1$ is said to be a δ-implementation of $\mathcal{A}_2$ if the one-sided Hausdorff distance from $\mathsf{Tdists}_{\mathcal{A}_1}$ to $\mathsf{Tdists}_{\mathcal{A}_2}$ is at most δ.

Definition 8.9. Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, closed task-PIOAs. For $\delta > 0$, $\mathcal{A}_1$ is said to δ-implement $\mathcal{A}_2$, written as $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$, if for every $\mu_1 \in \mathsf{Tdists}_{\mathcal{A}_1}$ there exists $\mu_2 \in \mathsf{Tdists}_{\mathcal{A}_2}$ such that $\mathbf{d}_u(\mu_1, \mu_2) \leq \delta$. Closed task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be δ-*equivalent*, written as $\mathcal{A}_1 \cong_\delta \mathcal{A}_2$, if $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$ and $\mathcal{A}_2 \leq_\delta \mathcal{A}_1$.

Metrics over probability distributions have been a subject of intense research in probability theory (see, for example, the books [Rac91] and [Dud76]). Because of their applicability to probabilistic safety and termination proofs, in this thesis we use the uniform metric and the discounted version of the uniform metric (see Section 8.4), to define approximate implementations for task-PIOAs. In the following sections we develop simulation-based techniques for inductively proving these different kinds of approximate implementation relations. As we shall see in the next section, the soundness of our approximate simulations relies only weakly on the choice of the metric. In fact, with the appropriate changes in the definition of approximate simulations, it is sound for proving approximate implementations with respect to any metric satisfying Lemma 8.2.

8.3.2 Expanded Approximate Simulations

In this section, we develop a sound technique for proving uniform approximate implementation relations for task-PIOAs. This technique is analogous to the simulation relation-based technique for proving exact implementation relations, but we use a new kind of simulation called *Expanded Approximate Simulation (EAS)*. The definition of Expanded Approximate Simulations (EAS) relies on an *expansion* operation on real valued functions. This operation generalizes the notion of expansion of a relation used in Definition 8.7.

Definition 8.10. Let x be an element of the set $\mathcal{X}$ and $\{\lambda_i\}_{i\in I}$ be a countable sequence of numbers such that $\sum_{i\in I} \lambda_i = 1$. If there exists a sequence $\{x_i\}$ in $\mathcal{X}$ such that $x = \sum_{i\in I} \lambda_i x_i$, then x is a *convex combination* of the $\{x_i\}'s$. A function $\phi : \mathcal{X} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is *convex* if for every $x = \sum_{i\in I} \lambda_i x_i$, $\phi(x) \leq \sum_{i\in I} \lambda_i \phi(x_i)$. If equality holds then the function is said to be *distributive*.

Definition 8.11. Given a function $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, the *expansion* of ϕ, written as $\hat{\phi}$, is a function $\hat{\phi} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ defined as:

$$\hat{\phi}(x_1, y_1) \triangleq \min_{\substack{\psi \in \mathsf{Disc}(\mathcal{X}\times\mathcal{Y}) \\ x_1 = \sum_x \psi(x,y) x \\ y_1 = \sum_x \psi(x,y) y}} \left[\max_{(x,y)\in \mathsf{supp}(\psi)} \phi(x, y) \right] \tag{8.1}$$

The value of $\hat{\phi}$ is defined in terms of a minimization problem over all joint distributions over $\mathsf{Disc}(\mathcal{X} \times \mathcal{Y})$ that have first and second marginals with means equal to x_1 and y_1, respectively. The function that is minimized is the maximum value of ϕ over all points in

the support of ψ. The following lemma gives an alternative, but equivalent definition for expansion of a function.

Lemma 8.3. *Given a function $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, suppose we define $\bar{\phi} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ as follows: For any $\epsilon \geq 0$, $\bar{\phi}(x_1, y_1) \leq \epsilon$ if and only if there exists a joint distribution $\psi \in \mathsf{Disc}(\mathcal{X} \times \mathcal{Y})$ such that:*

$$\max_{x,y \in \mathsf{supp}(\psi)} \phi(x, y) \leq \epsilon \text{ such that} \tag{8.2}$$

$$x_1 = \sum_{x,y \in \mathsf{supp}(\psi)} \psi(x, y)x, \tag{8.3}$$

$$y_1 = \sum_{x,y \in \mathsf{supp}(\psi)} \psi(x, y)y. \tag{8.4}$$

Then, $\hat{\phi} = \bar{\phi}$.

Proof. Let us fix $x_1 \in \mathcal{X}$ and $y_1 \in \mathcal{Y}$. First we show that $\bar{\phi}(x_1, y_1) \leq \hat{\phi}(x_1, y_1)$. Suppose $\hat{\phi}(x_1, y_1) = \epsilon$, for some $\epsilon > 0$. From Definition 8.11, we know that there exists a joint distribution $\psi \in \mathsf{Disc}(\mathcal{X} \times \mathcal{Y})$, satisfying Equations (8.3) and (8.4). Further the maximal value of ϕ over any $x, y \in \mathsf{supp}\,\psi$, must be at most ϵ. Therefore, $\bar{\phi}(x_1, y_1) \leq \epsilon = \hat{\phi}(x_1, y_1)$

Next we show that $\bar{\phi}(x_1, y_1) \geq \hat{\phi}(x_1, y_1)$. Suppose $\bar{\phi}(x_1, y_1) = \epsilon$. From definition of $\bar{\phi}$, we know that there exists a joint distribution $\psi \in \mathsf{Disc}(\mathcal{X} \times \mathcal{Y})$, satisfying Equations (8.3) and (8.4), and from Equation (8.2) we know that the maximal value of ϕ over any $x, y \in \mathsf{supp}\,\psi$ is at most ϵ. Since $\hat{\phi}$ minimizes over all possible joint distributions, $\hat{\phi}(x_1, y_1) \leq \epsilon = \bar{\phi}(x_1, y_1)$. ■

The consistency requirements imposed by Equations (8.3) and (8.4) constrain the choice of ψ to those joint distributions for which the expected values of x and y coincide with x_1 and y_1. Given ϕ, we say that joint distribution ψ is *feasible* for x_1 and y_1 if it satisfies these consistency requirements. If ϵ is the smallest nonnegative real for which there exists a feasible ψ that also satisfies Equation (8.2), that is, $\max_{x,y \in \mathsf{supp}(\psi)} \phi(x, y) \leq \epsilon$, then we say that ψ is an *optimal distribution* for $\hat{\phi}(x_1, y_1) = \epsilon$. The next proposition is a straightforward consequence of Definition 8.11.

Proposition 8.4. For any $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, $\hat{\phi} \leq \phi$.

Proof. Let us fix $x_1 \in \mathcal{X}$ and $y_1 \in \mathcal{Y}$. Suppose $\phi(x_1, y_1) = \epsilon$ for some $\epsilon \geq 0, x_1 \in \mathcal{X}, y_1 \in \mathcal{Y}$. The joint distribution $\psi = \delta_{x_1, y_1}$ is a feasible distribution for x_1 and y_1. Since, for the only point $(x_1, y_1) \in \mathsf{supp}(\psi)$, $\phi(x_1, y_1) = \epsilon$, and $\hat{\phi}$ minimizes over all possible choices of feasible joint distributions, $\hat{\phi}(x_1, y_1) \leq \epsilon$. ■

This next lemma states a key property of the expansion operation for functions; it relates the sublevel sets of a given function ϕ and its expanded version $\hat{\phi}$.

Lemma 8.5. *$x_1 \in \mathcal{X}$, $y_1 \in \mathcal{Y}$. For any $\epsilon \geq 0$, at any point (x, y) in the convex hull of the ϵ-sublevel set of ϕ, $\hat{\phi} \leq \epsilon$.*

Proof. Let $L_{\phi \leq \epsilon}$ be the ϵ-sublevel set of ϕ, for some $\epsilon \geq 0$. That is, $L_{\phi \leq \epsilon} = \{(x, y) \mid x \in \mathcal{X}, y \in \mathcal{Y}, \phi(x, y) \leq \epsilon\}$. If $L_{\phi \leq \epsilon}$ is convex then the result follows immediately from Proposition 8.4. Suppose the sublevel set $L_{\phi \leq \epsilon}$ is not convex (the shaded region in Figure 8-1).

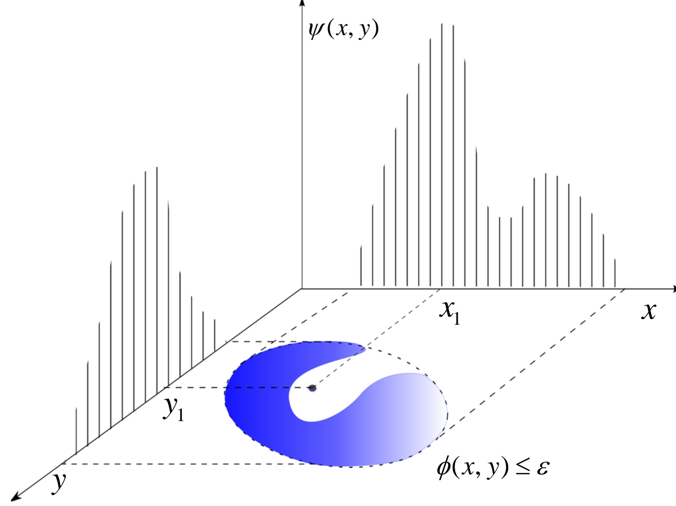

Figure 8-1: Marginal distributions of the optimal joint distribution ψ for $\hat{\phi}(x_1, y_1) = \epsilon$.

For any point (x_1, y_1) in the convex hull $L_{\phi\leq\epsilon}$, we can find a joint distribution ψ supported on $L_{\phi\leq\epsilon}$, such that the marginals of ψ (shown by vertical lines on the x and y axes) have mean x_1 and y_1, respectively. Such a ψ is an optimal distribution for $\hat{\psi}(x_1, y_1) \leq \epsilon$. Thus, for every point in the convex hull of $L_{\phi\leq\epsilon}$, $\hat{\phi} \leq \epsilon$. ■

Our new notion of approximate simulation for task-PIOAs is a function $\phi : \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2}) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ and the expansion of this function plays a key role in the definition of simulation. Informally, the simulation function ϕ gives a measure of similarity between two distributions over the execution fragments of two automata. If $\phi(\mu_1, \mu_2) \leq \epsilon$, then, first of all, it is possible to closely simulate from μ_2 anything that can happen from μ_1. Here closeness of simulation is measured with the $\mathbf{d}_u$ metric on the trace distributions. Secondly, if μ_1' and μ_2' are the distributions obtained by taking a step from μ_1 and μ_2, then μ_1' and μ_2' are also close in the sense that $\hat{\phi}(\mu_1', \mu_2') \leq \epsilon$.

Definition 8.12. Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are two comparable closed task-PIOAs, ϵ is a non-negative constant, and ϕ is a function $\mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2}) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. The function ϕ is an (ϵ, δ)-*expanded approximate simulation* from $\mathcal{A}_1$ to $\mathcal{A}_2$ if there exists a function $\mathsf{c} : R_1^* \times R_1 \to R_2^*$ such that the following properties hold:

1. *(Start condition)* $\phi(\bar{\nu}_1, \bar{\nu}_2) \leq \epsilon$.
2. *(Step condition)* If $\phi(\mu_1, \mu_2) \leq \epsilon$, $T \in R_1, \rho \in R_1^*$ and μ_1 is consistent with ρ, and μ_2 is consistent with $\mathsf{full}(\mathsf{c})(\rho)$, then $\hat{\phi}(\mu_1', \mu_2') \leq \epsilon$, where $\mu_1' = \mathsf{apply}(\mu_1, T)$ and $\mu_2' = \mathsf{apply}(\mu_2, c(\rho, T))$.
3. *(Trace condition)* If $\phi(\mu_1, \mu_2) \leq \epsilon$ then $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2)) \leq \delta$.

A function ϕ is an (ϵ, δ)-EAS if (i) the initial distributions $\bar{\nu}_1$ and $\bar{\nu}_2$ are ϵ-close in the sense of ϕ, (ii) if two distributions μ_1, μ_2 are ϵ-close in the sense of ϕ, then the distributions μ_1'—obtained by applying a new task to μ_1, and μ_2'—obtained by applying a corresponding sequence of tasks to μ_2, are also ϵ-close in the sense of $\hat{\phi}$, (iii) if μ_1, μ_2 are ϵ-close in the sense of ϕ, then the corresponding trace distributions are δ-close in the sense of $\mathbf{d}_u$. Example 8.1

shows a simple approximate simulation function. Approximate simulations are similar to simulation relations of Definition 8.7 with the relations R and $\mathcal{E}(R)$ replaced by the real valued simulation functions ϕ and $\hat{\phi}$. The following lemma relates expanded approximate simulation with the simulation relations of Definition 8.7.

Lemma 8.6. *Suppose ϕ is a (ϵ, δ)-expanded approximate simulation from $\mathcal{A}_1$ to $\mathcal{A}_2$. We define $R \subseteq \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2})$ as follows:*

$$\mu_1 \; R \; \mu_2 \iff \phi(\mu_1, \mu_2) = 0.$$

Then, R is a simulation relation from $\mathcal{A}_1$ to $\mathcal{A}_2$.

Proof. Since ϕ is a (ϵ, δ)-expanded approximate simulation, there exists $\mathsf{c} : ({R_1}^* \times R_1) \to {R_2}^*$ which satisfies all the conditions in Definition 8.12. We check that R satisfies the three required conditions.

Start condition. From the start condition of ϕ, $\phi(\mu_1, \mu_2) = 0$ which implies the $\mu_1 \; R \; \mu_2$.

Step condition. From the step condition of ϕ, we know that if $\phi(\mu_1, \mu_2) = 0$, $\rho \in {R_1}^*$, μ_1 is consistent with ρ, μ_2 is consistent with $\mathsf{full}(\mathsf{c})(\rho)$, and $T \in R_1$, then $\hat{\phi}(\mu_1', \mu_2') = 0$, where $\mu_1' = \mathsf{apply}(\mu_1, T)$ and $\mu_2' = \mathsf{apply}(\mu_2, \mathsf{c}(\rho, T))$.

From the definition of $\hat{\phi}$, there exists joint distribution ψ such that for every η_1, η_2 is the support of ψ, $\phi(\eta_1, \eta_2) = 0$, that is $\eta_1 \; R \; \eta_2$. Using this ψ as the weighting function it can be checked easily that $\mu_1' \; R \; \mu_2'$.

Trace condition. From the trace condition of ϕ, for every measurable set of traces C, $tdist(\mu_1)(C) = tdist(\mu_2)(C)$ which implies that $tdist(\mu_1) = tdist(\mu_2)$.

∎

One final remark on Definition 8.12 before we move on to prove its soundness. We can $\mathbf{d}_u$ with some other metric $\mathbf{d}$ on trace distributions, the definition can be naturally adapted to prove approximate implementations with respect to

8.3.3 Soundness of Expanded Approximate Simulations

This section culminates in Theorem 8.11 which states that (ϵ, δ)-expanded approximate simulations are sound with respect to δ-approximate implementations. First we prove key lemmas used in the proof of the theorem.

Lemma 8.7. *Suppose ϕ is a (ϵ, δ)-expanded approximate simulation from $\mathcal{A}_1$ to $\mathcal{A}_2$. For any $\mu_1 \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1})$ and $\mu_2 \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2})$, if $\hat{\phi}(\mu_1, \mu_2) \leq \epsilon$ then $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2)) \leq \delta$.*

Proof. Since $\hat{\phi}(\mu_1, \mu_2) \leq \epsilon$ we know that there exists a joint distribution ψ which is feasible for μ_1, μ_2, and for every $\eta_1, \eta_2 \in \mathsf{supp}(\psi)$, $\phi(\eta_1, \eta_2) \leq \epsilon$. So, for $i \in \{1, 2\}$, $\mu_i = \sum_{\eta_1, \eta_2 \in \mathsf{supp}(\psi)} \psi(\eta_1, \eta_2) \eta_i$ and from the trace condition of Definition 8.12 it follows that

$$tdist(\mu_i) = \sum_{\eta_1, \eta_2 \in \mathsf{supp}(\psi)} \psi(\eta_1, \eta_2) tdist(\eta_i).$$

We can then express $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2))$ as follows:

$$\begin{aligned}
& \sup_{C \in \mathcal{F}_{\mathsf{Traces}^*_{\mathcal{A}}}} |\ tdist(\mu_1)(C) - tdist(\mu_2)(C)| \\
= & \sup_{C \in \mathcal{F}_{\mathsf{Traces}^*_{\mathcal{A}}}} |\sum_{\eta_1,\eta_2} \psi(\eta_1,\eta_2) tdist(\eta_1)(C) - \sum_{\eta_1,\eta_2} \psi(\eta_1,\eta_2) tdist(\eta_2)(C)| \\
\leq & \sup_{C \in \mathcal{F}_{\mathsf{Traces}^*_{\mathcal{A}}}} \sum_{\eta_1,\eta_2} \psi(\eta_1,\eta_2) |(tdist(\eta_1)(C) - tdist(\eta_2)(C))|.
\end{aligned}$$

For any $\eta_1, \eta_2 \in \mathsf{supp}(\psi)$, $\phi(\eta_1,\eta_2) \leq \epsilon$ and since ϕ is an (ϵ,δ)-expanded approximate simulation, $\mathbf{d}_u(tdist(\eta_1), tdist(\eta_2)) \leq \delta$. From Definition 8.8, it follows that $|\ tdist(\eta_1)(C) - tdist(\eta_2)(C)| \leq \delta$. Therefore, we have $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2)) \leq \sum_{\eta_1,\eta_2} \psi(\eta_1,\eta_2)\delta \leq \delta$. ■

Lemma 8.8. *Suppose $\phi : \mathsf{Disc}(X_1) \times \mathsf{Disc}(X_2) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a function, $\mu_i \in \mathsf{Disc}(X_i)$ for $i \in \{1,2\}$, $\hat{\phi}(\mu_1,\mu_2) \leq \epsilon$ with optimal distribution ψ. Let $f_i : \mathsf{Disc}(X_i) \to \mathsf{Disc}(X_i)$ be distributive functions, for $i \in \{1,2\}$. If for each $\kappa_1, \kappa_2 \in \mathsf{supp}(\psi)$, $\hat{\phi}(f_1(\kappa_1), f_2(\kappa_2)) \leq \epsilon$, then $\hat{\phi}(f_1(\mu_1), f_2(\mu_2)) \leq \epsilon$.*

Proof: For each $\kappa_1, \kappa_2 \in \mathsf{supp}(\psi)$, let $\psi_{\kappa_1\kappa_2}$ be the optimal distribution for $\hat{\phi}(f_1(\kappa_1), f_2(\kappa_2)) = \epsilon$. We define a joint distribution ψ' on $\mathsf{Disc}(X_1) \times \mathsf{Disc}(X_2)$ as follows:

$$\psi' \triangleq \sum_{(\kappa_1,\kappa_2) \in \mathsf{supp}(\psi)} \psi(\kappa_1,\kappa_2) \psi_{\kappa_1,\kappa_2} \tag{8.5}$$

and show that ψ' is a feasible distribution for $f_1(\mu_1)$ and $f_2(\mu_2)$ and for any $\eta_1, \eta_2 \in \mathsf{supp}(\psi')$, $\phi(\eta_1,\eta_2) \leq \epsilon$.

1. For feasibility of ψ' we have to show that for $i \in \{1,2\}$, $f_i(\mu_i)$ equals:

$$\begin{aligned}
& \sum_{\eta_1 \in \mathsf{Disc}(X_1), \eta_2 \in \mathsf{Disc}(X_2)} \psi'(\eta_1,\eta_2)\eta_i \\
= & \sum_{\eta_1 \in \mathsf{Disc}(X_1), \eta_2 \in \mathsf{Disc}(X_2)} \left[\sum_{(\kappa_1,\kappa_2) \in \mathsf{supp}(\psi)} \psi(\kappa_1,\kappa_2) \psi_{\kappa_1,\kappa_2}(\eta_1,\eta_2) \right] \eta_i \\
= & \sum_{(\kappa_1,\kappa_2) \in \mathsf{supp}(\psi)} \psi(\kappa_1,\kappa_2) \left[\sum_{\eta_1 \in \mathsf{Disc}(X_1), \eta_2 \in \mathsf{Disc}(X_2)} \psi_{\kappa_1,\kappa_2}(\eta_1,\eta_2)\eta_i \right] \\
= & \sum_{(\kappa_1,\kappa_2) \in \mathsf{supp}(\psi)} \psi(\kappa_1,\kappa_2) f_i(\kappa_i) \quad \text{[from feasibility of } \psi_{\kappa_1,\kappa_2}] \\
= & f_i \left(\sum_{(\kappa_1,\kappa_2) \in \mathsf{supp}(\psi)} \psi(\kappa_1,\kappa_2)\kappa_i \right) \quad \text{[from distributivity of } f_i] \\
= & f_i(\mu_i) \quad \text{[from feasibility of } \psi].
\end{aligned}$$

2. For optimality of ψ' it suffices to show that for all $\eta_1, \eta_2 \in \mathsf{supp}(\psi')$, $\phi(\eta_1,\eta_2) \leq \epsilon$. If $\psi'(\eta_1,\eta_2) > 0$ then from Equation (8.5) it follows that there exists $\kappa_1, \kappa_2 \in$

$\mathsf{supp}(\psi)$ such that $\psi_{\kappa_1,\kappa_2}(\eta_1,\eta_2) > 0$. Since ψ_{κ_1,κ_2} is a optimal distribution for $\hat{\phi}(f_1(\kappa_1), f_2(\kappa_2)) = \epsilon$, from its optimality we know that for any $\nu_1, \nu_2 \in \mathsf{supp}(\psi_{\kappa_1,\kappa_2})$, $\phi(\nu_1,\nu_2) \leq \epsilon$. In particular, $\eta_1, \eta_2 \in \mathsf{supp}(\psi_{\kappa_1,\kappa_2})$ and so we have $\phi(\eta_1,\eta_2) \leq \epsilon$.

Lemma 8.9. *Let $\{\mu_i\}_{i\in I}$ be a countable family of discrete probability measures $\mu_i \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$ and let $\mu = \sum_{i\in I} \lambda_i \mu_i$ be a convex combination of $\{\mu_i\}$, where $\sum_{i\in I} \lambda_i = 1$. Let T be task of $\mathcal{A}$. Then $\mathsf{apply}(\mu, T) = \sum_{i\in I} \lambda_i \,\mathsf{apply}(\mu_i, T)$.*

Proof. Suppose p_1 and p_2 are the functions used in the definition of $\mathsf{apply}(\mu, T)$, and suppose for each $i \in I$, p^i_1 and p^i_2 be the functions used in the definition of $\mathsf{apply}(\mu_i, T)$. Fix a finite execution fragment α. We show that $p_1(\alpha) = \sum_i \lambda_i p^i_1(\alpha)$ and $p_2(\alpha) = \sum_i \lambda_i p^i_2(\alpha)$, from which it follows that $\mathsf{apply}(\mu, T)(\alpha) = p_1(\alpha) + p_2(\alpha) = \sum_i \lambda_i (p^i_1(\alpha) + p^i_3(\alpha)) = \sum_i \lambda_i \,\mathsf{apply}(\mu_i, T)$.

To prove that $p_1(\alpha) = \sum_i \lambda_i p^i_1(\alpha)$, we consider two cases. If $\alpha = \alpha' a q$ where $\alpha' \in \mathsf{supp}(\mu)$, $a \in T$, and $\alpha'.lstate, a, \eta) \in \mathcal{D}$, then, by Definition 8.3 $p_1(\alpha) = \mu(\alpha')\eta(q)$ and for each $i \in I$, $p^i_1(\alpha) = \mu_i(\alpha')\eta(q)$. Thus, $p_1(\alpha) = \sum_i \lambda_i p^i_1(\alpha)$. Otherwise, again by Definition 8.3 $p_1(\alpha) = 0$ and for each $i \in I$, $p^i_1(\alpha) = 0$, and the result holds trivially.

To prove that $p_2(\alpha) = \sum_i \lambda_i p^i_2(\alpha)$, we consider two cases. If T is not enabled in $\alpha.lstate$ then, by Definition 8.3, $p_2(\alpha) = \mu(\alpha)$, and for each $i \in I$, $p^i_2(\alpha) = \mu_i(\alpha)$. Thus, $p_2(\alpha) = \sum_i \lambda_i p^i_2(\alpha)$. Otherwise, again by Definition 8.3 $p_2(\alpha) = 0$ and for each $i \in I$, $p^i_2(\alpha) = 0$, and the result holds trivially. ∎

Proposition 8.10. Let $\{\mu_i\}_{i\in I}$ be a countable family of discrete probability measures $\mu_i \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$ and let $\mu = \sum_{i\in I} \lambda_i \mu_i$ be a convex combination of $\{\mu_i\}$, where $\sum_{i\in I} \lambda_i = 1$. Let ρ be a finite sequence of tasks. Then $\mathsf{apply}(\mu, \rho) = \sum_{i\in I} \lambda_i \,\mathsf{apply}(\mu_i, \rho)$.

Proof. The proof is by induction on the length of ρ. If ρ is the empty sequence, then for any $\eta \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$, $\mathsf{apply}(\eta, \rho) = \eta$ and it follows that $\mu = \sum_{i\in I} \lambda_i \mu_i = \sum_{i\in I} \lambda_i \,\mathsf{apply}(\mu_i, \rho)$. For the induction step, let $\rho = \rho' T$. By Definition 8.3, $\mathsf{apply}(\mu, \rho' T) = \mathsf{apply}(\mathsf{apply}(\mu, \rho'), T)$. By the induction hypothesis, $\mathsf{apply}(\mu, \rho') = \sum_i \lambda_i \,\mathsf{apply}(\mu_i, \rho')$ and thus, $\mathsf{apply}(\mu, \rho' T) = \mathsf{apply}(\sum_i \lambda_i \,\mathsf{apply}(\mu_i, \rho'), T)$. For each $i \in I$, $\mathsf{apply}(\mu_i, \rho')$ is a discrete probability measure in $\mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$. By Lemma 8.9, $\mathsf{apply}(\sum_i \lambda_i \,\mathsf{apply}(\mu_i, \rho'), T) = \sum_i \lambda_i \,\mathsf{apply}(\mathsf{apply}(\mu_i, \rho'), T)$. Using Definition 8.3 it follows that $\mathsf{apply}(\mu, \rho' T) = \sum_i \lambda_i \,\mathsf{apply}(\mu_i, \rho' T)$ as required. ∎

Theorem 8.11. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two closed comparable task-PIOAs. If there exists a (ϵ, δ)-expanded approximate simulation function from $\mathcal{A}_1$ to $\mathcal{A}_2$ then $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$.*

Proof. Let ϕ be the assumed (ϵ, δ)-expanded approximate simulation function from $\mathcal{A}_1$ to $\mathcal{A}_2$. Let μ_1 be the probabilistic execution of $\mathcal{A}_1$ generated by the starting distribution $\bar{\nu}_1$ and a (finite or infinite) task schedule $T_1, T_2, \ldots$. For each $i > 0$, we define ρ_i to be $\mathsf{c}(T_1 \ldots T_{i-1}, T_i)$. Let μ_2 be the probabilistic execution of $\mathcal{A}_2$ generated by $\bar{\nu}_2$ and the concatenation $\rho_1, \rho_2, \ldots$. It suffices to show that: $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2)) \leq \delta$.

For each $j \geq 0$, let us define $\mu_{1,j} \triangleq \mathsf{apply}(\bar{\nu}_1, T_1, \ldots, T_j)$ and $\mu_{2,j} \triangleq \mathsf{apply}(\bar{\nu}_2, \rho_1, \ldots, \rho_j)$. For $i \in \{1, 2\}$ and for each $j \geq 0$, $\mu_{i,j} \leq \mu_{i,j+1}$ and $\lim_{j\to\infty} \mu_{i,j} = \mu_i$. (the above uses Lemma 8.20 of Appendix 8.8). Observe that for every $j \geq 0$, $\mu_{1,j+1} = \mathsf{apply}(\mu_{1,j}, T_{j+1})$ and also that $\mu_{2,j+1} = \mathsf{apply}(\mu_{2,j}, \rho_{j+1})$.

Step 1. We prove by induction that for all $j \geq 0$, $\hat{\phi}(\mu_{1,j}, \mu_{2,j}) \leq \epsilon$.

Step 1a. For $j = 0$, $\mu_{1,0} = \bar{\nu}_1$ and $\mu_{2,0} = \bar{\nu}_2$. By the start condition of the simulation function, $\phi(\mu_{1,0}, \mu_{2,0}) \leq \epsilon$ and therefore by Proposition 8.4 $\hat{\phi}(\mu_{1,0}, \mu_{2,0}) \leq \epsilon$.

Step 1b. For the inductive step, we assume that $\hat{\phi}(\mu_{1,j}, \mu_{1,j}) \leq \epsilon$ and show that $\hat{\phi}(\mu_{1,j+1}, \mu_{1,j+1}) \leq \epsilon$. First of all, note that $\mu_{1,j+1} = \mathsf{apply}(\mu_{1,j}, T_{j+1})$ and $\mu_{2,j+1} = \mathsf{apply}(\mu_{2,j}, \mathsf{c}(\rho_j T_{j+1}))$. For $i \in \{1,2\}$, let us define $f_i : \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_i}) \rightarrow \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_i})$ as $f_1(\eta) \triangleq \mathsf{apply}(\eta, T_{j+1})$ and $f_2(\eta) \triangleq \mathsf{apply}(\eta, \mathsf{c}(\rho_j T_{j+1}))$. If we can apply Lemma 8.8, to the functions f_1 and f_2 then it follows that $\hat{\phi}(f_1(\mu_{1,j}), f_2(\mu_{2,j})) \leq \epsilon$ as required.

Step 1c. It remains to check that these two functions satisfy all the conditions in the hypothesis of Lemma 8.8. Distributivity of f_1 and f_2 follow from Proposition 8.10 (see Appendix B). Suppose $\hat{\phi}(\mu_{1,j}, \mu_{1,j}) \leq \epsilon$ with optimal distribution ψ, and suppose $\eta_1, \eta_2 \in \mathsf{supp}(\psi)$, we have to show that $\hat{\phi}(f_1(\eta_1), f_2(\eta_2)) \leq \epsilon$. Since $\eta_1, \eta_2 \in \mathsf{supp}(\psi)$, from optimality of ψ, we know that $\phi(\eta_1, \eta_2) \leq \epsilon$. Observe that for $i \in \{1,2\}$, $\mathsf{supp}(\eta_i) \subseteq \mathsf{supp}(\mu_{i,j})$, and thus η_1 is consistent with T_{j+1} and η_2 is consistent with $\mathsf{c}(\rho_j T_{j+1})$. Therefore, by the step condition on ϕ, $\hat{\phi}(\mathsf{apply}(\eta_1, T_{j+1}), \mathsf{apply}(\eta_2, \mathsf{c}(\rho_j T_{j+1}))) \leq \epsilon$. Since $f_1(\eta_1) = \mathsf{apply}(\eta_1, T_{j+1})$ and $f_2(\eta_2) = \mathsf{apply}(\eta_2, \mathsf{c}(\rho_j T_{j+1}))$, we have $\hat{\phi}(f_1(\mu_{1,j}), f_2(\mu_{2,j})) \leq \epsilon$, as required in the hypothesis of Lemma 8.8.

Step 2. From Lemma 8.7, for each $j \geq 0$, $\mathbf{d}_u(tdist\mu_{1,j}, tdist\mu_{2,j}) \leq \delta$. From Lemma 8.18 of Appendix 8.8 we know that for $i \in \{1,2\}$, $\lim_{j\to\infty} tdist(\mu_{i,j}) = tdist(\mu_i)$. From Lemma 8.2 we conclude that $\mathbf{d}_u(tdist(\mu_1), tdist(\mu_2)) =$
$\lim_{j\to\infty} \mathbf{d}_u(tdist(\mu_{1,j}), tdist(\mu_{2,j})) \leq \delta$.

■

The following simple example illustrates an application of (ϵ, δ)-expanded approximate simulations.

Example 8.1. Consider two task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$: For $i = \{1,2\}$, $X_i = \{x\}$, with $Q_i = type(x) = \{s_0, s_1, s_2\}$; $A_i = H_i = \{\mathsf{a}\}$; $\mathcal{R}_i = \{\mathsf{a}\}$. The initial distributions and the discrete probabilistic transitions for the two automata are defined as follows.

$$\bar{\nu}_1 = \begin{bmatrix}\frac{1}{10} & \frac{9}{10} & 0\end{bmatrix}, \quad \bar{\nu}_2 = \begin{bmatrix}\frac{9}{10} & \frac{1}{10} & 0\end{bmatrix}, \quad P_1 = \begin{bmatrix} 0 & \frac{9}{10} & \frac{1}{10} \\ \frac{2}{10} & \frac{8}{10} & 0 \\ \frac{1}{10} & \frac{9}{10} & 0 \end{bmatrix}, and \quad P_2 = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{3}{10} & \frac{7}{10} & 0 \\ \frac{3}{10} & \frac{2}{5} & \frac{3}{10} \end{bmatrix}.$$

The row vector $\bar{\nu}_i$ gives the initial distribution over the three states of automaton $\mathcal{A}_i$. The kth row of the transition matrix P_i, defines the probability distribution over states of $\mathcal{A}_i$ that arises when action a occurs at state s_k. For instance, the first row of P_1 indicates that in automaton $\mathcal{A}_1$, $s_0 \xrightarrow{a} \mu$, where μ is the discrete distribution $\begin{bmatrix}0 \ \frac{9}{10} \ \frac{1}{10}\end{bmatrix}$. Note the $\mathcal{A}_1$ and $\mathcal{A}_2$ are purely probabilistic automata, without any nondeterminism. In fact they are Markov chains (with all transitions labeled by the action a). The only interesting task schedules for these automata are finite or infinite sequences of a's. Let us define a function $\phi : \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2}) \rightarrow \mathbb{R}_{\geq 0} \cup \{\infty\}$, as

$$\phi(\mu_1, \mu_2) \triangleq \sum_i |lstate(\mu_1)(s_i) - lstate(\mu_2)(s_i)|$$

Recall that $lstate(\mu_1)$ denotes the lstate distribution corresponding to the distribution μ_1. Informally, ϕ measures the L_1 distance between the distribution over the states of $\mathcal{A}_1$ and $\mathcal{A}_2$. It is routine to check analytically that ϕ satisfies the start condition and the step condition of Definition 8.12 for any $\epsilon \geq \phi(\bar{\nu}_1, \bar{\nu}_2) = 1.6$ (the trace condition holds trivially). Figure 8-2 confirms this; it shows the $lstate$ distributions of μ_{1k} and μ_{2k} and the values of $\phi(\mu_{1k}, \mu_{2k})$, where $\mu_{ik} = \mathsf{apply}(\bar{\nu}_i, \mathsf{a}^k)$.

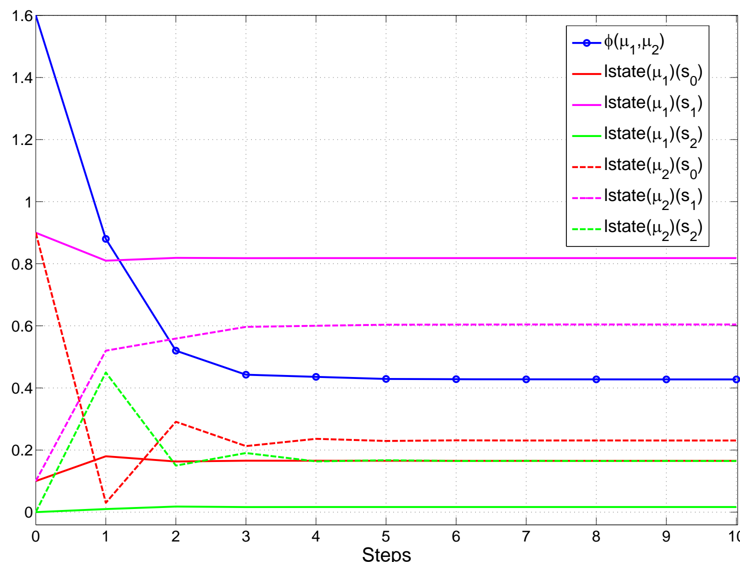

Figure 8-2: Discrepancy and $lstate$ distributions for $\mathcal{A}_1$ and $\mathcal{A}_2$.

8.3.4 Need for Expansion

In the step condition in the definition of EAS (Definition 8.12) it is required that if $\phi(\mu_1, \mu_2) \leq \epsilon$ then $\hat{\phi}(\mu_1', \mu_2') \leq \epsilon$. Indeed, if we replace this condition with the stronger condition—if $\phi(\mu_1, \mu_2) \leq \epsilon$ then $\phi(\mu_1', \mu_2') \leq \epsilon$—the resulting approximate simulation functions that we would obtain would be sound for proving approximate implementations. In fact this would have been sufficient for proving approximate implementation in Example 8.1. However, such *non-expanded* approximate simulation functions are considerably less powerful than EASs.

The key motivation for generalizing simulation relations to their current expanded form, first came from the verification of the Oblivious Transfer protocol in [CCK+06d]. In what follows, we describe an example from [CCK+06a], adapted to our setting, to illustrate that allowing $\hat{\phi}$ in the definition of EAS gives it more power.

Example 8.2. Consider the Rand automaton of Figure 8-3 *(left)*. When the choose action occurs at state r_0, it assigns a number between 1 and n to the variable z, where n is a even. The initial value of z is $\perp$ and the choice is made uniformly at random. The Trapdoor automaton assigns a random number between 1 and n to the variable y, but in this case the probabilities are slightly different. The first $\frac{n}{2}$ numbers are chosen with probability $\frac{1}{n} - \epsilon$ and the remaining are chosen with probability $\frac{1}{n} + \epsilon$, where $\epsilon \ll \frac{2}{n}$. The comp action of Trapdoor then applies a known permutation (e.g., if $y = n-1$ then $z = n$, else $z = (y+1)$

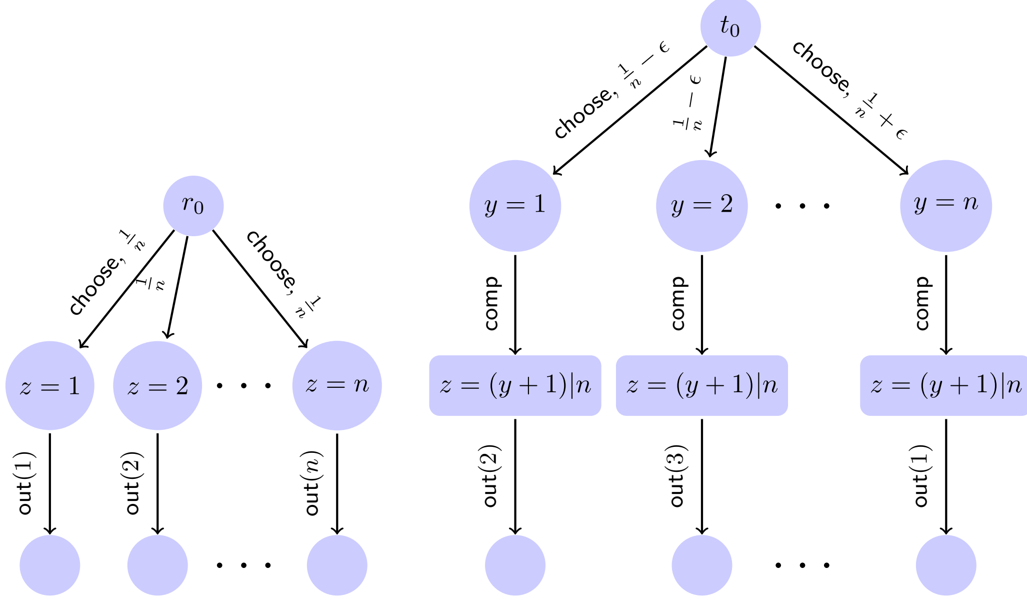

Figure 8-3: Rand and Trapdoor automata.

mod n) to y and assigns the permuted number to z. In both automata, the value of z is produced by an external out action.

We would like to use an approximate simulation to show that $\mathsf{Tdists}_{\mathsf{Trapdoor}}$ approximately implements $\mathsf{Tdists}_{\mathsf{Rand}}$. In doing so, it is natural to allow the steps that define z to correspond in the two automata. This means that the choose steps of Trapdoor which define y correspond to no step of Rand. We present an approximate simulation function that "ought to work" for this example. The function $\phi : \mathsf{Disc}(\mathsf{Frags}^*_{\mathsf{Trapdoor}}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathsf{Rand}}) \to \mathbb{R}_{\geq 0}$ is defined as

$$\phi(\mu_1, \mu_2) \triangleq \max_{\substack{s \in Q_{\mathsf{Trapdoor}} \\ u \in Q_{\mathsf{Rand}} \\ u.z \neq s.z}} [\eta_1(s) + \eta_2(u)] \qquad \text{if } \exists\ s \in \mathsf{supp}(\eta_1), u \in \mathsf{supp}(\eta_2), s.z \neq u.z,$$

$$\phi(\mu_1, \mu_2) \triangleq \max_{x \in \{1,\ldots,n\} \cup \{\bot\}} \left| \sum_{\substack{s \in Q_{\mathsf{Trapdoor}} \\ s.z = x}} \eta_1(s) - \sum_{\substack{u \in Q_{\mathsf{Rand}} \\ u.z = x}} \eta_2(u) \right| \qquad \text{otherwise,}$$

where $\eta_1 = lstate(\mu_1)$ and $\eta_2 = lstate(\mu_2)$. Informally, states corresponding to different values of z produce completely different outputs, and thus they should be relatively unrelated. The first condition assigns a large value to ϕ when there are states in the support of the two lstate distributions that have different values of z. This is captured by the first condition in the definition of ϕ, which returns a large value, namely, the sum of probabilities of two mismatched states, as the discrepancy between the corresponding distributions. The second condition is satisfied for η_1 and η_2 supported on states that have the same value of z; it measures the discrepancy between μ_1 and μ_2 as the maximum difference between the probability of states that correspond to any particular value of z including $\bot$. The summation is relevant for the distribution over states where $z = \bot$ and $y \in \{1, \ldots, n\}$ in the case of Trapdoor. The second summation could be simplified to $\eta_2(u)$, where u is the

unique state of Rand at which $u.z = x$.

Now we attempt to show that ϕ is a $(2\epsilon, 2\epsilon)$-simulation from $\mathsf{Trapdoor}$ to Rand without using the expanded version $\hat{\phi}$ of ϕ. The task correspondence mapping $\mathsf{c} : \mathcal{R}^*_{\mathsf{Trapdoor}} \times \mathcal{R}_{\mathsf{Trapdoor}} \to \mathcal{R}^*_{\mathsf{Rand}}$ as follows:

- $\mathsf{c}(\rho, \mathsf{choose}) = \lambda$,
- If ρ contains choose then $\mathsf{c}(\rho, \mathsf{comp}) = \mathsf{choose}$, otherwise $\mathsf{c}(\rho, \mathsf{comp}) = \lambda$,
- $\mathsf{c}(\rho, \mathsf{out}) = \mathsf{out}$

We check the start condition and the step condition for the interesting distributions that Rand and $\mathsf{Trapdoor}$ give rise to; the trace conditions can be checked easily from the discrepancy between the probabilistic executions.

Let $\mu_{11} = \mathsf{apply}(\delta_{t_0}, \mathsf{choose})$ and $\mu_{21} = \mathsf{apply}(\delta_{r_0}, \lambda) = \delta_{r_0}$. For all $s \in \mathsf{supp}(lstate(\mu_{11}))$, $s.z = t_0.z = \bot$, and all $u \in \mathsf{supp}(lstate(\mu_{21}))$, $u.z = r_0.z = \bot$. Hence by the second condition in the definition of ϕ, $\phi(\mu_{11}, \mu_{21}) = 0 \leq \epsilon$.

Let $\mu_{12} = \mathsf{apply}(\mu_{11}, \mathsf{choose})$ and $\mu_{22} = \mathsf{apply}(\mu_{21}, \lambda) = \mu_{21} = \delta_{r_0}$. For all $s \in \mathsf{supp}(lstate(\mu_{12}))$, $s.y \in \{1, \ldots, n\}$ and $s.z$ is still $\bot$. And for all $u \in \mathsf{supp}(lstate(\mu_{22}))$, $u.z = \bot$. Hence, again, the second condition in the definition of ϕ applies, and

$$\phi(\mu_{12}, \mu_{22}) = \left| \left(\frac{1}{n} + \epsilon + \frac{1}{n} + \epsilon + \ldots + \frac{1}{n} - \epsilon + \frac{1}{n} - \epsilon \right) - 1 \right| = 0 \leq \epsilon.$$

Next, let $\mu_{13} = \mathsf{apply}(\mu_{12}, \mathsf{comp})$ and $\mu_{23} = \mathsf{apply}(\mu_{22}, \mathsf{choose})$. Then, there exists $s \in \mathsf{supp}(\mu_{13})$ and $u \in \mathsf{supp}(\mu_{23})$, such that $s.z \neq u.z$, and by the first condition, $\phi(\mu_{12}, \mu_{22}) = \frac{2}{n} + \epsilon > 2\epsilon$. Therefore, ϕ is not a $(2\epsilon, 2\epsilon)$-approximate (unexpanded) simulation, and we cannot use ϕ to prove that $\mathsf{Trapdoor}$ is a good approximate implementation for Rand, at least not without using the expanded version of ϕ.

	$\delta_{y=1,z=2}$	$\delta_{y=2,z=3}$	$\delta_{y=3,z=4}$	$\delta_{y=4,z=1}$
$\delta_{z=1}$				$\frac{1}{4}$
$\delta_{z=2}$	$\frac{1}{4} - \epsilon$			ϵ
$\delta_{z=3}$		$\frac{1}{4} - \epsilon$	ϵ	
$\delta_{z=4}$			$\frac{1}{4}$	

Table 8.1: Optimal joint distribution ψ.

We show that ϕ *can* be used as an expanded approximate simulation function. In particular, we show that $\hat{\phi}(\mu_{13}, \mu_{23}) \leq 2\epsilon$, and we will use the witnessing joint distribution shown in the table of Table 8.2. Notice that the marginal distributions of ψ match with μ_{13} and μ_{23}. Further, for any ν_1, ν_2 in the support of ψ, ν_1 and ν_2 have the following properties: either (i) they are Dirac masses at states that have the same value of z, in which case $\phi(\nu_1, \nu_2) = \epsilon$ from the second condition in the definition of ϕ, otherwise (ii) for any $s \in Q_{\mathsf{Trapdoor}}$ and $u \in Q_{\mathsf{Rand}}$, $\nu_1(s) \leq \epsilon$ and $\nu_2(u) \leq \epsilon$, and therefore by the first condition $\phi(\nu_1, \nu_2) \leq 2\epsilon$. It follows that $\hat{\phi}(\mu_{13}, \mu_{23}) \leq 2\epsilon$.

By a similar argument it can be checked that $\hat{\phi}(\mu_{14}, \mu_{24}) \leq 2\epsilon$, where $\mu_{14} = \mathsf{apply}(\mu_{13}, \mathsf{out})$ and $\mu_{24} = \mathsf{apply}(\mu_{23}, \mathsf{out})$. This covers all the interesting cases and establishes that ϕ is a $(2\epsilon, 2\epsilon)$-expanded approximate simulation from Trapdoor to Rand.

8.3.5 Probabilistic Safety

In this subsection, we show how approximate implementations can be used for probabilistically reasoning about properties of trace distributions. Let $(\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}})$ be the measurable space of traces containing the traces of comparable task-PIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$, and $f : (\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}}) \rightarrow (Y, \mathcal{F}_Y)$ be a measurable function. For example, if $Y = \{0, 1\}$, then f could be a predicate on traces that we are interested in proving. For instance, we can define $f(\beta)$ to be 0 if a certain undesirable external action a occurs in β, and 1 otherwise. In general, f defines a property of traces that we are interested in verifying. Since f is a measurable function, it is actually a Y-valued random variable on Traces. We use the customary notation $[f = y] \triangleq \{\beta \in \mathsf{Traces} \mid f(\beta) = y\}$, for any $y \in Y$.

Lemma 8.12. *Let f be a measurable function (random variable) from $(\mathsf{Traces}, \mathcal{F}_{\mathsf{Traces}})$ to $(Y, \mathcal{F}_Y)$. Suppose $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$ and there exists $0 \leq p \leq 1$ such that for every $\mu_2 \in tdists(\mathcal{A}_2)$, $\mu_2([f = y]) \leq p$, for some $y \in Y$. Then, for all $\mu_1 \in tdist(\mathcal{A}_1)$, $\mu_1([f = y]) \leq \delta + p$.*

Proof. Fix $\mu_1 \in \mathsf{Tdists}_{\mathcal{A}_1}$. Since $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$, from Definition 8.9, we know that there exists $\mu_2 \in tdists(\mathcal{A}_2)$, such that $\mathbf{d}_u(\mu_1, \mu_2) \leq \delta$. We know that $\sup_{C \in \mathcal{F}_{\mathsf{Traces}}} |\mu_2(C) - \mu_1(C)| \leq \delta$. In particular, $|\mu_1([f = y]) - \mu_2([f = y])| \leq \delta$. As $\mu_2([f = y]) \leq p$, we have $\mu_1([f = y]) \leq p + \delta$ as required. ∎

Suppose that $\mathcal{A}_2$ triggers the "bad" action a with probability at most p. Formally, this means that the probability of any trace containing a, according to any trace distribution of $\mathcal{A}_2$, is at most p. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable closed task-PIOAs such that $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$, then from the Lemma 8.12 we can conclude that $\mathcal{A}_1$ triggers action a with probability at most $p + \delta$.

Violation of safety properties can also modeled as trace functions. Assume that violation of some safety property S is indicated by the occurrence of one of the external actions from the set $U \subseteq E_1 = E_2$. We define the function $f : \mathsf{Traces} \rightarrow \{0, 1\}$ as $f(\beta) \triangleq 0$ if some action from U occurs in the trace β, otherwise $f(\beta) \triangleq 1$. It can be easily checked that f is a measurable function and therefore is a boolean valued random variable. The event $[f = 0]$ corresponds to the set of traces in which S is violated. Now, if we know that in any trace distribution of $\mathcal{A}_2$ the probability of any U occurring is at most p and that $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$, then from Lemma 8.12 we can conclude that in any trace distribution of $\mathcal{A}_1$ the probability of occurrence of U, and hence the violation of S, is at most $\delta + p$.

8.4 Discounted Uniform Approximate Implementation

In the preceding section we defined uniform approximate implementation for PIOAs and proved that expanded approximate simulations are sound for proving this implementation relationship. We also demonstrated that uniform approximate implementations are suitable for probabilistically reasoning about certain classes of properties, like safety properties, where it is sufficient to quantify the absolute discrepancy in the trace distributions over all sets of traces. For certain other classes of properties the uniform metric is not suitable,

because the worst case discrepancy over all sets of traces does not convey useful information. We illustrate this with the following example.

Example 8.3. This example is the untimed version of Example 7.3 and models Ben-Or's randomized consensus protocol [BO83] for n fault-prone processors to agree on a valid value. The algorithm proceeds in a sequence of stages in each of which nonfaulty processes send and receive messages based on coin-flips and comparison of values. If the processes have access to perfectly random coins, then with some probability p, a stage ends successfully and all nonfaulty processes agree on a value, and after one communication round of a successful stage the consensus value is disseminated. An unsuccessful stage is followed by the beginning of the next stage.

The automaton in Figure 8-4 captures the termination behavior of the algorithm. The protocol starts in state s_{10} (corresponds to the states with $phase = 0$, $stage = 1$ of automaton Consensus of Example 7.3). The starting state for the successive stages are the states $s_{20}, s_{30}, \ldots$. If the the ith stage completes successfully then state s_{i1} is reached. The action try models the computation and communication within a stage. From stage s_{i0}, with probability p it leads to $s_{(i+1)0}$, the next stage, and with probability $1-p$ it leads to s_{i1}. The action decide marks the termination of the protocol and it takes s_{i1} to s_{i2} with probability 1.

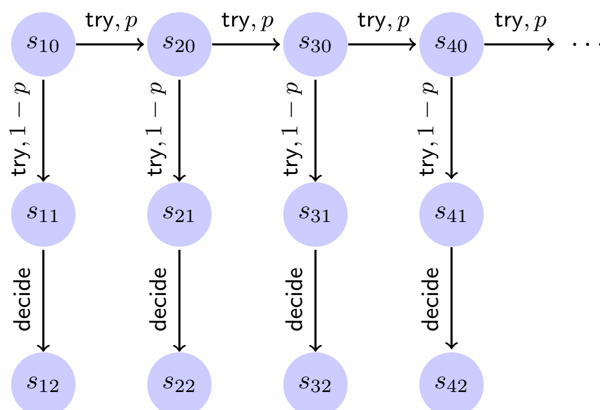

Figure 8-4: Automata representing Ben-Or consensus protocol.

Suppose PIOA $\mathcal{A}_1$ is an instance of the automaton in Figure 8-4 with perfect random coins, that is, $p = 1 - \frac{1}{2^n}$ and $1 - p = \frac{1}{2^n}$. Let $\mathcal{A}_2$ be a task-PIOA instance of the same automaton with slightly biased coins. We model the transition probabilities for $\mathcal{A}_2$ by $p + \epsilon$ and $1-p-\epsilon$, for a small positive ϵ. We would like to compare the probabilities of termination of $\mathcal{A}_1$ and $\mathcal{A}_2$ after a certain number of rounds, say k. With the uniform approximate implementation, we can show that the difference in these probabilities is less than δ, for a fixed $\delta > 0$. However, if individual probabilities of termination are themselves less than δ then this δ-approximation is too coarse and does not give us any useful information.

8.4.1 Discounted Uniform Metric on Traces

In the remainder of this section, we show how a discounted version of the uniform metric can be used to make more finer-grained comparison of probabilities of traces.

Definition 8.13. A probability distribution μ on execution fragments of $\mathcal{A}$ is said to be

finite if $\mathsf{Frags}^*_{\mathcal{A}}$ is a support for μ. A trace distribution μ of $\mathcal{A}$ is finite if $\mathsf{Traces}^*_{\mathcal{A}}$ is a support for μ.

Since any set of finite execution fragments is measurable, any finite probability distribution on execution fragments of $\mathcal{A}$ can also be viewed as a discrete probability measure on $\mathsf{Frags}^*_{\mathcal{A}}$. Likewise, a finite trace distribution can be viewed as a discrete distribution over $\mathsf{Traces}^*_{\mathcal{A}}$. In this section, we consider task-PIOAs with finite (trace) distributions and will treat these distributions as discrete distributions on execution fragments or traces.

Definition 8.14. For any $k \in \mathbb{N}$, the kth uniform metric is a function $\mathbf{d}_k : \mathsf{Disc}(\mathsf{Traces}_{\mathcal{A}}) \times \mathsf{Disc}(\mathsf{Traces}_{\mathcal{A}}) \rightarrow \mathbb{R}_{\geq 0}$ defined as:

$$\mathbf{d}_k(\mu_1, \mu_2) \triangleq \max_{\beta \in E^*, |\beta| = k} |\mu_1(\beta) - \mu_2(\beta)|.$$

The kth uniform metric measures the discrepancy between two trace distributions by looking at traces of length k only. We show that $\mathbf{d}_k$ is a pseudometric and that it satisfies the convergence property stated in Lemma 8.2.

Proposition 8.13. For all $k \in \mathbb{N}$, d_k is a pseudometric.

Proof. The symmetry property is easy to check. We prove that $\mathbf{d}_k$ satisfies the triangle inequality. Let μ_1, μ_2, μ_3 be distributions on E^*. $\mathbf{d}_k(\mu_1, \mu_3) = \max_{\beta \in E^*, |\beta| = k} |\mu_1(\beta) - \mu_3(\beta)|$. Suppose β_3 is a trace that realizes the supremum.

$$\begin{aligned}
|\mu_1(\beta_3) - \mu_3(\beta_3)| &\leq |\mu_1(\beta_3) - \mu_2(\beta_3)| + |\mu_2(\beta_3) - \mu_3(\beta_3)| \\
\mathbf{d}_k(\mu_1, \mu_3) &\leq \max_{\beta, |\beta| = k} |\mu_1(\beta) - \mu_2(\beta)| + \max_{\beta, |\beta| = k} |\mu_2(\beta) - \mu_3(\beta)|, \\
&\leq \mathbf{d}_k(\mu_1, \mu_2) + \mathbf{d}_k(\mu_2, \mu_3).
\end{aligned}$$

■

Lemma 8.14. *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are closed task-PIOAs. For $i \in \{1, 2\}$, let $\{\mu_{ij}\}_{j \in \mathbb{N}}$ be a chain of discrete probability distributions on the traces of $\mathcal{A}_i$ and let $\lim_{j \rightarrow \infty} \mu_{ij} = \mu_i$. Then for any $k \in \mathbb{N}$, $\lim_{j \rightarrow \infty} \mathbf{d}_k(\mu_{1j}, \mu_{2j}) = \mathbf{d}_k(\mu_1, \mu_2)$.*

Proof. The proof is the same as the proof of Lemma 8.2 for the $\mathbf{d}_u$ metric. ■

Given a family of positive discount factors $\{\delta_k\}_{k \in \mathbb{N}}$, we say that task-PIOA $\mathcal{A}_1$ δ_k-implements $\mathcal{A}_2$, if one-sided Hausdorff distance from $\mathsf{Tdists}_{\mathcal{A}_1}$ to $\mathsf{Tdists}_{\mathcal{A}_2}$ is at most δ_k, for every $k \in \mathbb{N}$.

Definition 8.15. Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, closed task-PIOAs and $\{\delta_k\}_{k \in \mathbb{N}}$ is a collection of positive real numbers, called *discount factors*. If for every trace distribution μ_1 *in*$\mathsf{Tdists}_{\mathcal{A}_1}$ there exists a trace distribution $\mu_2 \in \mathsf{Tdists}_{\mathcal{A}_2}$ such that for every $k \in \mathbb{N}$, $\mathbf{d}_k(\mu_1, \mu_2) \leq \delta_k$, then we say that $\mathcal{A}_1$ δ_k*-implements* $\mathcal{A}_2$ and write this as $\mathcal{A}_1 \leq_{\delta_k} \mathcal{A}_2$. $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be δ_k*-equivalent*, written as $\mathcal{A}_1 \cong_{\delta_k} \mathcal{A}_2$, if $\mathcal{A}_1 \leq_{\delta_k} \mathcal{A}_2$ and $\mathcal{A}_2 \leq_{\delta_k} \mathcal{A}_1$.

8.4.2 Discounted Approximate Simulation

We define a new kind of approximate simulation called *Discounted Approximate Simulation (DAS)* for proving discounted approximate implementations for task-PIOAs. Given a distribution μ over executions (or traces) we denote the longest execution (respectively trace) in the support of μ by $\mathsf{L}(\mu)$. We extend this notation to a pair of distributions, $\mathsf{L}(\mu_1, \mu_2) \triangleq \max(\mathsf{L}(\mu_1), \mathsf{L}(\mu_2))$, that is, it is the length of longest execution in the support of either μ_1 or μ_2.

Definition 8.16. Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are two comparable closed task-PIOAs, and $\{\phi_k\}_{k\in\mathbb{N}}$ is a collection of functions, where each $\phi_k : \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_1}) \times \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}_2}) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. Given a collection of real number pairs $\{\epsilon_k, \delta_k\}_{k\in\mathbb{N}}$, the collection $\{\phi_k\}$ is an (ϵ_k, δ_k)-*discounted approximate simulation* from $\mathcal{A}_1$ to $\mathcal{A}_2$ if there exists a function $\mathsf{c} : R_1^* \times R_1 \to R_2^*$ such that the following properties hold:

1. *(Start condition)* $\phi_0(\bar{\nu}_1, \bar{\nu}_2) \leq \epsilon_0$.
2. *(Step condition)* If for all $k \leq \mathsf{L}(\mu_1, \mu_2)$, $\phi_k(\mu_1, \mu_2) \leq \epsilon_k$, $T \in R_1, \rho \in R_1^*$, μ_1 is consistent with ρ, and μ_2 is consistent with $\mathsf{full}(\mathsf{c})(\rho)$, then for all $k \leq \mathsf{L}(\mu_1', \mu_2')$, $\phi_k(\mu_1', \mu_2') \leq \epsilon_k$, where $\mu_1' = \mathsf{apply}(\mu_1, T)$ and $\mu_2' = \mathsf{apply}(\mu_2, c(\rho, T))$.
3. *(Trace condition)* If for all $k \leq \mathsf{L}(\mu_1, \mu_2)$, $\phi_k(\mu_1, \mu_2) \leq \epsilon_k$ then for all $k \leq \mathsf{L}(tdist(\mu_1), tdist(\mu_2))$ $\mathbf{d}_k(tdist(\mu_1), tdist(\mu_2)) \leq \delta_k$.

A family of functions $\{\phi\}_k$ is an (ϵ_k, δ_k) discounted approximate simulation if (i) The initial distributions $\bar{\nu}_1$ and $\bar{\nu}_2$ are ϵ_0-close in the sense of ϕ_0. (ii) Suppose two distributions μ_1, μ_2 are ϵ_k-close in the sense of ϕ_k, for every $k \leq \mathsf{L}(\mu_1, \mu_2)$. then the distributions μ_1'—obtained by applying a new task to μ_1, and μ_2'—obtained by applying a corresponding sequence of tasks to μ_2, are also ϵ_k-close in the sense of ϕ_k, for every $k \leq \mathsf{L}(\mu_1', \mu_2')$. (iii) Finally, if μ_1, μ_2 are ϵ_k-close in the sense of ϕ_k, for every $k \leq \mathsf{L}(\mu_1, \mu_2)$. then the corresponding trace distributions are δ_k-close in the sense of $\mathbf{d}_k$ for every $k \leq \mathsf{L}(tdist(\mu_1), tdist(\mu_2))$. Example 8.1 shows a simple approximate simulation function. Approximate simulations are similar to simulation relations of Definition 8.7 with the relations R and $\mathcal{E}(R)$ replaced by the real valued simulation functions ϕ and $\hat{\phi}$.

Note that although we have not the expanded version of the ϕ_k's in defining the step condition, it might be possible to obtain more powerful expanded discounted approximate simulations by changing the requirement $\phi_k(\mu_1', \mu_2') \leq \epsilon_k$ to $\hat{\phi}_k(\mu_1', \mu_2') \leq \epsilon_k$.

8.4.3 Soundness of Discounted Approximate Simulation

We prove Theorem 8.15 which states that (ϵ_k, δ_k)- approximate simulations are sound with respect to δ_k-approximate implementations.

Theorem 8.15. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two closed isomorphic comparable task-PIOAs. If there exists a (ϵ_k, δ_k)-discounted approximate simulation function from $\mathcal{A}_1$ to $\mathcal{A}_2$ then $\mathcal{A}_1 \leq_{\delta_k} \mathcal{A}_2$.*

Proof. Let ϕ be the assumed (ϵ_k, δ_k)-discounted approximate simulation function from $\mathcal{A}_1$ to $\mathcal{A}_2$. Let μ_1 be the probabilistic execution of $\mathcal{A}_1$ generated by the starting distribution $\bar{\nu}_1$ and a finite task schedule $T_1, T_2, \ldots, T_n$. For each $i > 0$, we define ρ_i to be $\mathsf{c}(T_1 \ldots T_{i-1}, T_i)$. Let μ_2 be the probabilistic execution of $\mathcal{A}_2$ generated by $\bar{\nu}_2$ and the concatenation $\rho_1, \rho_2, \ldots, \rho_n$. It suffices to show that $\mathbf{d}_w(tdist(\mu_1), tdist(\mu_2)) \leq \delta$.

For each $j \geq 0$, let us define $\mu_{1,j} \triangleq \mathsf{apply}(\bar{\nu}_1, T_1, \ldots, T_j)$ and $\mu_{2,j} \triangleq \mathsf{apply}(\bar{\nu}_2, \rho_1, \ldots, \rho_j)$. For $i \in \{1,2\}$ and for each $j \geq 0$, $\mu_{i,j} \leq \mu_{i,j+1}$ and $\mu_{i,n} = \mu_i$. Observe that for every $j \geq 0$, $\mu_{1,j+1} = \mathsf{apply}(\mu_{1,j}, T_{j+1})$ and $\mu_{2,j+1} = \mathsf{apply}(\mu_{2,j}, \rho_{j+1})$.

We prove by induction that for all $j \geq 0$, for all $k \leq \mathsf{L}(\mu_{1,j}, \mu_{2,j})$, $\phi_k(\mu_{1,j}, \mu_{2,j}) \leq \epsilon_k$. For $j = 0$, $\mu_{1,0} = \bar{\nu}_1$ and $\mu_{2,0} = \bar{\nu}_2$. By the start condition of the simulation function, $\phi_0(\mu_{1,0}, \mu_{2,0}) \leq \epsilon$. For the inductive step, we assume that for all $k \leq \mathsf{L}(\mu_{1,j}, \mu_{2,j})$, $\phi_k(\mu_{1,j}, \mu_{2,j}) \leq \epsilon_k$. Then, from Part (ii) of Definition 8.12 it follows that for all $k \leq \mathsf{L}(\mu_{1,j+1}, \mu_{2,j+1})$, $\phi_k(\mu_{1,j+1}, \mu_{2,j+1}) \leq \epsilon_k$. In particular, for all $k \leq \mathsf{L}(\mu_1, \mu_2)$, $\phi_k(\mu_1, \mu_2) \leq \epsilon_k$, from which, using condition (iii) it follows that for all $k \leq \mathsf{L}(tdist(\mu_1), tdist(\mu_2))$, $d_k(tdist(\mu_1), tdist(\mu_2)) \leq \delta_k$. ∎

Example 8.4. (Continuation of Example 8.3.) Let $\epsilon_k = \delta_k = (p+\epsilon)^k - p^k$, for each $k \in \mathbb{N}$. We will show that $\mathcal{A}_1$ and $\mathcal{A}_2$ are δ_k-equivalent using the following discounted approximate simulation:

$$\text{for each } k, \quad \phi_k(\mu_1, \mu_2) = max_{\alpha, \mathsf{anum}(\alpha)=k} |\mu_1(\alpha) - \mu_2(\alpha)|, \tag{8.6}$$

where $\mu_1 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_1})$, $\mu_2 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_2})$, and $\mathsf{anum}(\alpha)$ is the number of occurrence of the action try in the execution α.

Proposition 8.16. The collection of functions $\{\phi_k\}$ defined above is indeed an (ϵ_k, δ_k)-discounted approximate simulation from $\mathcal{A}_1$ to $\mathcal{A}_2$.

Proof sketch. We check that the collection $\{\phi_k\}$ satisfies the three conditions in Definition 8.16.

Start condition. $\nu_1 = \nu_2 = \delta_{s_{10}}$, and therefore $\phi_0(\nu_1, \nu_2) = 0$.

Step condition. We define the task correspondence function in the obvious way, $\mathsf{c}(\rho, T) \triangleq T$, where ρ is a task schedule and T is a task for $\mathcal{A}_1$. Thus for any $\mu_1 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_1})$ and $\mu_2 \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_2})$ that are obtained from ν_1 and ν_2 by applying a sequence of tasks, $\mathsf{L}(\mu_1, \mu_2) = \mathsf{L}(\mu_1) = \mathsf{L}(\mu_2)$. Consider any $\mu_1 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_1})$,$\mu_2 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_2})$, and suppose $\mu_1 = \mathsf{apply}(\nu_1, \rho)$ and $\mu_2 = \mathsf{apply}(\nu_2, \mathsf{full}(\mathsf{c})(\rho))$. Let us denote $\mu_1' = \mathsf{apply}(\mu_1, T)$, and $\mu_2' = \mathsf{apply}(\mu_2, \mathsf{c}(\rho, T)) = \mathsf{apply}(\mu_2, T)$. Then, it suffices to show that for all $k \leq \mathsf{L}(\mu_1, \mu_2)$, $\phi_k(\mu_1', \mu_2') \leq (p+\epsilon)^k - \epsilon^k$. This part of the proof is by a case analysis on the types of tasks, $T = \{\mathsf{try}\}, \{\mathsf{decide}\}$ and the types of executions.

The interesting cases are for $T = \{\mathsf{try}\}$ and executions of the form $\alpha = \alpha' \,\mathsf{try}\, s_{k0}$ or $\alpha = \alpha' \mathsf{try}\, s_{(k-1)1}$, for some $k \leq \mathsf{L}(\mu_1)$. For the first case, $\mu_1'(\alpha) = \mu_1(\alpha')p$ and $\mu_2'(\alpha) = \mu_2(\alpha')(p+\epsilon)$, and therefore $\phi_{k+1}(\mu_1', \mu_2') = p|\mu_2(\alpha') - \mu_1(\alpha')| + \epsilon\mu_2(\alpha')$. From the inductive hypothesis, $|\mu_2(\alpha') - \mu_1(\alpha')| \leq \epsilon_k$. It follows that, $\phi_{k+1}(\mu_1', \mu_2') \leq p|(p+\epsilon)^k - p^k| + \epsilon(p+\epsilon)^k \leq \epsilon^{k+1}$. Likewise in the second case, $\mu_1'(\alpha) = \mu_1(\alpha')(1-p)$ and $\mu_2'(\alpha) = \mu_2(\alpha')(1-p-\epsilon)$, and performing a similar calculation as above, we can show that $\phi_{k+1}(\mu_1', \mu_2') \leq \epsilon_k$.

Trace condition. First of all, for any $\mu_1 \in \mathsf{Disc}(\mathsf{Execs}^*_{\mathcal{A}_1})$ that are obtained from ν_1 by applying a sequence of tasks, $\mathsf{L}(\mu_1) = \mathsf{L}(tdist(\mu_1))$. If β is a trace of the form $a^k d$, for some $k \geq 0$. Then, for $i \in \{1,2\}$, $tdist(\mu_i)(\beta) = \mu_i(\alpha)$, where $\alpha = s_{10}\mathsf{a}s_{20} \ldots s_{k0}\mathsf{try}s_{k1}\mathsf{decide}s_{k2}$. From which it follows that $|tdist(\mu_1)(\beta) - tdist(\mu_2)(\beta)| = |\mu_1(\alpha) - \mu_2(\alpha)| \leq \phi_{k+1}(\mu_1, \mu_2) \leq \epsilon_{k+1}$. On the other hand, if β is a trace of the

form a^{k+1}, for some $k \geq 0$. Then, for $i \in \{1,2\}$, $tdist(\mu_i)(\beta) = \mu_i(\alpha_1) + \mu_i(\alpha_2)$, where $\alpha_1 = s_{10}$ try $s_{20} \ldots s_{(k+1)0}$ try $s_{(k+1)1}$ and where $\alpha_2 = s_{10}$ try $s_{20} \ldots s_{(k+2)0}$. Thus, $|tdist(\mu_1)(\beta) - tdist(\mu_2)(\beta)| = |\mu_1(\alpha_1) + \mu_1(\alpha_2) - \mu_1(\alpha_1) - \mu_2(\alpha_1)| = |\mu_1(\alpha) - \mu_1(\alpha)|$, where $\alpha = s_{10}$ try $s_{20} \ldots s_{(k+1)0}$. Therefore, $|tdist(\mu_1)(\beta) - tdist(\mu_2)(\beta)| \leq \phi_{k+1}(\mu_1, \mu_2) \leq \epsilon_{k+1}$ as required.

8.5 Approximations for Task-PIOAs

In this section, we discuss how the notion of uniform approximate implementations and the soundness of EASs extendeds to general (not necessarily closed) task-PIOAs. In an analogous manner, discounted approximate implementation and DAS can also be extended.

The basic idea is to define a new notion of implementation following the approach of [CCK+06a]. As we defined external behavior in the case of PTIOAs, we formulate the external behavior of a task-PTIOA $\mathcal{A}$ as a mapping from possible "environments" for $\mathcal{A}$ to sets of trace distributions that can arise when $\mathcal{A}$ is composed with the given environment.

Definition 8.17. An *environment* for task-PIOA $\mathcal{A}$ is a task-PIOA $\mathcal{E}$ such that the composition of $\mathcal{A}$ and $\mathcal{E}$ is closed.

Definition 8.18. The *external behavior* of a task-PIOA $\mathcal{A}$, written as $\mathsf{ExtBeh}_{\mathcal{A}}$, is a function that maps each environment task-PIOA $\mathcal{E}$ for $\mathcal{A}$ to the set of trace distributions of the composition of $\mathcal{A}$ and $\mathcal{E}$.

Approximate implementation for general task-PIOAs can then be defined to be closeness of external behavior for all environments.

Definition 8.19. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable then $\mathcal{A}_1$ is said to δ-*implement* $\mathcal{A}_2$, for some $\delta \geq 0$, if for every environment task-PIOA $\mathcal{E}$ for both $\mathcal{A}_1$ and $\mathcal{A}_2$, for every $\mu_1 \in \mathsf{ExtBeh}_{\mathcal{A}_1}(\mathcal{E})$ there exists $\mu_2 \in \mathsf{ExtBeh}_{\mathcal{A}_2}(\mathcal{E})$ such that $\mathbf{d}_u(\mu_1, \mu) \leq \delta$.

Based on this modified definition of approximate implementation the soundness of expanded approximate simulations for general task-PIOAs follow as a Corollary to Theorem 8.11.

Corollary 8.17. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two comparable task-PIOAs. Suppose that for every environment $\mathcal{E}$ for both $\mathcal{A}_1$ and $\mathcal{A}_2$, there exists a $(\epsilon_{\mathcal{E}}, \delta)$-approximate simulation function from the composition of $\mathcal{A}_1$ and $\mathcal{E}$ to the composition of $\mathcal{A}_2$ and $\mathcal{E}$. Then $\mathcal{A}_1 \leq_\delta \mathcal{A}_2$.*

8.6 Related Work

A variety of implementation relations have beep proposed in the literature for probabilistic automata [LS91, LMMS98, CCK+06b, C. 96, SvdS05, BLB05]. These notions of implementation rely on equality of observable behavior. That is, every observable behavior of the concrete system must be exactly equal to some observable behavior of the abstract specification. The fragility of these equality-based implementation relations have been known for quite some time [JS90, DJGP02, GJP04] In [JS90] Jou and Smolka formalized "similarity" of traces using a metric and developed the corresponding notion of approximate equivalence for probabilistic automata. Approximation metrics for probabilistic systems in the context of *Labelled Markov Processes (LMP)* have been extensively investigated

and many fundamental results have been obtained by Desharnais, Gupta, Jagadeesan and Panangaden [DJGP02, DGJP03, DGJP04] and by van Breugel, Mislove, Ouaknine, and Worrell [vBW01b, vBMOW03, vBMOW05, MWMW04, MOW04]. The first set of authors introduced a Kantorovich-like metric for LMPs and presented the logical characterization of this metric. Van Breugel *et al.* have presented intrinsic characterizations of the topological space induced by the above metric. This characterization is based on a final coalgebra for a functor on the category of metric spaces and nonexpansive maps. Another interesting facet of this body of work is the polynomial time algorithm for computing the metric presented by van Breugel and Worrell in [vBW01a]. For *Generalized Semi-Markov Processes (GSMP)* [GJP04], Gupta, Jagadeesan and Panangaden have developed pseudo-metric analogues of bisimulation and have shown that certain observable quantitative properties are continuous with respect to the introduced metric. Kwiatkowska and Norman have developed the denotational semantics for a divergence-free probabilistic process algebra based on a metric on probability distribution over executions [KN96]. In the non-probabilistic setting, Girard and Pappas [GP05, GJP06] have developed the theory of approximate implementations for *Metric Transition Systems (MTS)*. The state space and the space of external actions of an MTS are metric spaces. Based on these metrics, the authors develop a hierarchy of approximation pseudo-metrics between MTSs measuring distance between reachable sets, sets of traces and bisimulations. The authors have also developed algorithms for exactly and approximately computing these metrics.

Our work differs from all of the above in at least one of the following ways: (a) the task-PIOA model allows both nondeterministic and probabilistic choices, and (b) the implementation relation in our framework is based on trace distributions and not bisimilarity of states. Approximate implementation is derived from a metric over trace distributions, and thus, we do not require the state spaces of the underlying automata nor the common space of external actions to be metric spaces. Metrics on trace distributions of PIOAs are used by Cheung in [Che06]to show that sets of trace distributions form closed sets in a certain metric space. This result is then used to show that finite tests are sufficient to distinguish between a members of a certain class of PIOAs. The metric used in the above work is related to our uniform metric but it is defined on the set $[0,1]^{Traces}$ whereas our uniform metric is exclusively defined on the set of trace distributions.

8.7 Summary

In this chapter we have introduced the notion of approximate implementations for task-structured probabilistic I/O automata. In particular, we have defined two different kinds of approximate implementations, based on the uniform metric and the discounted uniform metric on trace distributions of task-PIOAs. We proposed expanded approximate simulations and discounted approximate simulations for proving, the two proposed implementation relations, respectively. EAS and DAS can be used to approximately reason about probabilistic safety and termination properties. PIOAs can be nondeterministic and our construction does not require the underlying state spaces of the automata or the space of external actions to be metric spaces.

In our formulation of expanded approximate simulations, a simulation proof reduces to finding an optimal joint distribution satisfying certain constrains on the marginals. This is closely related to the well known Kantorovich optimal transportation problem [Kan]. For well-behaved classes of simulation functions, therefore, we would like to explore the

possibility of proving approximate simulations by solving optimization problems.

In the future, we want develop a new kind of *Discounted Expanded Approximate Simulations* that combines the features of EAS and DAS. We would also like to develop simulation based proof techniques where the simulation functions are functions of distributions over states and not functions of distributions over execution fragments. Finally, we would like to extend the notion approximate implementations to the Probabilistic Timed I/O Automaton framework 7.

8.8 Appendix: Limits of Chains of Distributions

All the definitions and lemmas in this Appendix are from [CCK+06b]. In this Appendix $\mathcal{A}$ will be a task-PIOA. Given a finite execution fragment α of $\mathcal{A}$, the cone of executions generated by this fragment C_α is the set of all execution fragments that extend α. Given a finite trace β of $\mathcal{A}$, C_α is the set of all traces that extend β.

Definition 8.20. If $\mu_1, \mu_2 \in \mathsf{Disc}(\mathsf{Frags}_{\mathcal{A}})$, such that for every $\alpha \in \mathsf{Frags}^*_{\mathcal{A}}$, $\mu_1(C_\alpha) \leq \mu_2(C_\alpha)$, then we write $\mu_1 \leq \mu_2$.

Definition 8.21. A *chain* of probability measures on execution fragments of $\mathcal{A}$ is an infinite sequence $\mu_1, \mu_2, \ldots$ of probability measures on execution fragments of $\mathcal{A}$ such that $\mu_1 \leq \mu_2 \ldots$. Given a chain, the *limit* of the chain is defined as a function μ on the σ-algebra generated by the cones of execution fragments of $\mathcal{A}$, as follows: for each $\alpha \in \mathsf{Frags}^*_{\mathcal{A}}$, $\mu(C_\alpha) \triangleq \lim_{i\to\infty} \mu_i(C_\alpha)$.

Standard measure theoretic arguments guarantee that μ can be extended uniquely to a probability measure on the σ-field generated by the cones of finite execution fragments.

Definition 8.22. If μ_1, μ_2 are probability measures on traces of $\mathcal{A}$, such that for every finite trace β of $\mathcal{A}$ $\mu_1(C_\beta) \leq \mu_2(C_\beta)$, then we write $\mu_1 \leq \mu_2$.

Definition 8.23. A *chain* of probability measures on traces of $\mathcal{A}$ is an infinite sequence $\mu_1, \mu_2, \ldots$ of probability measures on traces of $\mathcal{A}$ such that $\mu_1 \leq \mu_2 \ldots$. Given a chain of probability measure on traces, the *limit* of the chain is defined as a function μ on the σ-algebra generated by the cones of traces of $\mathcal{A}$, as follows: for each finite trace β of $\mathcal{A}$, $\mu(C_\beta) \triangleq \lim_{i\to\infty} \mu_i(C_\beta)$.

Again, μ can be extended uniquely to a probability measure on the σ-field generated by the cones of finite traces.

Lemma 8.18 (4 of [CCK+06b])**.** *Let $\mu_1, \mu_2, \ldots$ be a chain of measures on $\mathsf{Frags}_{\mathcal{A}}$ and let $\mu = \lim_{i\to\infty} \mu_i$, then $\lim_{i\to\infty} tdist(\mu_i) = tdist(\mu)$.*

Lemma 8.19 (11 of [CCK+06b])**.** *Let $\mu \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$ and σ be a finite task schedule for $\mathcal{A}$. Then $\mathsf{apply}(\mu, \rho) \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$.*

Lemma 8.20 (20 of [CCK+06b])**.** *Let $\mu \in \mathsf{Disc}(\mathsf{Frags}^*_{\mathcal{A}})$ and $\rho_1, \rho_2, \ldots$ be a finite or infinite sequence of task schedulers for $\mathcal{A}$. For each $i > 0$ let $\eta_i = \mathsf{apply}(\mu, \rho_1\rho_2 \ldots \rho_i)$. Let $\rho = \rho_1\rho_2 \ldots$ be the concatenation of the all the task schedulers, and let $\eta = \mathsf{apply}(\mu, \rho)$. Then the η_i's form a chain and $\eta = \lim_{i\to\infty} \eta_i$.*

Chapter 9

Conclusions

9.1 Evaluation

The objective of this PhD thesis project was to explore a general class of hybrid system models with the purpose of developing specification and verification for embedded software systems that interact with physical processes.

The language proposed here, HIOA, provides structures and compositional semantics for specifying a very general class of *nonprobabilistic* hybrid systems. We have found that a variety of system models including vehicle and air-traffic control systems [UL07, MWLF03], cardiac cell models used in systems biology [GMY+07], algorithms for mobile robotics [LMN], real-time and distributed algorithms [FDGL07, ALL+06, CLMT05], can be expressed naturally in HIOA . While in the thesis we focused on formal analysis of HIOA specifications, these specifications are also amenable to simulation. The TIOA subset of HIOA forms the backbone of the Tempo toolset [TEM07]. This toolset supports simulation and model checking, in addition to some of the verification techniques presented in the thesis, hence making it possible to apply different analysis techniques to the same hybrid system specification.

For the majority of the applications we have studied, direct automatic verification would be impossible, and therefore, we decided to focus on verification methods that depend on user guidance. Invariance and implementation verification techniques proposed here rely on inductive properties and simulation relations—which the user has to supply; but once these are obtained, the desired property can be deduced with relative ease. In fact, our theorem prover interface for HIOA partially, and sometimes completely, automates this deduction process. In practice, finding the invariants and checking them is often an iterative process, where one starts by guessing an invariant and then attempts to prove it; if the proof fails then the invariant is refined and so on. This iterative process can be performed more efficiently using our theorem prover interface because the existing proof script can be executed automatically on the modified invariant.

Apart from the case studies presented here, the theorem prover interface has been applied by Umeno [UL07] in a major case study concerning safety verification of an aircraft landing protocol. This is encouraging, and illustrates that it is *possible* to model and verify realistic hybrid system models in our framework. Constructing proofs, even with the aid our strategies, is not routine and takes significant effort. This is because our strategies are able to automate only shallow proofs. We expect that new domain-specific strategies will make the deductive verification practical for hybrid models that have relatively simple continuous dynamics.

Our optimization-based stability verification approach gives a family of methods in which the hardness of the optimization problem that one has to solve (for verifying stability) depends on the complexity of the dynamics of the hybrid model in question. For certain classes of hybrid systems, this means that stability can be verified automatically using standard mathematical programming tools. These types of methods can help us decide on the subclass of SHIOAs that we should restrict our attention to, while modeling the system, so as to make verification feasible with the available resources.

Techniques for verifying quantitative properties allow us to prove performance (or reliability) of a system within the same formal framework is used for proving correctness. We have extended the usual notions of implementation for (discrete) probabilistic automata to obtain approximate implementation relations. This lets us quantify how close an implementation is to an ideal specification. We have developed sound simulation-based proof techniques for establishing these relationships. We have recently used these techniques to verify the statistical zero-knowledge property of a classical authentication protocol [GPS06] (not presented in the thesis).

In order to state and verify quantitative properties for hybrid systems we have to incorporate probabilities in the hybrid system model. This motivated us to develop the Probabilistic Timed I/O Automata model. PTIOAs can model concurrent execution of components, discrete and continuous evolution, and probabilistic choices based on general continuous probability distributions. In developing semantics PTIOAs we had to first resolve nontrivial measurability related issues and then find a way for resolving nondeterminism. Defining a set of axioms that ensure measurability of reasonable sets of executions and traces, and adapting the task mechanism of [CCK+06a] for resolving nondeterminism, we were able to develop trace-based semantics for PTIOAs. Currently PTIOAs provide framework for modeling probabilistic hybrid systems that arise in, for example, randomized, real-time algorithms, timing based security protocols, and control systems with noisy sampling, but much work remains to be done in developing verification techniques for boolean and quantitative properties.

9.2 Future Directions

The state of research for probabilistic and nonprobabilistic hybrid systems are at two different levels. The latter being more mature, future efforts should focus on new software tools for verification and applications, while in the case of probabilistic hybrid systems emphasis should be on developing the basic verification techniques.

9.2.1 Modeling Probabilistic Hybrid Systems

We show that the composition of two PTIOAs is also a PTIOA only if the composite automaton satisfies an additional measurability requirement, namely, axiom **M2**. PTIOAs we encountered do satisfy **M2**, but we would like to define a subclass of PTIOAs that is closed under composition (without any extra assumptions). For obtaining this property it may be necessary to impose restrictions on the trajectories of PTIOAs or to impose additional measurability requirements.

A second improvement for the PTIOA framework would be to include input/output variables that will enable component automata to communicate continuously. Including input variables pose a technical challenge: in defining the probability measure over executions of such an automaton, the maximal progress assumption implied by time-action determinism

D3 will have to be removed. This is because a trajectory, in that case, would determined only in part by the state, and it will also depend on the inputs.

Finally, the relative strengths of the different scheduling mechanisms has to be studied. In this thesis we used oblivious task schedulers for resolving external nondeterminism and local schedulers for resolving internal nondeterminism. Other types of schedulers, such as Markovian, history-dependent, and randomized schedulers should be considered for PTIOAs. We can remove the time-action determinism axiom **D3** by assuming that each PTIOA component has a local scheduler, and therefore most of the scheduling conflicts are resolved, except for races. Smolka and Stark [ES03, Sta03], assume that such conflicts happen only with probability 0, and so can be ignored. The trace-based semantics obtained for these different scheduling mechanism will no doubt be different from the one we have proposed. Then, we have to draw a clear recipe of which semantics is suitable for what application domain.

9.2.2 Stability

The definition of stability used in the hybrid systems literature and also in this thesis is strictly concerned with the continuous state of the system. On the other hand, the notion of *self-stabilization* has been widely used in designing distributed algorithms [Dol00, NA02]. A distributed algorithm is said to be self-stabilizing if it stabilizes to correct behavior, from an arbitrary state, if the underlying components start "behaving well" from some point onward. This is a common notion in algorithms designed for somewhat fault-prone environments that must continue operations, even after periods of failures, e.g., the Internet, or a mobile computing system. It is necessary to propose a general definition of stability of hybrid systems, that encompasses both its discrete and its continuous variables. Theel et al. have recently applied Lyapunov function based techniques for verifying self-stabilization properties [DOT06], however, a theory that unifies these two distinct notions of stability remains unknown. This generalized notion of stability will enable us to prove stability of a hybrid system as a whole. Further, we can try to carry the techniques for developing self-stabilizing algorithms, like those in [Dol00], over to designing stable hybrid systems operating in uncertain real-world environments. For example, in a mobile network setting, if we lose location information about where nodes are, we can search for them in the real world, and restore the information to where it belongs.

Extending the the stability verification techniques proposed here to the stochastic setting is a promising direction for future research. Stability conditions for probabilistic switched systems using Lyapunov function-like arguments have been derived in [CL06, CL04]. It can be shown that for certain classes of PTIOAs, the problem of proving *almost sure stability*, using these Lyapunov arguments can be formulated as optimization problems that can be solved efficiently. In the simplest case, we consider PTIOAs where the discrete transitions are triggered by a discrete time Markov chain. That is, the mode switches occur periodically every Δ time and the new modes is chosen probabilistically. In this simple setting it can be shown that the Average Dwell Time (ADT) property reduces to detecting the minimum cost cycle in a graph, as in the case of one-clock initialized SHIOAs. It would be interesting to explore similar verification approaches for other, more expressive, classes of probabilistic hybrid systems.

9.2.3 Approximate Implementations

Research on quantitative and approximate verification techniques is in its early stages and much work remains to be done. In the context of discrete probabilistic automata, an immediate research goal, is to develop *discounted expanded simulations* as a more powerful substitute for the ordinary discounted simulation proposed here.

In our formulation of expanded approximate simulations, a simulation proof reduces to finding an optimal joint distribution satisfying certain constrains on the marginals. This is closely related to the well-known Kantorovich optimal transportation problem [Kan]. Exploring the connection between these two problems, it may become possible to prove approximate simulations by solving optimization problems.

Notions of approximate implementations for probabilistic hybrid will have to be robust with respect to small perturbations to 1. time of occurrence of events, and also 2. probability of occurrence of events. This means that we have to use metrics on trace distributions that take into account both the differences in timing and probability characteristics. Fortunately, there exists a rich body of literature (see, for example, [Rac91]) that offers many choices for such metrics on probability distributions. Based on these metrics we should develop notions of approximate implementation for PTIOAs and simulation-based techniques for establishing these relationships. The techniques introduced in Chapter 8 will be useful in attacking this problem, but tools from functional analysis will also have to be employed along the way.

9.2.4 Software Tools

The HIOA -PVS interface enables effective verification of hybrid system models with complex data-types and logical expressions. However, significant user interaction becomes necessary for models with nonlinear algebraic equations. We have to address this issue by developing a verification platform, based on Tempo, that integrates theorem provers, model checkers, computer algebra systems, and mathematical program solvers. As an example of how these different tools can be utilized with in the same framework, consider the safety and stability verification conditions of Chapters 4 and 5. These conditions could be checked by computer algebra systems and optimization tools, whereupon, these conditions can be used by theorem prover strategies for automatically deducing the properties.

The long-term success of any model-based software development methodology depends on how well *marginal efforts in development translate to quantifiable gains in quality*. Building coarse models and proving that they roughly conform to an ideal design should be relatively easy; and by refining models, improving designs, and expending extra effort in verification, it should be possible to prove closer conformance, and sharper guarantees. It is important that we create software development platforms that are recognized to have the above marginal gain property. Approximate implementations provide one possible basis for measuring conformance, and it is necessary to develop other, non-probabilistic metrics.

List of Symbols and Functions

$\mathcal{A} \leq \mathcal{B}$ — $\mathcal{A}$ implements $\mathcal{B}$, i.e., $\mathsf{Traces}_{\mathcal{A}} \subseteq \mathsf{Traces}_{\mathcal{B}}$, 27

$\mathcal{A} \leq_{\delta} \mathcal{B}$ — $\mathcal{A}$ δ-approximately implements $\mathcal{B}$ w.r.t. uniform metric, 166

$\mathcal{A} \leq_{\delta_k} \mathcal{B}$ — $\mathcal{A}$ δ_k-approximately implements $\mathcal{B}$ w.r.t. discounted uniform metric, 178

$\mathcal{A} \leq_{switch} \mathcal{B}$ — $\mathcal{A}$ switches slower than $\mathcal{B}$, 80

$\mathcal{D}$ — Set of discrete transitions of a hybrid automaton, 25

$\mathsf{Disc}(Q)$ — Set of discrete probability distributions over $(Q, \mathcal{F}_Q)$, 160

$\mathsf{Execs}_{\mathcal{A}}$ — Set of all executions of automaton $\mathcal{A}$, 26

$\mathsf{ExtBeh}_{\mathcal{A}}$ — External behavior of A, a map from environment($\mathcal{E}$) to $\mathsf{Tdists}_{\mathcal{A}\mathcal{E}}$, 151

$\mathsf{Frags}_{\mathcal{A}}$ — Set of all execution fragments of automaton $\mathcal{A}$, 26

$\mathcal{M}$ — Set of mode switching discrete transitions of a SHIOA, 35

$(S, \mathcal{F}_S)$ — A measurable space on set S; $\mathcal{F}_S$ is a σ-algebra over S., 136

$\mu_{\mathbf{x},B}$ — If unique action $a \in B$ is enabled at $\mathbf{x}$ such that $\mathbf{x} \xrightarrow{a} \mu$, then $\mu_{\mathbf{x},B} = \mu$, 138

$\mathsf{Reach}_{\mathcal{A}}$ — Set of all reachable states of automaton $\mathcal{A}$, 26

$\mathsf{supp}(\mu)$ — Support of discrete probability measure μ, 160

$\mathsf{Tdists}_{\mathcal{A}}$ — Set of trace distributions for $\mathcal{A}$, 151

$\mathbb{T}$ — Set of time points which is $\mathbb{R}_{\geq 0} \cup \{\infty\}$, 22

$\mathsf{Traces}_{\mathcal{A}}$ — Set of all traces of $\mathcal{A}$, 27

$\mathbf{v}, \mathbf{x}$ — A valuation for the set of variables V, X., 23

$b[i]$ — The ith element of array (or tuple) b, 22

$dom(f)$ — Domain of function f, 22

$dtype(v)$ — Dynamic type of variable v, 23

E_B — set of states from which some action $a \in B$ is enabled, 137

$E_{R,P}$ — set of states from which there exists a closed trajectory τ, such that $\tau.ltime \in R$ and $\tau.lstate \in P$, 138

$enabled(\mathbf{x})$	Set of actions enabled at state $\mathbf{x}$, 137
$enabled(a)$	Set of states at which action a is enabled, 25
$ltime$	The limit time of a trajectory or an execution fragment, 24
$N(\alpha)$	Number of mode switches over execution fragment α, 77
$range(f)$	Range of function f, 22
$S_{\tau_a}(\alpha)$	Number of extra switches in execution fragment α w.r.t. ADT τ_a, 78
$tdist(\rho)$	$tdist(\mu_\rho)$ or the trace distribution corresponding to the task schedule ρ, 151
$type(v)$	Static type of variable v, 23
$val(V)$	Set of all valuations of the variables V, 23

Index

abstraction, *see* implementation
Actuator, 62
ADT, 77
    equivalent, 80
automaton
    generalized PTIOA, 152
    PIOA, 160
    pre-PTIOA, 138
    Task DPTIOA, 139
    task-PIOA, 160
Average Dwell Time, *see* ADT

base, 143
basic set, 143
Bounce, 38
    in PVS, 113
Burner, 85

ClockSync, 48
compatible, 27
composition, 27, 29, 35, 141, 153
    in HIOA language, 49
    of PIOA, 161
compositionality, 12
cone, 162

DAI, 30
    in HIOA language, 47
deadline variable, 40
deterministic trajectories, 136
Dirac measure, 136
dtype, 23
    in HIOA language, 41
    continuous, 30
    discrete, 30
dynamic type, *see* dtype

environment, 181
environment automaton, 151
execution, 26, 140, 161
    fragment, 26, 140
    probabilistic, 146
expansion
    of function, 166
    of relation, 164
external behavior, 151

failure detector, 50, 129
flattening, 163
formal parameter, *see* parameter
forward simulation relation, 55
functions
    in HIOA language, 43

Helicopter, 58
hierarchical refinement, 12
hysteresis
    linear, 91
    scale-independent, 86
HystSwitch, 86

implementation, 27, 29, 35, 151, 153
    discounted approximate, 178
    proving inductively, 55
    uniform approximate, 165
inductive property, 54
invariant, 26, 53
    in HIOA language, 50
    in PVS, 117
    of composed automaton, 55
    proving inductively, 54

kinds
    of variables, 40

lifting, 164
LinHSwitch, 91
Lyapunov stability, *see* stability

Markov chain, 172
metric, 165
mode switch, 35, 77

NoisySensor, 154

parameter, 39
  constant, 44
  type, 40
polymorphism, 40
pre-HIOA, 29
pre-SHIOA, 34
precondition
  in HIOA language, 44
preconditions
  in PVS, 112
pseudometric, 165
PTIOA, 137
PVS strategy, *see* strategy

reachability, 26
reachable state, 26
reachable states, 26
restriction, 26

safety, 26
scheduler, 145
semi-ring, 137
Sensor, 58
SHIOA
  initialized, 35, 96
  linear, 35
  one-clock initialized, 90
  rectangular, 35
signature, 43
  in PVS, 110
simulation
  discounted approximate, 179
  expanded approximate, 168
simulation relation, 55
  in HIOA language, 50
  in PVS, 119
Spec, 50, 130
  in PVS, 115
stability, 26, 76
  asymptotic, 76
  exponential, 77
  global asymptotic, 76
  global exponential, 77
  uniform, 77
stable, 76
state models
  in HIOA language, 45
  in PVS, 110
  semantics, 30
static type, *see* type
strategy, 105, 123
substitutivity, 152
  HA, 28
Supervisor, 59
switched system, 75
switching simulation relation, 80

task correspondence, 164
task schedule, 145
see Task DPTIOA, 139
tasks, 139
Two-TaskRace, 130
theory instantiation, 120
thermostat, 47
Thermostat2, 100
Thermostat, 47
TimedChannel, 47
Timeout, 51
Timeout
  in PVS, 118
trace, 26, 161
  distribution, 148
  PTIOA, 140
trace basic set, 145
trace distribution, 151
trajectories
  in HIOA language, 45
  in PVS, 109, 113
trajectory, 24
  closed, 24
  open, 24
  point, 24
transitions
  in HIOA language, 44
  in PVS, 111, 112
type, 23
  constructors, 40
  parameter, 40
types
  in HIOA language, 40

valuation, 23
variable
  continuous, 23
  discrete, 23

variables
    in HIOA Language, 39
    in PVS, 109
vocabulary, 40

where-clause, 38

Bibliography

[A+95] R. Alur et al. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[ABD+00] Eugene Asarin, Olivier Bournez, Thao Dang, Amir Pnueli, and Oded Maler. Effective synthesis of switching controllers for linear systems. In *Proceedings of IEEE*, volume 88, pages 1011–1025, July 2000.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, P.-H. Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHH93] R. Alur, C. Courcoubetis T. A. Henzinger, and P. H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer-Verlag, 1993.

[AHS98] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of UITP '98*, July 1998.

[ALL+06] Myla Archer, HongPing Lim, Nancy Lynch, Sayan Mitra, and Shinya Umeno. Specifying and proving properties of timed I/O automata in the TIOA toolkit. In *In Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*. IEEE, 2006.

[Alu91] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanfor University, 1991.

[AP04] Rajeev Alur and George J. Pappas, editors. *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25-27, 2004, Proceedings*, volume 2993 of *LNCS*. Springer, 2004.

[Arc01] Myla Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.

[ASL93] P. J. Antsakis, J. A. Stiver, and M. D. Lemmon. Hybrid system modeling and autonomous control systems. In R. L. Grossman et al., editors, *Hybrid Systems*, volume 736 of *LNCS*, New York, 1993.

[AVM03] Myla Archer, Ben Di Vito, and Cesar Munoz. Developing user strategies in pvs: A tutorial. In *STRATA 2003*, Rome, Italy, 2003.

[BBM98] M. Branicky, V. Borkar, and S. Mitter. A unified framework for hybrid control: model and optimal control theory. *IEEE Transactions on Automatic Control*, 43(1):31–45, 1998.

[BCG06] C. Baier, F. Ciesinski, and M. Grer. ProbMela and model checking markov decision processes. In *ACM Performance Evaluation Review on Performance and Verification*, 2006. to appear.

[BCT02] Alexandre M. Bayen, Eva Cruck, and Claire Tomlin. Guaranteed overapproximations of unsafe sets for continuous and hybrid systems: solving the hamilton-jacobi equation using viability techniques. In Tomlin and Greenstreet [TG02], pages 90–104.

[BGL02] A. Bogdanov, S. Garland, and N. Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In *Formal Techniques for Networked and Distributed Sytems - FORTE 2002 : 22nd IFIP WG 6.1 International Conference*, pages 364–368, Texas, Houston, USA, November 2002.

[BGM93] A. Back, J. Guckenheimer, and M. Myers. A dynamical simulation facility for hybrid systems. In R. L. Grossman et al., editors, *Hybrid Systems*, volume 736 of *LNCS*, New York, 1993.

[BLB05] Manuela L. Bujorianu, John Lygeros, and Marius C. Bujorianu. Bisimulation for general stochastic hybrid systems. In Morari and Thiele [MT05], pages 198–214.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC*, pages 27–30, Montreal, Canada, August 1983.

[Bra95] M. Branicky. *Studies in hybrid systems: modeling, analysis, and control*. PhD thesis, MIT, Cambridge, MA, June 1995.

[Bro94] R. W. Brockett. Hybrid models for motion control systems. In H. L. Trentelman and J. C. Willems, editors, *Essays in Control: Perspectives in the Theory and its Applications*, pages 20–53, Boston, 1994. Birkhauser.

[BS67] N. P. Bhatia and G. P. Szegö. *Dynamical Systems: Stability Theory and Applications*, volume 35 of *Lecture notes in mathematics*. Springer-Verlag, Berlin; New York, 1967.

[C. 96] C. Baier. Polynomial-time algorithms for testing probabilistic bisimulation and simulation. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 50–61, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[CCK+05] Ran Canetti, Ling Cheung, Dilsun Kirli, Moses Liskov, Olivier Pereira Nancy Lynch, and Roberto Segala. Using probabilistic I/O automata to analyze an oblivious transfer protocol. Technical report, MIT LCS Technical

Reports - MIT-LCS-TR-1001, August 2005. Also in MIT CSAIL Technical Reports - MIT-CSAIL-TR-2005-055.

[CCK+06a] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-structured Probabilistic I/O Automata. In *Proceedings of the 8th International Workshop on Discrete Event Systems – WODES'2006*, 2006. IEEE catalog number 06EX1259.

[CCK+06b] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-structured probabilistic I/O automata. Technical Report MIT-CSAIL-TR-2006-060, Massachusetts Institute of Technology, Cambridge, MA, September 2006.

[CCK+06c] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Time-bounded task-PIOAs: a framework for analyzing security protocols. In *20th International Symposium on Distributed Computing, (DISC 2006)*, volume 4167 of *LNCS*, pages 238–253. Springer, 2006.

[CCK+06d] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Using task-structured probabilistic I/O automata to analyze an oblivious transfer protocol. Technical Report MIT-CSAIL-TR-2006-019, Massachusetts Institute of Technology, Cambridge, MA, March 2006. Available from http://theory.csail.mit.edu/tds/papers/Kirli/TR-2006-019.pdf.

[CES86] E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Che06] L. Cheung. *Reconciling nondeterministic and probabilistic choices.* PhD thesis, ICIS, Radboud University Nijmegen, The Netherlands, 2006.

[CL04] D. Chatterjee and D. Liberzon. On stability of stochastic switched systems. In *Proceedings of the 43rd Conference on Decision and Control*, pages 4125–4127, Paradise Island, Bahamas, December 2004.

[CL06] Debasish Chatterjee and Daniel Liberzon. Stability analysis of deterministic and stochastic switched systems via a comparison principle and multiple lyapunov functions. *SIAM Journal on Control and Optimization*, 45(1):174–206, March 2006.

[CLMT05] Gregory Chockler, Nancy Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: an assertional approach. In Pierre Fraigniaud, editor, *Proceedings of Nineteenth International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *LNCS*, pages 152 – 168, Cracow, Poland, September 2005. Springer.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[CLSV04] L. Cheung, N. A. Lynch, R. Segala, and F. Vaandrager. Switched probabilistic I/O automata. In *First International Colloquium on Theoretical Aspects of Computing*, July 2004.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at http://www.csl.sri.com/wift-tutorial.html.

[Cru91] Rene L. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[CSKN05] Stefano Cattani, Roberto Segala, Marta Z. Kwiatkowska, and Gethin Norman. Stochastic transition systems for continuous state spaces and non-determinism. In *FoSSaCS'05*, volume 3441 of *LNCS*, pages 125–139. Springer, 2005.

[dA97] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, CA, 1997. Technical Report STAN-CS-TR-98-1601.

[Dav83] Martin Davis. The prehistory and early history of automated deduction. *Automated Reasoning*, 1:1–28, 1983.

[Dav93] M. H. A. Davis. *Markov Models and Optimization*. Chapman & Hall, 1993.

[DDLP05] Vincent Danos, Jose Desharnais, Francois Laviolette, and Prakash Panangaden. Bisimulation and cocongruence for probabilistic systems. *Information and Computation, Special issue for selected papers from CMCS04*, 2005.

[Dev99] Marco Devillers. Translating IOA automata to PVS. Technical Report CSI-R9903, Computing Science Institute, University of Nijmegen, February 1999. Available at http://www.cs.ru.nl/research/reports/info/CSI-R9903.html.

[DGJP03] Josée Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Approximating labelled markov processes. *Inf. Comput.*, 184(1):160–200, 2003.

[DGJP04] Jose Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled markov processes. *Theor. Comput. Sci.*, 318(3):323–354, 2004.

[DGRV00] M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.

[DJGP02] J. Desharnais, R. Jagadeesan, V. Gupta, and P. Panangaden. The metric analogue of weak bisimulation for probabilistic processes. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS), Copenhagen, Denmark, 22-25 July 2002*, pages 413–422. IEEE Computer Society, 2002.

[DL97] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In *HART'97 (International Workshop on Hybrid and Real-Time Systems)*, volume 1201 of *LNCS*. Springer Verlag, March 1997.

[DLL97] R. DePrisco, Butler Lampson, and Nancy Lynch. Revisitng the paxos algorithm. In *Distributed Algorithms 11th Workshop WDAG'97*, pages 111–125, Berlin-Heidelberg, 1997.

[Dol00] Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[Doo53] J. L. Doob. *Stochastic Processes*. John Wiley & Sons, Inc., New York, 1953.

[DOT06] A. Dhama, J. Oehlerking, and O. Theel. Verification of orbitally self-stabilizing distributed algorithms using lyapunov functions and poincare maps. In *12th Intl. Conf. on Parallel and Distributed Systems (ICPADS'06)*, pages 23–30, 2006.

[Dud76] R. M. Dudley. *Probabilities and Metrics:Convergence of laws on metric spaces, with a view to statistical testing*. Number 45 in Lecture Notes Series. Aarhus Universitet, June 1976.

[Dud89] R. M. Dudley. *Real Analysis and Probability*. Wadsworth, Belmont, Calif, 1989.

[ES03] S. Smolka E. Stark, R. Cleaveland. A process-algebraic language for probabilistic I/O automata. In *Proc. CONCUR 03*, volume 2761 of *LNCS 2761*, pages 189–203, Marseille, France, September 2003. Springer.

[FDGL07] Rui Fan, Ralph E. Droms, Nancy D. Griffeth, and Nancy A. Lynch. The dhcp failover protocol: A formal perspective. In *Formal Techniques for Networked and Distributed Systems - FORTE 2007*, pages 211–226, 2007.

[FK98] Enrique D. Ferreira and Bruce H. Krogh. Switching controllers based on neural network: Estimates of stability regions and controller performance. In *Hybrid Systems: Computation and Control 1998*, pages 126–142, 1998.

[Flo67] Robert Floyd. Assigning meanings to programs. In *Sympyposium on Applied Mathematics. Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[Fre05] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In Morari and Thiele [MT05], pages 258–273.

[GCMF93] Gordon, M. J. C., Melham, and Thomas F. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch prover. Technical report, DEC Systems Research Center, 1991. Available at `http://nms.lcs.mit.edu/Larch/LP`.

[GGC81] Keith O. Geddes, Gaston H. Gonnet, and Bruce W. Char. Maple user's manual. Technical Report CS-81-25, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1981.

[Gir81] M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects ofTopology and Analysis*, number 915 in Lecture Notes in Mathematics, pages 68–85. Springer-Verlag, 1981.

[GJP04] Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Approximate reasoning for real-time probabilistic processes. *The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, 00:304–313, 2004.

[GJP06] Antoine Girard, A. Agung Julius, and George J. Pappas. Approximate simulation relations for hybrid systems. In *IFAC Analysis and Design of Hybrid Systems*, Alghero, Italy, June 2006.

[GLTV03] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. *IOA User Guide and Reference Manual*. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003. Available at http://theory.lcs.mit.edu/tds/ioa.html.

[GMY+07] Radu Grosu, Sayan Mitra, Pei Ye, Emilia Entcheva, I. V. Ramakrishnan, and Scott A. Smolka. Learning cycle-linear hybrid automata for excitable cells. In *HSCC*, pages 245–258, 2007.

[GNU] GNU. GLPK - GNU linear programming kit. Available from http://www.gnu.org/directory/libs/glpk.html.

[GP05] Antoine Girard and George J. Pappas. Approximation metrics for discrete and continuous systems. In *IEEE Transactions on Automatic Control*, 2005.

[GPS06] Marc Girault, Guillaume Poupard, and Jacques Stern. On the fly authentication and signature schemes based on groups of unknown order. *Journal of Cryptology*, 19(4):463—487, 2006.

[Hen96] 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.

[Her02] Holger Hermanns. *Interactive Markov Chains : The Quest for Quantified Quality*. Springer Berlin / Heidelberg, 2002.

[Hes04] João P. Hespanha. Stochastic hybrid systems: Application to communication networks. In Alur and Pappas [AP04], pages 387 – 401.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 460–483, 1997.

[HK96] Thomas A. Henzinger and Peter W. Kopke. State equivalences for rectangular hybrid automata. In *International Conference on Concurrency Theory (CONCUR'96)*, pages 530–545, 1996.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms*

for the Construction and Analysis of Systems (TACAS'06), volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *ACM Symposium on Theory of Computing*, pages 373–382, 1995.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[HL94] Connie Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time system. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994. IEEE Computer Society Press.

[HLM03] J.P. Hespanha, D. Liberzon, and A.S. Morse. Hysteresis-based switching algorithms for supervisory control of uncertain systems. *Automatica*, 39:263–272, 2003.

[HLS00] J. Hu, J. Lygeros, and S. Sastry. Towards a theory of stochastic hybrid systems. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control, 3rd Int. Workshop (HSCC 2000)*, volume 1790 of *LNCS*, pages 160–173, 2000, 2000.

[HM99] J.P. Hespanha and A. Morse. Stability of switched systems with average dwell-time. In *Proceedings of 38th IEEE Conference on Decision and Control*, pages 2655–2660, 1999.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.

[HT06] João P. Hespanha and Ashish Tiwari, editors. *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*, volume 3927 of *LNCS*. Springer, 2006.

[Int06] SRI International. The PVS bibliography, 2006. http://pvs.csl.sri.com/papers/pvs-bib/pvs-bib.html.

[JS90] C.-C. Jou and S. A. Smolka. Equivalences, congruences and complete approximations for probabilistic processes. In *CONCUR 90*, number 458 in LNCS. Springer-Verlag, 1990.

[Kan] L. Kantorovich. On the transfer of masses. *Doklady Akademii Nauk*, 37(i2):227–229.

[Kha02] H. K. Khalil. *Nonlinear Systems*. Prentice Hall, New Jersey, 3rd edition, 2002.

[KLMG05] D. Kaynar, N. Lynch, S. Mitra, and S. Garland. *TIOA Language*. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2005.

[KLSV03] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time system. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium*, Cancun,Mexico, December 2003.

[KLSV04] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917a, MIT Laboratory for Computer Science, 2004. Available at http://theory.lcs.mit.edu/tds/reflist.html.

[KLSV05] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. URL:http://www.morganclaypool.com/doi/abs/10.2200/S00006ED1V01Y200508CSL001. Also available as Technical Report MIT-LCS-TR-917.

[KN96] Marta Z. Kwiatkowska and Gethin Norman. Probabilistic metric semantics for a simple language with recursion. In *MFCS '96: Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, pages 419–430, London, UK, 1996. Springer-Verlag.

[KNSW04] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In *In Proc. FORMATS/FTRTFT'04*, volume 3253 of *LNCS*, pages 293–308. Springer-Verlag, September 2004.

[Lib03] Daniel Liberzon. *Switching in Systems and Control*. Systems and Control: Foundations and Applications. Birkhauser, Boston, June 2003.

[Lim01] Hongping Lim. Translating timed I/O automata specifications for theorem proving in PVS. Master's thesis, Massachusetts Institute of Technology, February 2001.

[LKLM05] Hongping Lim, Dilsun Kaynar, Nancy Lynch, and Sayan Mitra. Translating timed I/O automata specifications for theorem proving in pvs. In *Proceedings of Formal Modelling and Analysis of Timed Systems (FORMATS'05)*, number 3829 in LNCS, Uppsala, Sweden, September 2005. Springer.

[LL96] Gunter Leeb and Nancy Lynch. Proving safety properties of the steam boiler controller. In Egon Boerger Jean Raymond Abrial and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 11654 of *LNCS*, 1996.

[LLL99] Carolos Livadas, John Lygeros, and Nancy A. Lynch. High-level modeling and analysis of TCAS. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99),Phoenix, Arizona*, pages 115–125, December 1999.

[LMMS98] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 112–121, New York, NY, USA, 1998. ACM Press.

[LMN] Nancy Lynch, Sayan Mitra, and Tina Nolte. Motion coordination using virtual nodes. In *Proceedings of 44th IEEE Conference on Decision and Control (CDC05)*, Seville, Spain, December.

[LPY99] Gerardo Lafferriere, George J. Pappas, and Sergio Yovine. A new class of decidable hybrid systems. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 137–151, London, UK, 1999. Springer-Verlag.

[LS91] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94(1):1–28, 1991.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, August 2003.

[LTS99] John Lygeros, Claire Tomlin, and Shankar Sastry. Controllers for reachability specifications for hybrid systems. In *Automatica*, volume 35, March 1999.

[Lue79] David G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, and Applications*. John Wiley and Sons, Inc., New York, 1979.

[LV96] Nancy Lynch and Frits Vaandrager. Forward and backward simulations - part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[Lyn96a] 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.

[Lyn96b] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[MA03] Sayan Mitra and Myla Archer. Developing strategies for specialized theorem proving about untimes, timed, and hybrid I/O automata. In *STRATA 2003*, Rome, Italy, 2003.

[MA04] Sayan Mitra and Myla Archer. Reusable pvs proof strategies for proving abstraction properties of I/O automata. In *STRATEGIES 2004, IJCAR Workshop on strategies in automated deduction*, Cork, Ireland, July 2004.

[MA05] Sayan Mitra and Myla Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theoretical Computer Science*, 125(2):45–65, 2005.

[Mat74] Mathlab Group. *MACSYMA primer: introductory section*. Cambridge, MA, USA, 1974.

[Meg01] Alexandre Megretski. On automatic search for invariants of hybrid systems. In *American Control Conference*, pages 217–223, Arlington, VA, 2001.

[Mit01] Sayan Mitra. HIOA - a specification language for hybrid Input/Output automata. Master's thesis, Department of Computer Science and Automation, IISc, Indian Institute of Science, Bangalore, 2001.

[ML06] Sayan Mitra and Nancy Lynch. Approximate simulations for task-structured probabilistic I/O automata. In *LICS workshop on Probabilistic Automata and Logics (PAul06)*, Seattle, WA, August 2006.

[ML07] Sayan Mitra and Nancy Lynch. Proving approximate implementation relations for probabilistic I/O automata. *Electronic Notes in Theoretical Computer Science*, 174(8):71–93, 2007.

[MLL06] Sayan Mitra, Daniel Liberzon, and Nancy Lynch. Verifying average dwell time by solving optimization problems. In Hespanha and Tiwari [HT06], pages 476–490. Full version: http://theory.lcs.mit.edu/~mitras/research/hscc06_full.pdf.

[MM01] C. Muñoz and M. Mayero. Real automation in the field. Technical Report NASA/CR-2001-211271 Interim ICASE Report No. 39, ICASE-NASA Langley, ICASE Mail Stop 132C, NASA Langley Research Center, Hampton VA 23681-2199, USA, December 2001.

[MMS05] J. Misra, G. Morrisett, and N. Shankar, editors. *Workshop on The Verification Grand Challenge*, SRI International, Menlo Park, CA, 2005.

[MMT91] 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.

[Mor96] A. S. Morse. Supervisory control of families of linear set-point controllers, part 1: exact matching. *IEEE Transactions on Automatic Control*, 41:1413–1431, 1996.

[MOW04] M. W. Mislove, J. Ouaknine, and J. Worrell. Axioms for probability and nondeterminism. *ENTCS*, 2004.

[MP81] Z. Manna and A. Pnueli. *The Correctness problem in Computer Science*, chapter The temporal framework for concurrent programs. Academic Press, 1981.

[MP03] Oded Maler and Amir Pnueli, editors. *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3-5, 2003, Proceedings*, volume 2623 of *LNCS*. Springer, 2003.

[MT05] Manfred Morari and Lothar Thiele, editors. *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings*, volume 3414 of *LNCS*. Springer, 2005.

[MWLF03] Sayan Mitra, Yong Wang, Nancy Lynch, and Eric Feron. Safety verification of model helicopter controller using hybrid Input/Output automata. In Maler and Pnueli [MP03], pages 343–358.

[MWMW04] D. Pavlovic M. W. Mislove, J. Ouaknine and J. Worrell. Duality for labelled markov processes. In *Proceedings of FOSSACS 04*, volume 2987 of *LNCS*. Springer, 2004.

[NA02] Mikhail Nesterenko and Anish Arora. Stabilization-preserving atomicity refinement. *J. Parallel Distrib. Comput.*, 62(5):766–791, 2002.

[NK93] A. Nerode and W. Kohn. Models for hybrid systems: Automata, topologies, stability. In R. L. Grossman et al., editors, *Hybrid Systems*, volume 736 of *LNCS*, New York, 1993.

[NW03] Toh Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2003.

[Oga97] Katsuhiko Ogata. *Modern control engineering (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[ORR+96] 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 LNCS, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *LNCS*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.

[OS01] S. Owre and N. Shankar. Theory interpretations in PVS. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 2001.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, 1993.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[Qua] Quanser. Control challenges: 3 DOF helicopter http://www.quanser.com/english/html/products/fs_product_challenge.asp?lang_code=english&pcat_code=exp-spe&prod_code=S1-3dofheli&tmpl=1.

[Rac91] Svetlozar T. Rachev. *Probability metrics and the stability of stochastic models.* John Wiley & Sons, 1991.

[Seg95a] R. Segala. A compositional trace-based semantics for probabilistic automata. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR ' 95)*, volume 962 of *LNCS*, pages 234–248, Philadelphia, PA, USA, August 1995.

[Seg95b] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1995.

[Seg96] R. Segala. Testing probabilistic automata. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR ' 96)*, volume 1119 of *LNCS*, pages 299–314, Pisa, Italy, August 1996.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[SSM04] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Constructing invariants for hybrid systems. In Alur and Pappas [AP04], pages 539–554.

[Sta03] E. Stark. On behavior equivalence for probabilistic I/O automata and its relationship to probabilistic bisimulation. *Journal of Automata, Languages, and Combinatorics*, 8(2):361–395, 2003.

[SvdS05] Stefan Strubbe and A. J. van der Schaft. Bisimulation for communicating piecewise deterministic markov processes (cpdps). In Morari and Thiele [MT05], pages 623–639.

[TEM07] Tempo toolset, version 0.1.9 beta, August 2007. `http://www.veromodo.com/tempo/`.

[TG02] Claire Tomlin and Mark R. Greenstreet, editors. *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002, Stanford, CA, USA, March 25-27, 2002, Proceedings*, volume 2289 of *LNCS*. Springer, 2002.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[TK02] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In Tomlin and Greenstreet [TG02], pages 465–478.

[TPL02] Paulo Tabuada, George J. Pappas, and Pedro U. Lima. Composing abstractions of hybrid systems. In Tomlin and Greenstreet [TG02], pages 436–450.

[UL07] Shinya Umeno and Nancy A. Lynch. Safety verification of an aircraft landing protocol: A refinement approach. In *HSCC*, pages 557–572, 2007.

[vBMOW03] F. van Breugel, M. Mislove, J. Ouaknine, and J. B. Worrell. An intrinsic characterization of approximate probabilistic bisimilarity. In *Proceedings of FOSSACS 03*, LNCS. Springer, 2003.

[vBMOW05] F. van Breugel, M. W. Mislove, J. Ouaknine, and J. Worrell. Domain theory, testing and simulation for labelled markov processes. *Theoretical Computer Science*, 2005.

[vBW01a] Franck van Breugel and James Worrell. An algorithm for quantitative verification of probabilistic transition systems. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 336–350, London, UK, 2001. Springer-Verlag.

[vBW01b] Franck van Breugel and James Worrell. Towards quantitative verification of probabilistic transition systems. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 421–432, London, UK, 2001. Springer-Verlag.

[VCL06] L. Vu, D. Chatterjee, and D. Liberzon. Input-to-state stability of switched systems and switching adaptive control. *Automatica*, 2006.

[vdSS00] A. van der Schaft and H. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Springer, London, 2000.

[Vit02] Ben Di Vito. A PVS prover strategy package for common manipulations. Technical report, NASA Langley Research Center, 2002.

[Wil90] H.P. Williams. *Model building in mathematical programming.* J. Wiley, New York, 1990. third edition.

[WIM+02] Yong Wang, Masha Ishutkina, Sayan Mitra, Nancy A. Lynch, and Eric Feron. Design of Supervisory Safety Control for 3DOF Helicopter using Hybrid I/O Automata, 2002. http://gewurtz.mit.edu/ishut/darpa_sec_mit/papers/quanser.ps.

[WL96] 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.

[WLD95] 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 Workshop on Verification and Control of Hybrid Systems)*, volume 1066 of *LNCS*, pages 101–113. Springer-Verlag, October 1995.

[ZHYM00] G. Zhai, B. Hu, K. Yasuda, and A. Michel. Stability analysis of switched systems with stable and unstable subsystems: An average dwell time approach. In *Proceedings of the 2000 American Control Conference (2000)*, chicago,illinois, 2000.