UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Verification and Validation of Hybrid Systems

Permalink
https://escholarship.org/uc/item/7n3634tx

Author
Jin, Xiaoqing

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Verification and Validation of Hybrid Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xiaoqing Jin

August 2013

Dissertation Committee:

Professor Gianfranco Ciardo, Chairperson
Professor Eamonn Keogh
Professor Iulian Neamtiu
Professor Walid Najjar

Copyright by
Xiaoqing Jin
2013

The Dissertation of Xiaoqing Jin is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Completing my Ph.D degree is the most challenging commitment I have made in my life. My doctoral journey would not be memorable without the guidance and company of many that I have met along the way. They are the ones who endured with me through the best and the worst of times.

Foremost, I am grateful and thankful to my advisor Prof. Gianfranco Ciardo, who introduced me to this research topic through his passion, enthusiasm, vision, and encouragement to reach this pinnacle. Through discipline and patience, he gave me not only the knowledge but also the techniques that would help steer our research in the right direction. Without his continuous support and guidance, this thesis would have been impossible.

I would also like to thank my other committee members Prof. Eamonn Keogh, Prof. Iulian Neamtiu, and Prof. Najjar Walid for their patience and suggestions. Special thanks to Prof. Rajiv Gupta who introduced me to the research of combining software testing and verification. They have been supportive of my thesis through the years since my candidacy.

I would like to express my sincere gratitude to my colleagues Dr. Alexandre Donzé, Dr. Jyotirmoy V. Deshmukh, Dr. Koichi Ueda, Dr. James Kapinski, and Dr. Ken Butts, for their professional suggestions and ideas and also for providing the industrial model used in this thesis.

My fellow members of the logic and stochastic verification lab: Yang Zhao, Malcolm Mumme, Yousra Lembachar, Xin He, Jude Ezeobiejesi, Hind Al Hakami, Mengyi Guo, Daimeng Wang, and Lei Wang all deserve my special thanks for the brainstorming discussions and the invaluable assistance with the completed projects.

Special thanks to all my lovely friends, Chong Ding, Curtis Yu, Di Zhang, Dorian Perkins, Jacob Relles, Li Yan, Masoud Akhoondi, Pamela Bhattacharya, Stefan Lukesch, Toni Poelzleitner, and Zhe Wu for their friendship and company that have made this journey a wonderful and an unforgettable experience.

My sincere thanks to UCR dancesport. My coach Fook Tham and Virginia Barbour help me to balance my Ph.D life with learning elegant dances and competitive competitions. They showed me a getaway from theories and graphs, and taught me to be persistent and dedicated to my commitments. I also want to thank Andrew Nguyen, Ashley Thompson, Danni Lin, Hoang Nguyen, Ivan Perez, Jennifer Huey, Jingyi Liu, Ryan Pletcher, Shujuan Wang, and Xitlali Tapia for all these years' amazing memories.

Last but not the least, I am deeply thankful to my parents, Jingzhong Jin and Youlan Ji, who are always there by my side to share tears and joy. Their trust and support has always been a source of courage and strength that has helped me through this journey.

To my parents and friends, who always have faith in my endeavors.

ABSTRACT OF THE DISSERTATION

Verification and Validation of Hybrid Systems

by

Xiaoqing Jin

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2013
Professor Gianfranco Ciardo, Chairperson

Hybrid systems tightly integrate software-based discrete control systems and continuous physical phenomena. Better methods and tools for design, modeling, and analysis are necessary as these systems become more complex, powerful, and prevalent in our daily lives. There are two main approaches to model hybrid systems. One is to discretize them into fully discrete systems. We focus on two discrete models: a network of communicating FSMs (NCFSM) and event-condition-action (ECA) rules. In this dissertation, we make several contributions to verifying discrete systems. First, for an NCFSM, we symbolically encode and verify the properties of livelocks, strong connectedness, and dead transitions. We also design a symbolic equivalence checker to automatically generate test cases for fault-based testing. Second, for ECA rules, we verify both termination and confluence properties and then provide the first practical experimental results for the confluence property.

The second modeling method is to keep the continuous dynamics by using hybrid automata or block diagrams. Most verification problems for hybrid automata are undecidable even under severe limitations. However, using block diagrams, such as Simulink, researchers model a hybrid system as an input-output signals mapping box.

This approach has been widely adopted by industry due to its scalability. Combining simulation and temporal logic, the validation technique is able to check whether the temporal behavior of the system meets a set of requirements. One significant challenge to the formal validation is that system requirements are often imprecise, non-modular, evolving, or even simply unknown. In this dissertation, we make the several contributions to tackle the requirement defects. First, we compare the performance of the two existing falsification engines. Second, we design a requirement mining framework, an instance of counterexample-guided inductive synthesis, which is able to mine formal requirements from a closed-loop model. Third, we observe the importance of the monotonicity of formulas for synthesis and use a satisfiability modulo theories (SMT) solver to prove this property. Fourth, we propose weighted temporal logics to improve the performance of this mining framework. This framework has the following two applications: mined requirements can be used to validate future modifications of the model and enhance understanding of legacy models; the framework can also guide the process of bug-finding through simulations. We present two case studies for requirement mining: a simple automobile transmission controller and an industrial airpath control engine.

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**

1.1 Challenges . . . 3
1.2 Modeling and verification . . . 6
1.3 Verification or testing . . . 9
1.4 Verification or validation . . . 11
1.5 Requirement defects . . . 14
1.6 Contributions . . . 16
1.7 Organization . . . 18

2 Models for Hybrid Systems **19**

2.1 Overview of dynamic systems . . . 19
2.2 Modeling discrete systems . . . 24
2.2.1 A network of communicating FSMs . . . 25
2.2.1.1 CFSM . . . 25
2.2.1.2 NCFSM . . . 28
2.2.2 ECA rules . . . 32
2.2.2.1 ECA rule syntax . . . 32
2.2.2.2 ECA rule semantics . . . 35
2.2.2.3 ECA rules for a smart home . . . 37
2.3 Modeling hybrid systems . . . 40
2.3.1 Hybrid automata . . . 41
2.3.1.1 Hybrid automata semantics . . . 43
2.3.1.2 A thermostat example . . . 45
2.3.2 Signals and hybrid systems . . . 47
2.3.2.1 A four-speed automatic transmission example . . . 48
2.4 Conclusion . . . 51

3 Symbolic Verification of Discrete Systems **53**

3.1 Decision diagrams . . . 54
3.2 LTL and CTL . . . 56
3.3 Symbolic verification of a network of communicating FSMs . . . 60
3.3.1 Symbolic encoding of an NCFSM . . . 60

3.3.2 Reachability analysis for an NCFSM . . . 63
3.3.3 Symbolic observable product machine generation . . . 65
3.3.4 Symbolic livelocks verification . . . 69
3.3.5 Symbolic strong connectedness verification . . . 71
3.3.6 Symbolic dead transition verification . . . 71
3.3.7 Symbolic equivalence verification and its application . . . 72
3.3.8 Experimental results . . . 79
3.4 Petri nets . . . 82
3.5 Symbolic verification of a set of ECA rules . . . 84
3.5.1 Transforming a set of ECA rules into a PN . . . 86
3.5.1.1 Occurring phase . . . 88
3.5.1.2 Triggering phase . . . 88
3.5.1.3 Performing phase . . . 89
3.5.1.4 ECA rules to PN translation algorithms . . . 91
3.5.2 Verifying properties . . . 93
3.5.2.1 Termination . . . 94
3.5.2.2 Confluence . . . 95
3.5.3 Experimental results . . . 98
3.6 Conclusion . . . 102

4 Verification and Validation of Hybrid Systems 103
4.1 Verification of hybrid automata . . . 104
4.2 STL and MTL . . . 107
4.3 Quantitative semantics . . . 109
4.4 Falsification of hybrid systems . . . 114
4.5 Experimental results . . . 116
4.6 Conclusion . . . 120

5 Requirement Mining for Hybrid Systems 122
5.1 Parametric STL . . . 124
5.2 Weighted STL and parametric weighted STL . . . 125
5.3 Requirement mining framework . . . 128
5.4 Revisiting the falsification problem . . . 130
5.5 Parameter synthesis . . . 131
5.5.1 Satisfaction monotonicity . . . 134
5.6 Case studies . . . 138
5.6.1 Automatic transmission model . . . 139
5.6.2 Industrial diesel engine model . . . 146
5.7 Conclusion . . . 148

6 Summary and Future Research 150
6.1 Summary . . . 150
6.2 Future research . . . 152

Bibliography 156

List of Figures

1.1 The traditional V design process. The whole V process iterates until the release of the final product. Both of the software (SW) and hardware (HW) have to go through design, implementation, verification, and testing. 9

1.2 The MBD V design process. Rapid prototyping is also known as model-in-the-loop simulation. SIL (software-in-the-loop) incorporates the production software code into the plant model and performs offline simulation while HIL (hardware-in-the-loop) combines the actual physical hardware with the plant model and performs real-time simulation. 12

2.1 A hybrid system diagram. The system interacts with the physical world, represented as a plant which is guided by ordinary differential equations (ODE), and a discrete cyber world, represented as a controller which is guided by a state transition system. Sensors monitor the plant and provide inputs to the controller through an analog-to-digital (A/D) converter. The controller outputs go through a digital-to-analog (D/A) converter and then drive the actuators. 20

2.2 The simplified gear shift logic of a four-speed transmission. 23

2.3 An example of NCFSM $M_{ex} = (M_1, M_2)$ where $\mathcal{S}_1 = \mathcal{S}_2 = \{1, 2\}$, $\mathcal{X}_1 = \{a, b\}$, $\mathcal{X}_2 = \{c, d\}$, $\mathcal{Y}_1 = \{c, x, y\}$, $\mathcal{Y}_1 = \{a, b, x\}$, and $s_1 = s_2 = 1$. M_1 and M_2 communicate with each other through a global channel with buffer size 1. Edge b/c means that, in order to take the transition, the precondition of the state is the presence of an input symbol b and the postcondition is the generation of an output symbol c. If CFSMs M_1 and M_2 are at state 1, the presence of symbol a makes M_1 take the transition a/x and move to state 2 with an output symbol x. 26

2.4 The observable product machine M_{obs} for NCFSM M_{ex} in Figure 2.3 where $\mathcal{Z}_{int} = \{a, b, c\}$, $\mathcal{X}_{ext} = \{a, b, c, d\}$, $\mathcal{Y}_{ext} = \{x, y\}$, and $\mathcal{S}_{st} = \{(\epsilon, 1, 1), (\epsilon, 2, 1), (\epsilon, 2, 2)\} \subset \{\epsilon\} \times \mathcal{S}_1 \times \mathcal{S}_2$, and $\mathbf{s}_{init} = (\epsilon, 1, 1)$. 31

2.5 The designed syntax of ECA rules. "/" is integer division. "*number*" is a constant $\in \mathbb{N}$. The superscript "+" indicates the match-one-or-more quantifier and "*" the match-zero-or-more quantifier. 33

2.6 ECA rules for the automatic light control subsystem of a smart home for senior housing. 38

2.7 The HA for a thermostat controlling the indoor room temperature using a heater which is represented as a continuous variable x. A heating device will be automatically turned on if the room temperature, x, drops below 20°C and be turned off once x reaches 25°C. The temperature will fall according to $\dot{x} = -Kx$ without the heater and rises according to $\dot{x} = K(H - x)$, where K and H are constants for thermodynamics. 45
2.8 One simulation result of the thermostat in Figure 2.7 in the duration of 24 hours with initial room temperature $x_0 = 21$ and the heater is on. The values chosen for the constants are $H = 32$ and $K = 0.2$. All switching points are marked by dark circles. 47
2.9 The closed-loop Simulink model of an automatic transmission controller. The inputs to the model are the throttle position and the brake torque. The outputs are gear position, engine speed in rpm, and the vehicle speed in mph. The discrete gear shift logic is done using Stateflow. The continuous plant model in *Vehicle* modular is constructed using continuous block in Simulink. 49
2.10 Falsifying trace for the automatic transmission controller and the requirement that RPM never goes beyond 4500 rpm or speed beyond 120 mph. 50

3.1 A simple example of system evolution for M_{ex} in Figure 2.3. Figures (a), (c), (e), and (g) are MDDs encoding states while (b), (d), and (f) are MDDs encoding transitions. Shaded arrows mean to apply transitions to states and slim arrows point to the result. 60
3.2 A demonstration of applying fully-reduced and identity-reduced rules. Figure (a) encodes transitions in quasi-reduced rule. Applying identity-reduced rule on variable w'_1 in (a) results in (b) and applying fully-reduced rule on variable w_1 in (b) results in (c). 61
3.3 QFI-reduced MDDs encoding transition $\mathcal{T}$ of M_{ex} in Figure 2.3, where (a) for $\mathcal{T}_1$, (b) for $\mathcal{T}_2$, and (c) for $\mathcal{T}_\beta$. 62
3.4 The BFS and saturation algorithms for reachability analysis. 64
3.5 Composition operation on the transition relation 66
3.6 The algorithm for the operation to combine the effect of sequential firing transitions into one transition. 67
3.7 The BFS and saturation algorithms for computing transition transitive closure. 68
3.8 A simple variation of NCFSM M_{ex} in Figure 2.3. The small modification is that local transition $1\,[M_1, a/x\rangle\, 2$ is changed to $1\,[M_1, a/d\rangle\, 2$. The resulting new NCFSM contains a livelock when giving external input c at the initial state. 69
3.9 A fully masked mutant of M_{ex} in Figure 2.3, $\overline{M_{ex}}^{(1)} = (M_1, M_2)$. The introduced modification is that local transition $2\,[M_1, b/c\rangle\, 1$ of M_{ex} is changed to $2\,[M_1, b/c\rangle\, 2$ of $\overline{M_{ex}}^{(1)}$. 72
3.10 A partially masked mutant of M_{ex} in Figure 2.3, $\overline{M_{ex}}^{(2)} = (M_1, M_2)$. The introduced modification is that local transition $2\,[M_1, b/c\rangle\, 1$ of M_{ex} is changed to $1\,[M_1, b/d\rangle\, 2$ of $\overline{M_{ex}}^{(2)}$. 73
3.11 Encoding distance function f_{dis}, next-state-pair function $\mathcal{G}$, and distinguishable-state-pair $\mathcal{D}$ of M_{ex} in Figure 2.3 and $\overline{M_{ex}}^{(2)}$ in Figure 3.10. 76
3.12 The algorithms for the *PairRelProd* operator and the test generation. . 77

3.13 The PN for ECA rules in Figure 2.6. 87
3.14 Transforming ECA rules into a PN: $a -[k]\rightarrow b$ means "an arc from a to b with cardinality k; $a -[k]\!-\!\circ\, b$ means "an inhibitor arc from a to b with cardinality k. 90
3.15 Processing **par**. 91
3.16 Processing **seq**. 91
3.17 The initialization phase for the smart home example. 93
3.18 Algorithms to verify the termination property. 95
3.19 Explicit algorithms to verify the confluence property. 97
3.20 Fully symbolic algorithm to verify the confluence property. 98
3.21 A termination counterexample (related to rules r_4 to r_7). 100
3.22 A confluence counterexample (related to rules r_8 and r_9). 100

4.1 Fixed-point algorithms for the forward and backward analysis on state regions of HA. 105
4.2 The initialized RHA for the thermostat in Figure 2.7. 106
4.3 With respect to STL formula $\varphi = \mathsf{G}(\mathbf{x} < 15 \wedge \mathbf{x} > -15)$, using Boolean semantics, signals x_1 and x_2 will be abstracted to the same Boolean signal, straight true for the whole time domain. Thus, both of them satisfy φ. However, it is straightforward that x_1 is closer to the threshold to violate φ, comparing to x_2. Unfortunately, Boolean semantics is unable to recognize this difference. 110
4.4 After introduced some random noise to the system generating x_1 in Figure 4.3, the resulting signal violates the same STL formula $\varphi = \mathsf{G}(\mathbf{x} < 15 \wedge \mathbf{x} > -15)$. 111
4.5 The closed-loop Simulink model of a thermostat system. This model is more advanced than the one in Section 2.3.1.2. It details the physical plant in the House block that allows designers to specify thermal properties, such as the materials of walls and windows, the geometry of the house, the temperature of the heater flow. Moreover, the model also estimates the heating cost for this specific house. The inputs of the model are the indoor temperature threshold for the thermostat and the initial outdoor temperature that is able to affect the thermal resistance of the whole system. 116
4.6 A counterexample of the formula with the condition that the predefined heat flow temperature is 50°C, $\varphi^{50}_{\mathtt{eff}} = \mathsf{G}((\mathit{TempDiff} > 2.5) \Rightarrow (\mathsf{F}_{[0,0.25]}(\mathsf{G}_{[0,0.25]}(\mathit{TempDiff} < 1))))$. 118
4.7 A counterexample of the formula with the condition that the predefined heat flow temperature is 80°C, $\varphi^{80}_{\mathtt{eff}} = \mathsf{G}((\mathit{TempDiff} > 2.5) \Rightarrow (\mathsf{F}_{[0,0.25]}(\mathsf{G}_{[0,0.25]}(\mathit{TempDiff} < 1))))$. 119
4.8 Simulation results of 100 runs against $\varphi_{\mathtt{cost}} = \mathsf{G}(\mathit{HeatCost} < 8)$. 119

5.1 Flowchart of the requirement mining framework. This framework is an instance of a counterexample-guided inductive synthesis procedure. After given a PSTL or PWSTL template requirement formula, the synthesis engine will find an appropriate valuation for parameters in the parametric formula to form a STL or WSTL formula as a candidate requirement. The candidate requirement will be used by the falsification engine to search a counterexample to falsify the formula. If found, the counterexample trace, together with previous traces, is passed to the synthesis engine to start the next iteration. 128
5.2 The FindParam algorithm for synthesizing a candidate valuation from a given template requirement in PWSTL. 132
5.3 Validity domain of a simple formula for a trace $\mathbf{x}$ obtained from the automatic transmission model. The FindParam algorithm will return valuation v_1 (resp. v_2) depending on whether the time (resp. scale) parameter is optimized first. The contour lines are isolines for the satisfaction function ρ. 133
5.4 The robustness satisfaction surface of PSTL $\varphi_{\texttt{gs}}$ with $v(\tau) = 7$ consists of 100 sample points from the input domain. The inputs are the position of throttle `throttle` and the initial time to start to brake $t_{\texttt{brake}}$ with the range of `throttle` in [0,100]% and [0,20] seconds for $t_{\texttt{brake}}$. 145
5.5 The robustness satisfaction surface of PWSTL $\varphi_{\texttt{gs_w}}$ with $v(\tau) = 7$ and $\omega = 1000$ consists of 100 sample points from the input domain. The inputs are the position of throttle `throttle` and the initial time to start to brake t with the range of `throttle` in [0,100]% and [0,20] seconds for t. 145
5.6 The simulation trace (in blue) for the signal $\mathbf{x}$ denoting the difference between the intake manifold pressure and its reference value found when mining $\varphi_{\texttt{settling_time}}(\tau, \pi)$ displays unstable behavior. The maximum error threshold that we expected to mine is depicted in red. The ideal $\mathbf{x}$ signal is in green. The values along the axes have been suppressed for proprietary reasons. We remark that the actual values are irrelevant and the intention is to show an oscillating behavior arising from a real bug in the design. 147

List of Tables

3.1 Test case generation results (time in seconds and memory in MB). . . . 80
3.2 Results of verifying the ECA rules for a smart home in Figure 2.6. . . . 99

4.1 The falsification results for the thermostat system in Figure 4.5. 117

5.1 Proving monotonicity with an SMT solver. Time is measured in seconds. 138
5.2 Results on mining requirements for the automatic transmission control model. We compare runs of requirement mining algorithm using either S-Taliro or Breach as falsifiers. In each case and for each template formula, we give the parameters valuations found, the time spent in falsification, and in parameter synthesis, the number of simulations, and the averaged time spent computing the quantitative satisfaction of the formula by one trace. Time is measured in seconds. 139
5.3 Results on STL- and WSTL-based mining requirements for the automatic transmission control model using Breach. In each case and for each template formula, we give the chosen weight vector, the parameters valuations found, the time spent in falsification and in parameter synthesis, the number of simulations, and the averaged time spent computing the quantitative satisfaction of the formula by one trace. Time is measured in seconds. 143

Chapter 1

Introduction

Since the late 1980s, information technology (IT) has gone through several significant revolutions, from supercomputers to personal computers, to portable and embedded systems. Coupled with sensor networks, embedded systems, such as those in portable devices [208], automobiles [167, 209], aerospace [82], manufacturing [175], smart grids [191], telecommunication systems [153], civil infrastructure [206], and healthcare [147], have had tremendous impacts on every aspects of our daily lives. These advances have significantly reduced reliance on manual labor by introducing accurate high performance automation systems. Furthermore, advancements in computing and communication technologies have dramatically increased their capabilities over the years, and at the same time reduced their cost and size. Linked together through networks, these systems become more prevalent and form a new concept: cyber-physical systems (CPSs) [25]. CPSs are defined as the integration of computational communication systems and physical processes [140] with more emphasis on the link between physical processes (physical side) and the information processing (cyber side). Research on CPSs has been extensively supported by many governmental research authorities [77, 165] and

has become a subject of great interest in the past decade. CPSs provide us with the ability to observe and modify the physical environment achieving stability, reliability, robustness, and efficiency. Hybrid systems are a broad interdisciplinary domain which links many research domains such as real-time systems, wireless sensor networks, control theory, model-based development, multi-objective optimization, and formal verification [175]. Also, a great impetus is coming from the increasing demand for green energy, better healthcare systems, more efficient and safer transportation systems, and space exploration. This impetus pushes engineers to design and develop more complex and larger scaled hybrid systems. Already in development, technologies such as autonomous vehicles [97], smart homes [76], and robotic surgery [51] may become ubiquitous in the near future.

One fundamental characteristic of these pervasive systems is the presence of both continuous and discrete dynamic behaviors [8]. Hybrid systems are the integration of continuous physical processes and discrete software systems. At a higher level of abstraction, a hybrid system "jumps" between different control modes based on conditions of the system and its physical environment. This discrete "jump" behavior is the final observable result, produced by the control software inside the system. The control software is responsible for collecting, processing, and sharing information from both the system and its environment. In order to adapt or to change the environment the controller then makes a decision about which mode the system should be operated in. At a lower level of abstraction, both the system and its environment are governed by naturally continuous concrete physical laws, such as Newton's laws of motion or thermodynamics. Take an automobile as an example. One of the variables used to describe the system state, such as speed, is continuous. Depending on the speed, the automatic transmission control system decides between shifting to a higher/lower gear or staying

in the current gear. Each gear represents a different control mode of the vehicle and is discrete. Briefly, a hybrid system consists of a discrete controller, such as the gear shifting logic of the transmission system, and a continuous physical environment, such as the speed and the acceleration of a vehicle.

1.1 Challenges

Due to rapid innovation in hardware technologies, modern hybrid systems have become multi-layered, multi-functional, and increasingly large and complex. The following features can be observed in typical hybrid systems [140]:

- **The integration of functional sensors and actuators using networks.** At a low level, the mechanism that samples a physical system affects the amount of data to be transferred. The ability to handle a large volume of data to facilitate information collecting and sharing hinges on the hardware architecture and the network infrastructure. Moreover, the methods to handle network delay, packet loss, and data rate constraints largely affect the stability and sometimes even the correctness of the hybrid system. At a high level, the topology and network protocol can affect system performance. The system not only processes informations from sensors but also interacts with its physical environment through actuators.

- **Having one or more processing units.** Since the 1970s, BMW has used microprocessors and engine sensors to enhance the performance and efficiency of its vehicles. A recent BMW vehicle has more than 70 networked microprocessors. In the avionic industry, microprocessors are used to increase operational safety. Boeing's 777 jet airliner has more than 1,200 networked microprocessors. These microprocessors read data from multiple sensors, calculate, interpret, and compare

certain metrics before adjusting actuators to maintain system performance.

- **Being reactive and adaptive to the environment.** Safety is critical in systems which run indefinitely and in situations where suspending and rebooting are not feasible. Moreover, time constraints, such as a maximum response time, may be required. Sometimes the performance resulting from responding as soon as possible is unacceptable.

- **Heterogeneous architectures.** Hybrid systems often have mixed system architectures consisting of both hardware and software. Thus, the system must handle both digital and analog signals as well as manage the resulting accuracy and precision problems [211].

- **High concurrency.** Almost all components are connected through an internal network and most of the physical processes occur simultaneously. Multiple electronic control units (ECUs) share the data from various sensors. Moreover, sensors and actuators are normally deployed on a distributed network. Thus, a high concurrency between control and computation of the system is imperative for both correctness and performance.

- **Severely limited resources.** Due to constraints on the physical environment or the cost budget, a hybrid system normally operates with severely limited resources, such as network bandwidth, memory, power supply, and number of available registers, For example, a controller area network (CAN) bus, which is a standardized message-based protocol for automotive applications, only allows up to 64 bytes of data payload per frame [192].

All these features make the design of a hybrid system complicated. First, many trade-offs have to be made in order to operate with severely limited physical resources. Second, reactive systems are quite sensitive to timing, requiring both hardware and software to meet hard deadlines. Third, high concurrency, restricted timing requirements, and heterogeneous architectures can cause difficulties during the modeling, testing, and validation stages of the system. Finally, the non-deterministic nature of environmental processes results in the unpredictability of the system behavior, regardless of the precision of the computation or sensors.

The significant social and economic impact of these systems calls for higher reliability and predictability in designs [141], especially for life-critical systems such as assisted living, automotive, or aerospace systems. Without consideration, mild malfunctions, including infrequent and bizarre circuit glitches, could cause disasters. For example, the crash of a Korean Air Boeing 747 in Guam in 1977 was caused by an insufficient minimum safe altitude warning (MSAW) system. The system failed to detect that the altitude of the aircraft was below the safe limitation and caused 228 casualties out of 254 people on board [103]. Moreover, it is important to assure that all possible system behaviors are within legal system's designed manners. Failure to take this into account may result in loss of human lives. According to an FDA study in the mid-90s [123], about 7.7% of the deaths and injures caused by medical device failures are caused by faulty software, and defects introduced during software maintenance are blamed for almost 80% of the incidents. The most well-known incident is related to Therac-25, a radiation therapy machine. In rare circumstances, Therac-25 delivered approximately 100 times the intended dose to patients due to a race condition between concurrent tasks that caused the key components of the system to fail to rotate into their correct positions [142]. At least three patients died because of this software controlling error.

A more recent example is the premature failure of the demonstration of autonomous rendezvous technology (DART) spacecraft in 2005. Repeated excessive thruster firings in response to incorrect data from biased navigational system estimates resulted in the failure of this $95 million project [120].

Numerous examples evidence the dangerous and catastrophic side of these powerful systems. Societies, industries, and government institutions must consider any adverse outcome unacceptable for these life critical products and urge industries to constantly strive for safety and perfection. Stricter regulations and higher standards are made for industries of life critical products, such as IEC 61508 [121] for general electric and electronic industries, AS/EN 9100 [1] for aerospace industries, and ISO 26262 [122] for automotive industries. However, these regulations and standards only recommend extensive use of traditional technologies, such as testing and simulation. Developing new technologies to better design, model, test, and verify hybrid systems is a major challenge for both academia and industry.

1.2 Modeling and verification

Along with the emerging of hybrid systems of the late 1980s, researchers started to design formal methods for modeling, specification, and verification of hybrid systems. One solution is to discretize the continuous variables and analyze the resulting fully discrete systems, since the computational subsystems of a hybrid system are normally modeled as discrete transition systems [110]. This avoids analysis in the continuous domain and paves the way for using automata theory and discrete temporal logic to model and analyze hybrid systems. Petri nets (PNs) [161] and event-condition-action (ECA) rules [156] are both used to model non-deterministic and complex discrete

systems. Automata-based methods include finite state machines (FSMs), communicating FSMs (CFSMs), networks of CFSMs (NCFSMs) [115], and timed automata [9, 10]. Particularly, timed automata provide a solution to model continuous time as an integer clock under the restriction that clock values can only be assigned to or compared with constants. The equivalent to the automata-theoretic approach, model checking [74], a logic-based method, analyzes the resulting discrete transition systems using temporal logics such as computational tree logic (CTL) [69] or linear temporal logic (LTL) [170]. This formal verification technique is useful for analyzing both hardware and software systems [67]. Most importantly, this technique is able to provide valuable counterexamples to debug the system under inspection when it fails to preserve certain properties [214].

Model checking techniques can be classified as either explicit or symbolic. Famous tools, such as SPIN [118], Murphi [193], and SPOT [87], enumerate the state space of the system explicitly and search the state transition graph to verify critical properties. However, all these model checker tools have to tackle a formidable challenge, the state explosion problem [202]. Sometimes, the state space of the system is too large to be constructed and searched. Normally, explicit methods require a considerable amount of memory and time to perform the analysis [181]. On the other hand, in order to save memory, symbolic strategies choose to implicitly encode transition relations and sets of states using decision diagrams, such as *binary decision diagrams* (BDDs) [47], *multi-way decision diagrams* (MDDs) [130], *algebraic decision diagrams* (ADDs) [135], and *edge-valued multi-way decision diagrams* (EVMDDs) [63]. Symbolic encodings mitigate the state explosion problem by encoding large sets of states in a relatively small and compact logic representation. Also, algorithms to manipulate sets are efficient in symbolic encodings. Thus, symbolic model checkers [157] when introduced in early 90's, they were considered one of the biggest breakthroughs to tackle the state explosion problem.

Symbolic model checkers, such as CadenceSMV [158], NuSMV [65], SMART [62], and VIS [102] have shown their success in scaling to large state spaces.

The benefit of discretization is the ability to use results from existing research. However, modeling the complex events and conditions of hybrid systems, as well as the concurrent and non-deterministic environment remains challenging, even after discretization. Also, the modeling language has to be expressive, flexible, and extensible enough to reason about interesting properties. Although discretization alleviates the stress of analyzing a hybrid system in an infinitely continuous domain, it still has many weaknesses. Intricate concurrency and non-determinism still drive the analysis to difficulty situations even in a discrete domain, which is safer. Discretization provides a trade-off between complexity and precision, sometimes even at the cost of correctness. Thus, it is important to have a generalized formal model that faithfully represents both discrete and continuous processes, such as hybrid automata (HA).

HA enhance FSMs with a finite set of continuous variables that are used in differential equations that describe physical phenomena. This combination results in a powerful and expressive formalism for both modeling and analyzing a hybrid system. However, verification of HA has been shown to be extremely difficult [14]. Most the problems are undecidable, even under severely restricted assumptions [8].

Although model checking is successful in the verification of discrete systems, the methodology cannot be directly applied to hybrid systems whose state spaces are generally infinite. Thus, researchers extended the syntax of temporal logic in order to reason about the continuous signals of hybrid systems, resulting in metric temporal logic (MTL) [13, 132] and signal temporal logic (STL) [152]. These logics have sophisticated expressive power, but the decidability problem of these logics is intrinsically NP-hard [14].

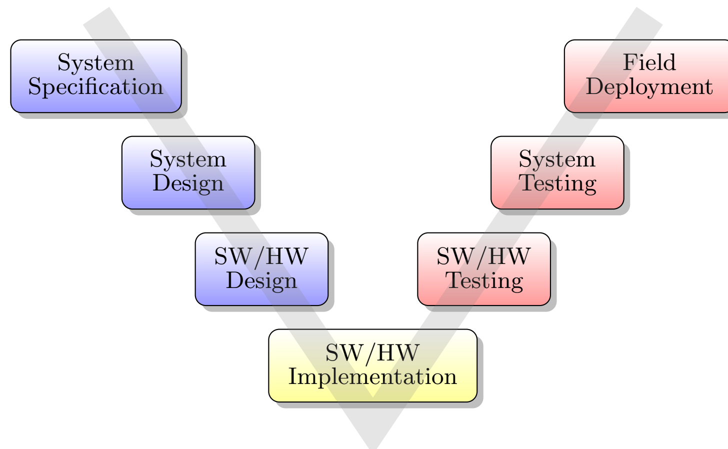

Figure 1.1: The traditional V design process. The whole V process iterates until the release of the final product. Both of the software (SW) and hardware (HW) have to go through design, implementation, verification, and testing.

1.3 Verification or testing

For better quality designs, industries adopt the traditional waterfall "V" shape design process shown in Figure 1.1. At the bottom of the V process is the hardware and software implementation stage. The left edge of the V represents the design process for the system specification, the whole architecture, and each subsystem. The right edge shows the reliance on test and verification, including software and hardware testing, integrated system testing, as well as field deployment and system calibration. Industries are also regulated by higher standards for life-critical products. For example, as already mentioned general electric and electronic industries have to comply with IEC 61508 [121], aerospace industries have to follow AS/EN 9100 [1], and automotive industries have to abide by ISO 26262 [122]. Although these standards provide guidelines for the entire design and development process, with regard to safety and quality control and for risk and failure reduction, they only suggest structural coverage as a metric for testing.

Although exhaustive testing is hardly reasonable due to application complexity and deadlines related to time-to-market, testing still consumes a tremendous amount

of engineering work compared to design. During the system implementation life cycle, unit testing, integration testing, system testing, and regression testing are normally required [122, 154] in different stages. Unit testing is done at the coding stage to evaluate each individual unit. When units are integrated into subsystems or large components, integration testing is performed to make sure that each unit cooperates correctly with others in order to achieve a set of functionalities as a whole. Eventually, engineers will test the whole system to evaluate whether the implemented system meets the desired performance and specifications. Three often suggested and used code-based coverage metrics are statement coverage, branch coverage, and modified condition/decision coverage (MCDC) [54, 122, 154]. Engineers try to get higher syntactic coverage when testing and hope that it will find sufficient bugs to assure quality. However, pure structural coverage only targets the discrete events while generated test cases can only cover a small portion of the potentially infinite state space. Another severe limitation of testing is inherited from the nature of hybrid systems, which constantly interact with and respond to the environment. Since we are not capable of fully controlling the highly concurrent, non-deterministic environment, it is difficult to provide and simulate sensor data for various environmental conditions.

Although the traditional V process has been proven successful and helpful in practice, it leaves all testing and verification until the implementation of the actual system is finished, as shown in Figure 1.1. Bugs introduced during implementation are relatively easy and cheap to fix and only the testing edge of the V process must be re-applied. The worst case is having errors and bugs in the initial specifications as this requires restarting the entire V process which is expensive and time-consuming. Fixing a bug in the design is also costly and challenging, since it may require software redesign, and, at times, even hardware redesign. This situation is quite common in practice which

motivates new design methodologies that adopt early verification and validation, such as model-based design (MBD) process.

1.4 Verification or validation

MBD provides a model-centric design process that represents mathematical expressions and equations with visual models and uses models to design, analyze, verify, validate, and test dynamic systems [131]. MBD helps reduce development time and improve product quality by taking advantage of early verification and validation. Today, MBD has been widely adopted in the design of industrial-scale hybrid systems such as control systems in automobiles and avionics [188, 164]. Models specified in Simulink [186] are able to express complex dynamics, capture discrete state-machine behavior by allowing both boolean and real-valued variables, and allow a layered design approach through modularity and hierarchical compositions. Thus, the high-level system model created during the specification design stage can serve as an executable specification. Performing high-fidelity simulations on this model sheds light on incomplete and inconsistent specifications. MBD helps to reduce the amount of errors found at later design stages. Moreover, it acts as a bridge between implementation and requirements and provides engineers a prototype system to verify their designs earlier in the process compared to the traditional process.

The MBD process can also fit in a V process as shown in Figure 1.2. First, designers capture the dynamic characteristics of the plant, the physical part of the system, using differential, logic, and algebraic equations. This is collectively called the plant model, and examples include the rotational dynamics model of the camshaft in an automobile engine, the thermodynamic model of an internal combustion engine, and

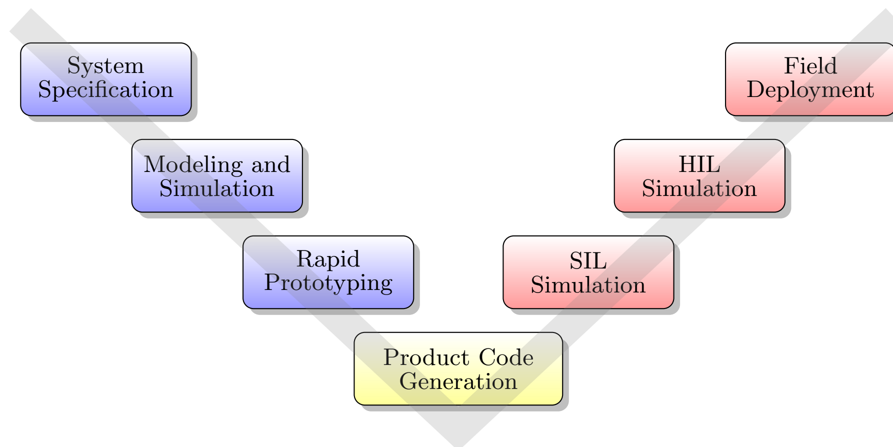

Figure 1.2: The MBD V design process. Rapid prototyping is also known as model-in-the-loop simulation. SIL (software-in-the-loop) incorporates the production software code into the plant model and performs offline simulation while HIL (hardware-in-the-loop) combines the actual physical hardware with the plant model and performs real-time simulation.

atmospheric turbulence models. Control designers then employ control laws, which are implemented as a controller, to regulate the behavior of the physical system. The closed-loop model consists of at least one plant and one controller. Once controller synthesis is done, designers perform extensive offline simulations on the closed-loop model. The objective is to analyze the time response of the dynamic system by observing the time-varying simulation signals of interest. These signals are the results of stimulating the closed-loop model. Simulation can immediately reveal errors and bugs in specification, modeling, and implementation. An important aspect of this step is validation, such as checking whether the time response of the closed-loop system matches a set of requirements (also called specifications). If the simulation behavior is deemed unsatisfactory, then the designer refines or tunes the controller design and repeats the validation step. The whole validation is done on a closed-loop model and is known as rapid prototyping or model-in-the-loop (MIL). After several iterations, the prototype system will be automatically translated into a production code. This process is at the bottom of the V process in Figure 1.2. SIL (software-in-the-loop) and HIL (hardware-in-the-loop) will

then be applied to test the implementation and the system is simulated offline and in real-time, respectively. Finally, the system is deployed and calibrated.

Verification techniques are powerful and expressive, but most problems are undecidable [14] when analyzing hybrid systems. MBD largely depends on simulation. Thus, researchers transplant the logics used for verification to simulation. This transplantation results in an incomplete simulation-based solution that can be used to search for falsification counterexamples with respect to some temporal logic properties. Incompleteness means that even if no falsification instance is found after extensive simulations, there is still no guarantee that the property holds for the system as there may be corner cases unexplored by the simulations. However, if it falsifies the property, the counterexample provides evidence signals leading the system to undesirable behaviors, and is useful for debugging. S-Taliro [20] integrates the space robustness metric [92] defined on MTL [132] to analyze signals of hybrid systems and uses stochastic sampling and optimization techniques [19, 163, 182] to guide the search. Breach [83] supports not only the space robustness metric, but also time robustness metric [22] defined on STL [152].

Simulation-based validation is more practical than verification since it does not need to handle the layered structure and the equations behind hybrid systems. It can easily analyze industrial-sized models. However, without the logical foundation from verification, pure simulation approaches are reduced to ad-hoc testing. The amount of allowed simulation time increases confidence in the system design, but does not maximize the potential benefits from early verification and validation. Logic-enhanced simulation-based validation gives us hope for applying verification techniques to hybrid systems.

1.5 Requirement defects

Ideally, design requirements, also named design specifications, should range from higher system level requirements that describe the desirable system performance in suitable formal logics, to lower implementation level requirements that accurately describe the expected inputs and outputs, as well as the requested function of each component unit. For instance, a good system-level requirement of a transmission controller design would be "the car should not shift from gear 2 to gear 1 and back to gear 2 within 2 seconds."

However, in reality, industry requirements are rarely documented in formal language, but instead as informal and vague statements, often in colloquial languages. Occasionally, they are open statements, such as, "the idle engine speed would be less than 3000 rpm (revolutions per minute)." Also these requirements would be common knowledge that is insufficient to evaluate test results. Moreover, a problematic situation in industry is the existence of large amounts of undocumented legacy code and legacy models. This makes it difficult for successors to take over a project after the departure of the original designers. At times, the specifications are wrong or obsolete and should be eliminated to prevent misleading and misunderstanding.

Requirement defects are considered pitfalls for software engineering and may be hidden in all stages of the development process. Inadequate requirements result in the delayed or aborted projects, cost overrun, or software failures after delivery [124, 148]. Since hybrid systems embrace both software and hardware, requirement defects are a more complex and severe obstacle for hybrid system designers. First, we analyze the reasons behind requirement defects:

- The designing process is a highly dynamic procedure and is characterized by rapid

evolution through the whole design life cycle. This quickly makes documentation of design requirements obsolete but engineers do not want to spend extra time and make the extra effort to track and update documentation.

- Design engineers normally do not have the training in expressing design requirements using formal logic languages. Thus, the resulting documentation may be incomplete, inadequate, or inaccurate. Ambiguous, unclear, or poorly written requirements more often mislead the validation and test than convey the concrete requirement and cause system failures [148].

- The lack of sufficient requirement management systems exacerbates requirement defects. Lack of adherence to standards, such as using informal requirement representations, designing a sufficient requirement management software that is able to check the correctness, completeness, and consistency of requirements, is a long standing obstacle. In return, lacking traceable requirements makes designers more reluctant to update requirements when they are needed.

Requirement defects are costly and time-consuming to fix. Techniques to discover requirements to prevent the defects are highly sought [138]. Based on practical experiences, researchers recommend well-known inspection methods, such as scenarios and checklists [172], or scenario-based formal methods, such as UML [39], Statecharts [106], and design behavior trees (DBT) [86]. Unfortunately, hybrid systems heavily interact with the physical world. Some scenarios or items in the checklist may be very hard or even infeasible to prepare. Others require the designed system to enter certain states and meet some preconditions in order to be checked. Thus, these practices may not achieve the deserved results, and requirement defects for hybrid systems can hardly be resolved by traditional testing and inspection methods.

Another approaches that actively mine requirements from programs and circuits are well-studied [15, 16, 88, 100, 139, 144, 183, 185, 207]. Requirement mining techniques are numerous and vary based on the kind of requirements that are mined, such as automata, temporal rules, and sequence diagrams. They also vary based on the input to the miner. The input to techniques based on static analysis or model checking is the source code, while the input to dynamic techniques is a set of execution traces. Techniques based on mining temporal rules [15, 185] try to automatically learn an automaton representing the temporal specifications, and usually focus on application programming interface (API) usage in libraries. The individual components within these libraries are often terminating programs, and requirement automata focus on legal interaction patterns between components of libraries. In contrast, we need to mine behavioral requirements from a closed-loop hybrid control system which has vastly different semantics from that of a software program. Thus, in the industrial setting, techniques that act as a physician to detect model requirements are highly desired to help defeat requirement defects.

1.6 Contributions

This thesis explores techniques used in the modeling, analysis, verification, and validation of hybrid systems. It uses academic solutions to bridge the critical gap in practical industry settings. The main contributions of this thesis are summarized as follows:

- We provide a survey on the state-of-the-art of hybrid system modeling methods and give thorough cost and benefit analysis of major methods.
- For the representative discrete system, a network of communicating FSMs, we

provide efficient solutions to symbolically encode the system and to verify critical and user desired properties. Specially, we demonstrate an application that uses a symbolic equivalent checker to automatically generate a test suite for fault-based testing [126].

- For a second discrete formalism, ECA rules, we successfully develop fully symbolic algorithms to verify both termination and confluence properties. To the best of our knowledge, we provide the first practical results to analyze the confluence property [129].

- We propose a requirement mining framework [128] to alleviate requirement defects. The framework automatically mines requirements and represents the mined requirements in temporal logic. This work can be efficiently used to help designers effectively validate systems and provide valuable counterexamples for debugging. We demonstrate the practical applicability of our mining framework in two case studies: a simple automatic transmission controller, and an industrial closed-loop model of the airpath-control in an automobile engine model.

- We enhance the requirement mining framework with efficient strategies for synthesizing parameters of monotonic temporal logic formulas. We propose to formulate the query for monotonicity in fragments of first-order logic with quantifiers, real arithmetic, and uninterpreted functions. This query is then answered by a high-performance satisfiability modulo theories (SMT) solver.

- We further improve the mining framework by introducing weighted temporal logics [127] that are able to distinguish the different contributions associated with the components of the logic expression. Furthermore, these logics can to neutralize

the affect of using various measurement units. Most importantly, the new logics help the framework handle discrete events and speed up the mining process.

1.7 Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses the two main approaches for modeling hybrid systems. The first one transforms hybrid systems into discrete systems while the second one retains the original dynamic systems. For each approach, we analyze the costs and benefits, and provide detailed examples. Chapter 3 explores the first approach to verify discrete systems using symbolic techniques. Chapter 4 details the advantages and disadvantages of retaining the original dynamic systems and introduces a state-of-the-art temporal logic designed for analyzing continuous signals of hybrid systems. Chapter 5 proposes a general framework for mining high-level specifications from a closed-loop hybrid system. It also proposes improved temporal logics used to enhance the result. Finally, in Chapter 6, we offer the conclusion and the future work.

Chapter 2

Models for Hybrid Systems

This chapter starts with a brief overview of dynamic systems and the formalisms used to model hybrid systems. Before proceeding, we establish some symbolic conventions. Let $\mathbb{R}$ be the set of real numbers, $\mathbb{Z}$ be the set of integers, and $\mathbb{N}$ be the set of natural numbers. Also, expressions are used to represent subsets of the these sets, e.g., $\mathbb{R}^{\geq 0}$ represents the subset of real numbers greater than or equal to zero. $\mathbb{R}^n$ denotes a vector space with n dimensions. Vectors in $\mathbb{R}^n$ are denoted in bold font and their components are indexed from 1 to n, e.g., $\mathbf{p} = (p_1, \cdots, p_n)$.

2.1 Overview of dynamic systems

Hybrid systems, ubiquitous in engineering applications, are dynamic systems where both continuous and discrete behaviors are present. For better explanation, we start with a review of dynamic systems. Dynamic systems are systems which automatically react to the environment or to external inputs. A dynamic system can be described by the evolution of the system state over time. System states are classified into three types:

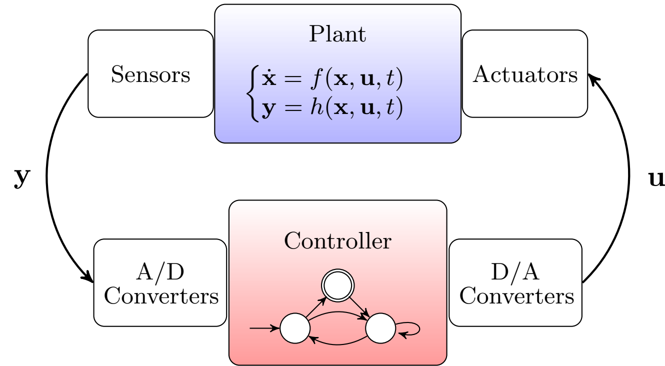

Figure 2.1: A hybrid system diagram. The system interacts with the physical world, represented as a plant which is guided by ordinary differential equations (ODE), and a discrete cyber world, represented as a controller which is guided by a state transition system. Sensors monitor the plant and provide inputs to the controller through an analog-to-digital (A/D) converter. The controller outputs go through a digital-to-analog (D/A) converter and then drive the actuators.

- A continuous state is a state that has values in $\mathbb{R}^n$, where $n \geq 1$. We use $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ to denote a continuous state. For instance, the continuous variables such as speed and acceleration can be used to describe the state of an automobile. A continuous dynamic system is a system that only has continuous states.

- A discrete state is a state that has values in a countable set. For example, if we use the gear position to describe the state of a four-speed automobile, its value could be any of the four values 1, 2, 3, and 4. A discrete dynamic system is a system where all states are discrete.

- A hybrid state is a state where parts of the state are continuous while others are discrete. If we use speed, acceleration, and gear position to describe the state of an automobile, then the state is hybrid. Thus, an automobile is normally considered a hybrid system.

The two main components of a typical hybrid system are a real-time control system and a plant model, as shown in Figure 2.1. The plant models the continuous dynamics of the physical system and its environment while the controller reacts to and regulates the plant. Commonly, the controller is digital, but the plant is analog. Thus, the hybrid system relies on sensors to monitor the plant. The controller pulls the digitized outputs of sensors through an analog-to-digital (A/D) converter and uses them to make logical decisions. Then, the control actions are converted to analog signals through a digital-to-analog (D/A) converter and are applied to actuators to intentionally regulate the plant. All components of a hybrid system form a closed-loop system so that the system is able to constantly react to and regulate the physical world to achieve the desired performance. In order to analyze the behavior a hybrid system, the concept of time is key.

The physical dynamics modeled by the plant evolve over continuous time $\mathbb{T} = \mathbb{R}^{\geq 0}$. This type of system is generally defined as a continuous-time system, and its state space model [27, 45] is guided by the following set of ordinary differential equations (ODE) [149]:

$$\begin{cases} \dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t) \\ \mathbf{y}(t) = h(\mathbf{x}(t), \mathbf{u}(t), t), \end{cases} \tag{2.1}$$

where $t \in \mathbb{T}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^p$, $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{T} \to \mathbb{R}^n$, $h : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{T} \to \mathbb{R}^p$; $\mathbf{x}$ is a continuous state of the system; the vector space $\mathbb{R}^n$ is known as the state space of the system and n is normally referred to as order or dimension of the system; t is real time; $\dot{\mathbf{x}}(t)$ is the first derivative of $\mathbf{x}(t)$; $\mathbf{u}$ and $\mathbf{y}$ represent the input and output functions, respectively, while m and p are their dimensions. We call the continuous-time state evolution *continuous flow*. The function f specifies which direction the state of

the system flows and is known as *controlled vector field*. Note that (2.1) only involves first order differential equations. However, the system dynamics for many applications are described by higher order differential equations. These systems can be transformed to an equivalent state space form by introducing new variables [18].

The classic approach to analyze continuous-time systems is to use sampled-data theory [146]. This theory assumes that the measurements and control actions are polled at a fixed sampling rate. Thus, at discrete instants of time k, the input and state are measured and the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{u}$ are defined, where k ranges over the set of nonnegative integers $\mathbb{N}$. Thus, the time domain changes to $\mathbb{T} = \mathbb{N}$ and the system is defined as a discrete-time system. The state space model of a discrete-time system is then described as a set of difference equations of the form [166]:

$$\begin{cases} \mathbf{x}(k+1) = f(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{y}(k+1) = h(\mathbf{x}(k), \mathbf{u}(k)), \end{cases} \tag{2.2}$$

where k is the time-stamp when sampling, $\mathbf{x}(k)$ an n-dimensional state vector at time k, $\mathbf{u}(k)$ an m-dimensional input vector at time k, and $\mathbf{y}(k)$ a p-dimensional output vector at time k. Functions f and h in (2.2) can be written in matrix format as:

$$\begin{cases} f(\mathbf{x}, \mathbf{u}) = A\mathbf{x} + B\mathbf{u} \\ h(\mathbf{x}, \mathbf{u}) = C\mathbf{x} + D\mathbf{u}, \end{cases} \tag{2.3}$$

where A is an n-by-n matrix, B an n-by-m matrix, C a p-by-n matrix, and D a p-by-m matrix. Sampling not only helps to transform a continuous-time system into a discrete-time system, but also simplifies the system. It provides the ability to analyze challenging problems more efficiently [46]. This efficiency comes with cost, however. For example, discrete-time models might miss some system behavior if under-sampled. Thus, it is critical to choose an appropriate sampling period in order to reduce the

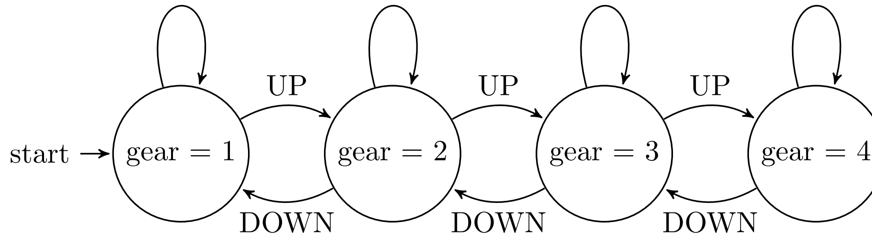

Figure 2.2: The simplified gear shift logic of a four-speed transmission.

risk of fatal errors in some applications. Moreover, both discrete-time and continuous-time systems can be further classified into linear or nonlinear systems depending on whether the differential equations or difference equations are linear or nonlinear. Note that most physical phenomena are nonlinear, which is one of the main reasons analyzing continuous time dynamic systems is computationally expensive. For nonlinear hybrid systems, researchers can only approach them through piecewise-constant polyhedral approximation [112] or Taylor model-based flow-pipe construction [53] and hope that the approximation analysis has enough precision and good convergence for the original system.

The correct computation is merely the first step. Time is also an important concept for the controller of a hybrid system. The time when computations are performed affects the order of operations, which is critical to the interaction between the plant and the controller [184]. Since the controller handles various types of sensors and actuators, naturally, it is an asynchronous or event-driven system due to varying sampling schemes, potential packet loss, and possible transmission delays. Normally, this type of system is formalized as an automaton, an FSM or an NCFSM. Take our four-speed automobile example again. A simplified version of the gear shift logic is shown in Figure 2.2. The transmission position can only be one of the four gear values. The

transition between any adjacent gears depends on a finite set of predefined events. For example, if the transmission is at gear 3 and the speed of the vehicle increases to a threshold predefined according to the current gear position, the gear shift logic emits the gear `UP` signal which causes the physical gear to move up. We call this discrete mode transition, *discrete jump.* Considerations such as how fast the controller responds to the environmental changes, how long signals take to reach the controller from sensors or arrive at actuators from the controller, and how frequently the controller samples the environment are critical to the design of a controller. Although the controller might not be directly affected by physical time, it has to change the physical environment before or at a specific time.

A hybrid system naturally involves both continuous flows and discrete jumps. It normally has a heterogeneous architecture and works in a highly concurrent fashion. How to maintain the safety, stability, robustness, and reliability of the system strongly depends on verification and validation analysis.

2.2 Modeling discrete systems

Recall that one option to model a hybrid system is to discretize a continuous plant model to formalize the whole system into a discrete system. Then, we can analyze it using well-developed theories and tools. However, the complexity of events and conditions as well as the non-determinism of the environment call for more powerful and expressive formalisms such as a network of communicating FSMs (NCFSM) [115], event-condition-action (ECA) rules [156], or Petri nets (PNs) [161]. These formalisms provide better support for modeling and analysis. Here, we focus on the two important and powerful formalisms: NCFSM and ECA rules.

2.2.1 A network of communicating FSMs

The formalism of a network of communicating FSMs (NCFSM) was originally proposed to model concurrent systems [44], such as communication systems and multiprocessor computers, consisting of several components communicating with each other via first-in-first-out (FIFO) queues. Each component is modeled as a communicating FSM (CFSM). In general, the state space of an NCFSM with unbounded queues is infinite but the slow environment assumption [115], e.g., the system only reacts to the environmental events only after consuming all internal events, avoids the need to manage infinite state spaces. Fortunately, this assumption is satisfied by most systems. Here, we tackle an NCFSM in a slow environment and model the communication channel as a single global queue of size one.

2.2.1.1 CFSM

Let $K \geq 2$ be the number of CFSM components in an NCFSM.

Definition 2.1. (communicating finite state machine) A communicating finite state machine (CFSM) M_k $(1 \leq k \leq K)$ is a six-tuple $(\mathcal{S}_k, \mathcal{X}_k, \mathcal{Y}_k, \delta_k, \lambda_k, s_k)$, where:

- $\mathcal{S}_k$ is a finite set of local states, represented as circles.
- $\mathcal{X}_k$ is a finite set of input symbols, which can come from the environment or other CFSMs.
- $\mathcal{Y}_k$ is a finite set of output symbols, which can be absorbed by the environment or become input symbols to other CFSMs.
- $\delta_k : \mathcal{S}_k \times \mathcal{X}_k \rightarrow \mathcal{S}_k$ is the local state transition function.
- $\lambda_k : \mathcal{S}_k \times \mathcal{X}_k \rightarrow \mathcal{Y}_k$ is the output function.

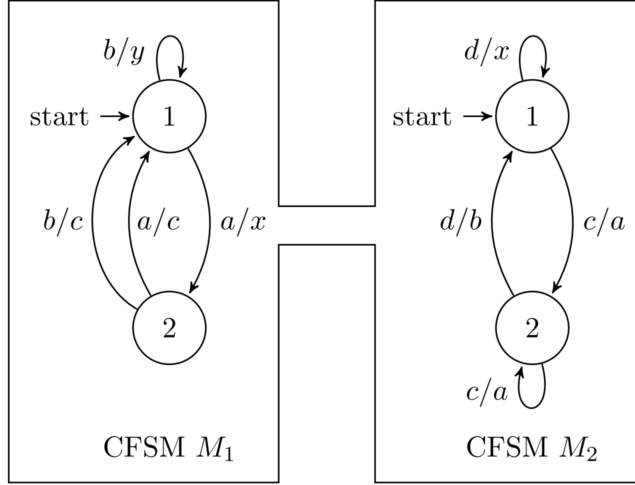

Figure 2.3: An example of NCFSM $M_{ex} = (M_1, M_2)$ where $\mathcal{S}_1 = \mathcal{S}_2 = \{1, 2\}$, $\mathcal{X}_1 = \{a, b\}$, $\mathcal{X}_2 = \{c, d\}$, $\mathcal{Y}_1 = \{c, x, y\}$, $\mathcal{Y}_1 = \{a, b, x\}$, and $s_1 = s_2 = 1$. M_1 and M_2 communicate with each other through a global channel with buffer size 1. Edge b/c means that, in order to take the transition, the precondition of the state is the presence of an input symbol b and the postcondition is the generation of an output symbol c. If CFSMs M_1 and M_2 are at state 1, the presence of symbol a makes M_1 take the transition a/x and move to state 2 with an output symbol x.

- $s_k \in \mathcal{S}_k$ is the initial state, indicated by the start mark. □

Given a CFSM M_k, for all $i, j \in \mathcal{S}_k$, $a \in \mathcal{X}_k$, and $b \in \mathcal{Y}_k$, if $\delta_k(i, a) = j$ and $\lambda_k(i, a) = b$, we say that, when M_k is in local state i and receives input symbol a, it goes to local state j and emits output symbol b. We write this local, or internal, transition as $i\,[_{M_k,\,a/b}\rangle\, j$. Take CFSM M_1 in Figure 2.3 for example. M_1 has two states and the local transition $1\,[_{M_1,\,a/x}\rangle\, 2$ shows the precondition, the presence of symbol a, and the postcondition, the generated symbol x, of the transition from state 1 to state 2.

This definition assumes that each CFSM component M_k is deterministic and completely specified since for any $i \in \mathcal{S}_k$ and $a \in \mathcal{X}_k$ there exists exactly one $j \in \mathcal{S}_k$ and one $b \in \mathcal{Y}_k$ such that $i\,[_{M_k,\,a/b}\rangle\, j$. In practice, even if CFSMs are non-deterministic, we can always transform them into deterministic ones using the powerset construction method [119]. Also, if CFSMs are not completely specified, we can add self-loops to generate an empty symbol ϵ to complete all missing transitions without changing the

behavior of the system. A CFSM is initially connected if every local state can be reached from its initial local state s_k, and is strongly connected if every local state can be reached from any other local state by feeding the CFSM an appropriate input string.

Extending δ_k and λ_k to input sequences by recursively applying λ_k and δ_k results in for all $i \in \mathcal{S}_k$, $a \in \mathcal{X}_k$, and $\alpha \in \mathcal{X}_k^*$,

$$\begin{aligned} \delta_k(i,\epsilon) &= i & \delta_k(i,a\alpha) &= \delta_k(\delta_k(i,a),\alpha) \\ \lambda_k(i,\epsilon) &= \epsilon & \lambda_k(i,a\alpha) &= \lambda_k(i,a)\lambda_k(\delta_k(i,a),\alpha), \end{aligned}$$

where ϵ is the empty symbol. We define that input sequence $\alpha \in \mathcal{X}_k^*$ distinguishes local states $i, j \in \mathcal{S}_k$ if $\lambda_k(i,\alpha) \neq \lambda_k(j,\alpha)$, and two local states are distinguishable if there is a sequence that distinguishes them, otherwise they are equivalent. A CFSM is minimal if it does not contain any two equivalent local states.

The main difference between a CFSM and an FSM is the ability to communicate. This means that a transition in one CFSM might cause transitions in other CFSMs. It is straightforward to change a CFSM into an FSM with a FIFO channel that links all FSMs and the environment. This channel is used to transport symbols within the system as well as between the system and the environment. In the networks and distributed systems literature, it is quite common to represent symbols as messages. Here, we use buffer β to represent this FIFO channel.

In most real systems, the communication between CFSMs cannot be interrupted by the environment. If the output symbol a of a CFSM can be absorbed by another CFSM as an input symbol, then the system does not accept any other input symbol from the environment until a has been consumed. This property is named *slow environment* in the literature [115]. Another property states that the input alphabets of any two CFSMs are disjoint. This property is important to prevent fatal errors due to race conditions. We assume that both properties hold for our systems.

2.2.1.2 NCFSM

Definition 2.2. (network of communicating finite state machine) A network of CFSMs (NCFSM) is a $(K+1)$-tuple $M = (\beta, M_1, M_2, ..., M_K)$, where each M_k, for $1 \leq k \leq K$, is a CFSM and β is a FIFO buffer with size 1 which can contain symbols in transit among CFSMs. The semantics of an NCFSM is that of its product machine M_{prod}, another six-tuple $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, \delta, \lambda, \mathbf{s}_{init})$, where:

- $\mathcal{Y} = \bigcup_{1 \leq k \leq K} \mathcal{Y}_k$ is a finite set of output symbols.
- $\mathcal{X} = \bigcup_{1 \leq k \leq K} \mathcal{X}_k$ is a finite set of input symbols and $\mathcal{X}_k \cap \mathcal{X}_h = \emptyset$ for $1 \leq k < h \leq K$.
- $\mathcal{S}$ is a finite set of global states. A global state is a $(K+1)$-tuple $(i_\beta, i_1, ..., i_K)$ where $i_\beta \in \{\epsilon\} \cup \mathcal{Y} \cup \mathcal{X}$ and $i_k \in \mathcal{S}_k$, for $1 \leq k \leq K$. Thus, $\mathcal{S}$ is the cross-product $(\{\epsilon\} \cup \mathcal{Y} \cup \mathcal{X}) \times \mathcal{S}_1 \times ... \times \mathcal{S}_K$.
- $\delta : \mathcal{S} \times (\{\epsilon\} \cup \mathcal{X}) \rightarrow \mathcal{S}$ is the global state transition function.
- $\lambda : \mathcal{S} \times (\{\epsilon\} \cup \mathcal{X}) \rightarrow \mathcal{Y}$ is the global output function.
- $\mathbf{s}_{init} = (\epsilon, s_1, ..., s_K) \in \mathcal{S}$ is the initial global state. $\square$

Let $\mathcal{Z}_{int} = \mathcal{Y} \cap \mathcal{X}$ be the set of internal symbols that can be placed in buffer β; we underline these symbols to stress they are internal, e.g. in $\underline{a}$. Let $\mathcal{Y}_{ext} = \mathcal{Y} \setminus \mathcal{X}$ be the set of external output symbols of the NCFSM. $\mathcal{Y}_{ext}$ includes all output symbols that can be absorbed by the environment and is disjoint from $\mathcal{Z}_{int}$. Let $\mathcal{X}_{ext}$ be the set of external input symbols that only the environment can place into buffer β, but can also include any symbol in $\mathcal{Z}_{int}$, thus, $\mathcal{X} \setminus \mathcal{Y} \subseteq \mathcal{X}_{ext} \subseteq \mathcal{X}$.

Given $\mathbf{i} = (i_1, ..., i_K)$, define $\mathbf{i}|_{k:j_k}$ to be the vector $(i_1, ..., j_k, ..., i_K)$, obtained from $\mathbf{i}$ by setting the kth component to j_k. A global state $(i_\beta, i_1, ..., i_K)$ is considered

stable if $i_\beta = \epsilon$, and we represent it as $\mathbf{i}$. Otherwise, it is unstable and we represent it as $a.\mathbf{i}$, where $a \in \mathcal{Z}_{int}$. Let $\mathcal{S}_{st}$ and $\mathcal{S}_{unst}$ be the set of reachable stable and unstable states of the NCFSM, respectively, and let $\mathcal{S}_{rch} = \mathcal{S}_{st} \cup \mathcal{S}_{unst}$ be the set of reachable states.

If there is a local transition $i_k \,[M_k, a/x\rangle\, j_k$, λ and δ are defined as:

- $\delta(\mathbf{i}, a) = \mathbf{i}|_{k:j_k}$ and $\lambda(\mathbf{i}, a) = x$, with $a \in \mathcal{X}_{ext}$, and $x \in \mathcal{Y}_{ext}$. The system transitions between stable global states, so we write $\mathbf{i}\,[M, a/x\rangle\,\mathbf{i}|_{k:j_k}$. M is omitted when it is clear from the context.

- $\delta(\mathbf{i}, a) = x.\mathbf{i}|_{k:j_k}$ and $\lambda(\mathbf{i}, a) = \epsilon$, with $a \in \mathcal{X}_{ext}$ and $x \in \mathcal{Z}_{int}$. The system transitions from a stable global state to an unstable global state, so we write $\mathbf{i}\,[a/\underline{x}\rangle\, x.\mathbf{i}|_{k:j_k}$.

- $\delta(a.\mathbf{i}, \epsilon) = \mathbf{i}|_{k:j_k}$ and $\lambda(a.\mathbf{i}, \epsilon) = x$, with $a \in \mathcal{Z}_{int}$ and $x \in \mathcal{Y}_{ext}$. The system transitions from an unstable global state to a stable global state, so we write $a.\mathbf{i}\,[\underline{a}/x\rangle\,\mathbf{i}|_{k:j_k}$.

- $\delta(a.\mathbf{i}, \epsilon) = x.\mathbf{i}|_{k:j_k}$ and $\lambda(a.\mathbf{i}, \epsilon) = \epsilon$, with $a \in \mathcal{Z}_{int}$ and $x \in \mathcal{Z}_{int}$. The system transitions between unstable global states, so we write $a.\mathbf{i}\,[\underline{a}/\underline{x}\rangle\, x.\mathbf{i}|_{k:j_k}$.

Due to the slow environment assumption [115], a buffer of size one is sufficient, as β can only contain zero or one symbol from $\mathcal{Z}_{int}$. Moreover, when in unstable global states, the internal symbol held in β is not externally observable.

Also, we can extend δ and λ on global stable states to accept finite external input symbol sequences as follows:

$$\begin{aligned} \delta(\mathbf{i}, \epsilon) &= \mathbf{i} & \lambda(\mathbf{i}, \epsilon) &= \epsilon \\ \delta(\mathbf{i}, a\alpha) &= \delta(\delta(\mathbf{i}, a), \alpha) & \lambda(\mathbf{i}, a\alpha) &= \lambda(\mathbf{i}, a)\lambda(\delta(\mathbf{i}, a), \alpha) \\ \delta(a.\mathbf{i}, \alpha) &= \delta(\mathbf{i}, a\alpha) & \lambda(a.\mathbf{i}, \alpha) &= \lambda(\mathbf{i}, a\alpha). \end{aligned}$$

Definition 2.3. (reachable stable global states) The set of reachable stable global

states is defined as:

$$\mathcal{S}_{st} = \{\mathbf{j} : \exists\alpha \in \mathcal{X}^*_{ext}, \delta(\mathbf{s}_{init}, \alpha) = \mathbf{j}\} \subseteq \mathcal{S},$$

while the set of reachable unstable global states contains all unstable global states reachable from any reachable stable global state:

$$\mathcal{S}_{unst} = \{a^{(n)}.\mathbf{i}^{(n)} : \exists\mathbf{i}^{(0)} \in \mathcal{S}_{st}, \exists a^{(0)} \in \mathcal{X}_{ext}, \exists a^{(1)} \cdots a^{(n)} \in \mathcal{Z}^+_{int},$$

$$\mathbf{i}^{(0)} \,{\scriptstyle [a^{(0)}/\underline{a^{(1)}}\rangle}\, a^{(1)}.\mathbf{i}^{(1)} \,{\scriptstyle [\underline{a^{(1)}}/\underline{a^{(2)}}\rangle}\, \cdots \,{\scriptstyle [\underline{a^{(n-1)}}/\underline{a^{(n)}}\rangle}\, a^{(n)}.\mathbf{i}^{(n)}\} \subseteq \mathcal{S}. \quad \square$$

Thus, the set of reachable global states is $\mathcal{S}_{rch} = \mathcal{S}_{st} \cup \mathcal{S}_{unst}$. Given an NCFSM, due to the slow environment assumption, neither the unstable global states traversed nor the symbols in β are observable; only the symbols in $\mathcal{Y}_{ext}$ are. An NCFSM behaves as a black box. Thus, we care more about the transitions between stable global states, instead of invisible local transitions. These global state transitions have the following definitions:

Definition 2.4. (stable global transition and stable output function) NCFSM in stable global state $\mathbf{i}$, if presented with external symbol a, proceeds through the sequence of transitions:

$$\mathbf{i} \,{\scriptstyle [a/\underline{a^{(1)}}\rangle}\, a^{(1)}.\mathbf{i}^{(1)} \,{\scriptstyle [\underline{a^{(1)}}/\underline{a^{(2)}}\rangle}\, \cdots \,{\scriptstyle [\underline{a^{(n-1)}}/\underline{a^{(n)}}\rangle}\, a^{(n)}.\mathbf{i}^{(n)} \,{\scriptstyle [\underline{a^{(n)}}/b\rangle}\, \mathbf{j},$$

where $n \geq 0$, $\mathbf{j}$ is a stable global state, and b is an external output symbol which will be absorbed by the environment. The effect of this sequence can be merged and represented as $\mathbf{i} \,{\scriptstyle [\![M, a/b\rangle\!\rangle}\, \mathbf{j}$ where M can be omitted when it is clear from the context. Correspondingly, the stable global transition is defined as $\delta_{obs}(\mathbf{i}, a) = \mathbf{j}$. Similarly, the stable global output function is defined as $\lambda_{obs}(\mathbf{i}, a) = b$. $\square$

Definition 2.5. (observable product machine) The observable product machine M_{obs} of an NCFSM is a six-tuple $(\mathcal{S}_{st}, \mathcal{X}_{ext}, \mathcal{Y}_{ext}, \delta_{obs}, \lambda_{obs}, \mathbf{s}_{init})$, where:

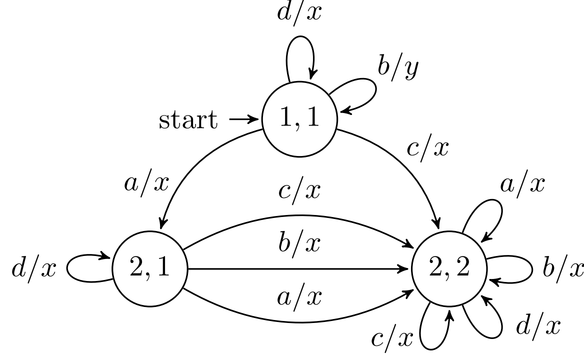

Figure 2.4: The observable product machine M_{obs} for NCFSM M_{ex} in Figure 2.3 where $\mathcal{Z}_{int} = \{a, b, c\}$, $\mathcal{X}_{ext} = \{a, b, c, d\}$, $\mathcal{Y}_{ext} = \{x, y\}$, and $\mathcal{S}_{st} = \{(\epsilon, 1, 1), (\epsilon, 2, 1), (\epsilon, 2, 2)\} \subset \{\epsilon\} \times \mathcal{S}_1 \times \mathcal{S}_2$, and $\mathbf{s}_{init} = (\epsilon, 1, 1)$.

- $\mathcal{S}_{st}$ is the set of reachable stable global states of M_{prod}.
- $\mathcal{X}_{ext}$ is the set of external input symbols of M_{prod}.
- $\mathcal{Y}_{ext}$ is the set of external output symbols of M_{prod}.
- $\delta_{obs} : \mathcal{S}_{st} \times \mathcal{X}_{ext} \rightarrow \mathcal{S}_{st}$ is the stable global state transition function of M_{prod}.
- $\lambda_{obs} : \mathcal{S}_{st} \times \mathcal{X}_{ext} \rightarrow \mathcal{Y}_{ext}$ is the stable global output function of M_{prod}.
- $\mathbf{s}_{init}$ is the initial global state of M_{prod}. $\square$

Consider NCFSM M_{ex} in Figure 2.3, with $\mathcal{Z}_{int} = \{a, b, c\}$, $\mathcal{X}_{ext} = \{a, b, c, d\}$, $\mathcal{Y}_{ext} = \{x, y\}$, $\mathbf{s}_{init} = (\epsilon, 1, 1)$, and $\mathcal{S}_{st} = \{(\epsilon, 1, 1), (\epsilon, 2, 1), (\epsilon, 2, 2)\} \subset \{\epsilon\} \times \mathcal{S}_1 \times \mathcal{S}_2$. The global state with $i_1 = 1$ (M_1) and $i_2 = 2$ (M_2) can only be unstable, as all incoming transitions (not counting self-loops) to $i_2 = 2$ and $i_1 = 1$ only emit symbols in $\mathcal{Z}_{int}$. The corresponding M_{obs} is shown in Figure 2.4. M_{obs} is required to verify the observational equivalence of NCFSMs [150] between the specification and the implementation when testing.

Let $a_1/b_1, \cdots, a_n/b_n \in (\mathcal{X}_{ext} \times \mathcal{Y}_{ext})^*$ be a sequence from a state $\mathbf{i} \in \mathcal{S}_{st}$ if

$\lambda_{obs}(\mathbf{i}, a_1 \cdots a_n) = b_1 \cdots b_n$. Let $Tr(\mathbf{i})$ be the set of all sequences from $\mathbf{i} \in \mathcal{S}_{st}$ and let $Tr(M) = Tr(\mathbf{s}_{init})$. Then, we have $Tr(M_{obs}) = Tr(M)$.

2.2.2 ECA rules

Another powerful formalism that can be used to model and analyze discrete systems is event-condition-action (ECA) rules [156]. ECA rules are known for their expressiveness to describe complex events and reactions. Thus, this event-driven formalism is widely used to specify complex systems [2, 23], e.g., for industrial-scale management, and to improve efficiency when coupled with technologies such as embedded systems and sensor networks. Active database management systems (DBMS) also enhance security and semantic integrity of traditional DBMSs using ECA rules; these are now found in most enterprise DBMSs and academic prototypes thanks to the SQL3 standard [133]. ECA rules are used to specify a hybrid system's response to events [23], and are written in the format: *On the occurrence of a set of events, if certain conditions hold, perform these actions.* This self-triggering manner for monitoring a physical system uses resources efficiently, while providing stability to the system due to a flexible knowledge-based control [21].

2.2.2.1 ECA rule syntax

ECA rules have the syntax: **on** *events* **if** *condition* **do** *actions*. If the events have been activated and the boolean condition is satisfied, the rule is triggered and its actions will be performed. In DBMSs, events are normally produced by explicit database operations such as insert and delete [2] while, in reactive systems, they are produced by sensors monitoring environment variables [23], e.g., temperature. Many current ECA languages can model the environment and distinguish between **environmental** and

$$
\begin{aligned}
\mathit{env_vars} &:= \textbf{environmental}\ \mathit{env_var} && \text{READ-ONLY BOUNDED NATURAL NUMBER}\\
\mathit{loc_vars} &:= \textbf{local}\ \mathit{loc_var} && \text{READ-AND-WRITE BOUNDED NATURAL NUMBER}\\
\mathit{factor} &:= \mathit{loc_var} \mid \mathit{env_var} \mid (\ \mathit{exp}\ ) \mid \mathit{number} && \text{“}\mathit{number}\text{” IS A CONSTANT} \in \mathbb{N}\\
\mathit{term} &:= \mathit{factor} \mid \mathit{term} * \mathit{term} \mid \mathit{term}\ /\ \mathit{term} && \text{“/” IS INTEGER DIVISION}\\
\mathit{exp} &:= \mathit{exp} - \mathit{exp} \mid \mathit{exp} + \mathit{exp} \mid \mathit{term}\\
\mathit{rel_op} &:= \geq \mid \leq \mid =\\
\mathit{assignment} &:= \mathit{env_var}\ \textbf{into}\ \mathit{loc_var}\ [, \mathit{assignment}]\\
\mathit{ext_ev_decl} &:= \textbf{external}\ \mathit{ext_ev}\ [\ \textbf{activated when}\ \mathit{env_var}\ \mathit{rel_op}\ \mathit{number}\ ]\ [\ \textbf{read}\ (\mathit{assignment})\ ]\\
\mathit{int_ev_decl} &:= \textbf{internal}\ \mathit{int_ev}\\
\mathit{ext_evs} &:= \mathit{ext_ev} \mid (\mathit{ext_evs}\ \textbf{or}\ \mathit{ext_evs}) \mid (\mathit{ext_evs}\ \textbf{and}\ \mathit{ext_evs})\\
\mathit{int_evs} &:= \mathit{int_ev} \mid (\mathit{int_evs}\ \textbf{or}\ \mathit{int_evs}) \mid (\mathit{int_evs}\ \textbf{and}\ \mathit{int_evs})\\
\mathit{condition} &:= (\mathit{condition}\ \textbf{or}\ \mathit{condition}) \mid (\mathit{condition}\ \textbf{and}\ \mathit{condition}) \mid \textbf{not}\ \mathit{condition} \mid \mathit{exp}\ \mathit{rel_op}\ \mathit{exp}\\
\mathit{action} &:= \textbf{increase}\ (\mathit{loc_var},\ \mathit{exp}) \mid \textbf{decrease}\ (\mathit{loc_var},\ \mathit{exp}) \mid\\
&\quad\ \ \textbf{set}\ (\mathit{loc_var},\ \mathit{exp}) \mid \textbf{activate}\ (\mathit{int_ev})\\
\mathit{actions} &:= \mathit{action} \mid (\mathit{actions}\ \textbf{seq}\ \mathit{actions}) \mid (\mathit{actions}\ \textbf{par}\ \mathit{actions})\\
\mathit{ext_rule} &:= \textbf{on}\ \mathit{ext_evs}\ [\textbf{if}\ \mathit{condition}]\ \textbf{do}\ \mathit{actions}\\
\mathit{int_rule} &:= \textbf{on}\ \mathit{int_evs}\ [\textbf{if}\ \mathit{condition}]\ \textbf{do}\ \mathit{actions}\ [\textbf{with priority}\ \mathit{number}]\\
\mathit{system} &:= [\mathit{env_vars}]^{+}[\mathit{loc_vars}]^{*}[\mathit{ext_ev_decl}]^{+}[\mathit{int_ev_decl}]^{*}[\mathit{ext_rule}]^{+}[\mathit{int_rule}]^{*}
\end{aligned}
$$

Figure 2.5: The designed syntax of ECA rules. “/” is integer division. “*number*” is a constant $\in \mathbb{N}$. The superscript “+” indicates the match-one-or-more quantifier and “*” the match-zero-or-more quantifier.

local variables [4, 29, 56, 145, 162, 178]. Thus, we designed a language to address these issues, able to handle more general cases and allow different semantics for environmental and local variables shown in Figure 2.5.

Environmental variables are used to represent environment states that can only be measured by sensors but not directly modified by the system. For instance, if

we want to increase the temperature in a room, the system may choose to turn on a heater, eventually achieving the desired effect, but it cannot directly change the value of the temperature variable. Thus, environmental variables capture the nondeterminism introduced by the environment, beyond the control of the system. On the other hand, local variables can be both read and written by the system. They may be associated with an actuator, a record value, or an intermediate value describing part of the system state; we provide operations to set (absolute change) their value to an expression, or increase or decrease (relative change) it by an expression; these expressions may depend on environmental variables.

Events can be combinations of atomic events activated by environmental or internal changes. We use keywords **external** and **internal** to classify them. An external event can be **activated when** the value of an environmental variable crosses a threshold; at that time, it may take a snapshot of some environmental variables and **read** them into local variables to record their current values. Only the action of an ECA rule can **activate** internal events. Internal events are useful to express internal changes or required actions within the system. These two types of events cannot be mixed within a single ECA rule. Thus, rules are external or internal, respectively. Then, we say that a state is stable if only external events can occur in it, unstable if actions of external or internal rules are being performed (including the activation of internal events, which may then trigger internal rules). The system is initially stable and, after some external events trigger one or more external rules, it transitions to unstable states where internal events may be activated, triggering further internal rules. When all actions complete, the system is again in a stable state, waiting for environmental changes that will eventually trigger external events.

The condition portion of an ECA rule is a boolean expression on the value of

environmental and local variables; it can be omitted if it is the constant true.

The last portion of a rule specifies which actions must be performed, and in which order. Most actions are operations on local variables which do not directly affect environmental variables, but may cause some changes that will probably be reflected in their future values. Thus, all environmental variables are read-only from the perspective of an action. Actions can also **activate** internal events. Moreover, to handle complex action operations, the execution semantics can be specified as any partial order described by a series-parallel graph; this is obtained through an appropriate nesting of **seq** operators, to force a sequential execution, and **par** operators, to allow an arbitrary concurrency. The keyword **with priority** enforces a priority for internal rules. If no priority is specified, the default priority of an internal rule is 1, the lowest priority, the same as that of external rules.

2.2.2.2 ECA rule semantics

In this section, we focus on the choices of execution semantics for the designed ECA rule syntax, in order to support the modeling of reactive systems. The first choice is how to couple the checking of events and conditions for our ECA rules. There are at least two options: immediate and deferred. The event-condition checking is immediate if the corresponding condition is immediately evaluated when the events occur; it is deferred if the condition is evaluated at the end of a cycle with a predefined frequency. One critical requirement for the design of reactive systems is that the system should respond to external events from the environment [140] as soon as possible. Thus, we choose immediate event-condition checking: when events occur, the corresponding condition is immediately evaluated to determine whether to trigger the rule.

Although we choose to use immediate event-condition checking, deferred check-

ing can still be modeled using immediate checking, for example by adding an extra variable for the system clock and changing priorities related to rule evaluation to synchronize rule evaluations. However, the drawback of deferred checking is that the design has to be tolerant of false ECA rule triggering or non-triggering scenarios. Since there is a time gap between event activation and condition evaluation, the environmental conditions that trigger an event might change during this period of time, causing rules supposed to be triggered at the time of event activation to fail because the "current " condition evaluation is now inconsistent.

Another important choice is how to handle and model the concurrent and nondeterministic nature of reactive systems. We introduce the concept of batch for external events, similar to the concept of transaction in DBMSs [176]. Formally, the boundary of a batch of external events is defined as the end of the execution of all triggered rules. Then, the system starts to receive external events and immediately evaluates the corresponding conditions. The occurrence of an external event closes a batch if it triggers one or more ECA rules; otherwise, the event is added to the current batch. Once the batch closes and the rules to be triggered have been determined, the events in the current batch are cleaned-up to prevent multiple (and erroneous) triggerings of rules.

For example, consider ECA rules r_a: "**on** a **do** $\cdots$" and r_{ac}: "**on** (a **and** c) **do** $\cdots$", and assume that the system finishes processing the last batch of events and is ready to receive external events for the next batch. If external events occur in the sequence "$c, a, \ldots$" the first occurrence of c alone cannot trigger any rule so it begins, but does not complete, the current batch. Then, the occurrence of a triggers both rule r_a and r_{ac}, and thus, closes the current batch. Both rules are triggered and will be executed concurrently. This example shows how, when the system is in a stable state,

the occurrence of a single external event may trigger one or more ECA rules, since there is no "contention" within a batch on "using" an external event: rule r_a and r_{ac} share event a and both rules are triggered and executed. If instead the sequence of events is "$a, c, \ldots$", event a by itself constitutes a batch, as it triggers rule r_a. This event is then discarded by the clean-up so, after executing r_a and any internal rule (recursively) triggered by it, the system returns to a stable state and the subsequent events "$c, \ldots$" begin the next batch.

Also, all external events in one batch are processed concurrently, and the system finishes processing all events in one batch prior to considering the next one. Thus, unless there is a termination error, the system will process all triggered rules, including those triggered by the activation of internal events during the current batch, before considering new external events. This batch definition provides maximum nondeterminism on event order, which is useful to discover design errors in a set of ECA rules.

Under this semantics, during rule execution the system does not respond to any external events and temporarily becomes unresponsive. Therefore, rule execution should be as close to instantaneous as possible. However, from a verification perspective, environmental changes and external event occurrences are nondeterministic and asynchronous. Thus, our semantic allows the verification process to explore all possible combinations without missing errors due to the order in which events occur and environmental variables change.

2.2.2.3 ECA rules for a smart home

Next, the expressiveness of the designed ECA rules will be shown through an automatic light control subsystem in a smart home for senior housing. Figure 2.6 lists the

environmental	*Mtn, ExtLgt, Slp*
local	*lMtn, lExtLgt, lSlp, lgtsTmr, intLgts*
external	*SecElp* **read** (*Mtn* **into** *lMtn*, *ExtLgt* **into** *lExtLgt*, *Slp* **into** *lSlp*) *MtnOn* **activated when** $Mtn = 1$ *MtnOff* **activated when** $Mtn = 0$ *ExtLgtLow* **activated when** $ExtLgt \leq 5$
internal	*LgtsOff, LgtsOn, ChkExtLgt, ChkMtn, ChkSlp*
(R_1) When the room is unoccupied for 6 minutes, turn off lights if they are on.	
r_1	**on** *MtnOff* **if** ($intLgts > 0$ **and** $lgtsTmr = 0$) **do set** (*lgtsTmr*, 1)
r_2	**on** *SecElp* **if** ($lgtsTmr \geq 1$ **and** $lMtn = 0$) **do increase** (*lgtsTmr*, 1)
r_3	**on** *SecElp* **if** ($lgtsTmr = 360$ **and** $lMtn = 0$) **do** (**set** (*lgtsTmr*, 0) **par activate** (*LgtsOff*))
r_4	**on** *LgtsOff* **do** (**set** (*intLgts*, 0) **par activate** (*ChkExtLgt*))
(R_2) When lights are off, if external light intensity is below 5, turn on lights.	
r_5	**on** *ChkExtLgt* **if** ($intLgts = 0$ **and** $lExtLgt \leq 5$) **do activate** (*LgtsOn*)
(R_3) When lights are on, if the room is empty or a person is asleep, turn off lights.	
r_6	**on** *LgtsOn* **do** (**set** (*intLgts*, 6) **seq activate** (*ChkMtn*))
r_7	**on** *ChkMtn* **if** ($lSlp = 1$ **or** ($lMtn = 0$ **and** $intLgts \geq 1$)) **do activate** (*LgtsOff*)
(R_4) If the external light intensity drops below 5, check if the person is asleep and set the lights intensity to 6. If the person is asleep, turn off the lights.	
r_8	**on** *ExtLgtLow* **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*))
r_9	**on** *ChkSlp* **if** ($lSlp = 1$) **do set** (*intLgts*, 0)
(R_5) If the room is occupied, set the lights intensity to 4.	
r_{10}	**on** *MtnOn* **do** (**set** (*intLgts*, 4) **par set** (*lgtsTmr*, 0))

Figure 2.6: ECA rules for the automatic light control subsystem of a smart home for senior housing.

requirements (R_1 to R_5) of such a system. Using motion and pressure sensors, the system attempts to reduce energy consumption by turning off the lights in unoccupied rooms or if the occupant is asleep. Passive sensors emit signals when an environmental variable value crosses a significant threshold. The motion sensor measure is expressed by the boolean environmental variable *Mtn*. The system also provides automatic adjustment

for indoor light intensity based on an outdoor light sensor, whose measure is expressed by the environmental variable $ExtLgt \in \{0, ..., 10\}$. A pressure sensor detects whether the person is asleep and is expressed by the boolean environmental variable *Slp*.

MtnOn, *MtnOff*, and *ExtLgtLow* are external events activated by the environmental variables discussed above. *MtnOn* and *MtnOff* occur when *Mtn* changes from 0 to 1 or from 1 to 0, respectively. *ExtLgtLow* occurs when *ExtLgt* drops below 6. External event *SecElp* models the system clock, occurs every second, and takes a snapshot of the environmental variables into local variables *lMtn*, *lExtLgt*, and *lSlp*, respectively. Additional local variables *lgtsTmr* and *intLgts* are used. Variable *lgtsTmr* is a timer for R_1, to convert the continuous condition "the room is unoccupied for 6 minutes" into 360 discretized *SecElps* events. Rule r_1 initializes *lgtsTmr* to 1 whenever the motion sensor detects no motion and the lights are on. The timer then increases as time elapses, provided that no motion is detected (rule r_2). If the timer reaches 360, internal event *LgtsOff* is activated to turn off the lights and to reset *lgtsTmr* to 0 (rule r_3). Variable *intLgts* acts as an actuator control to adjust the internal light intensity.

Our ECA rules contain internal events to express internal system actions or checks not observable from the outside. *LgtsOff*, activated by rule r_3 or r_7, is used to turn the lights off and activate another check on an outdoor light intensity through the internal event *ChkExtLgt* (rule r_4). *ChkExtLgt* activates *LgtsOn* if $lExtLgt \leq 5$ (rule r_5). *ChkSlp* is activated by rule r_8 to check whether a person is asleep. If true, the event triggers an action that turns the lights off (rule r_9). Internal event *ChkMtn*, activated by rule r_6, activates *LgtsOff* if the room is unoccupied and all lights are on, or if the room is occupied but the occupant is asleep (rule r_7).

When translating a set of requirements expressed in plain English into ECA rules, two similar ECA rules may have very subtle differences. For example, consider R_4

in Figure 2.6 that is translated to rules r_8 and r_9 using the external event *ExtLgtLow*. Another translation could have directly used the external event *SecElp* as rule r'_8: **on** *SecElp* **if** (*lExtLgt* ≤ 5) **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*)). The difference is that r_8 triggers the change of light setting as soon as the exterior light intensity falls below 6 monitored by external event *ExtLgtLow*, while r'_8 would do this at the first occurrence of *SecElp* after the exterior light falls below 6. The hybrid system using rule r'_8 is more tolerant of the potential small gap between these two events' occurrence but it avoids the need to use an external event which might result in reducing state space dramatically, recalling the state space explosion problem.

2.3 Modeling hybrid systems

Although all aforementioned discrete modeling methods have strong and well-developed computing theory to support analysis, discretizing inherited continuous variables and dynamics is still a detour to model hybrid systems. Researchers conclude that a good modeling language for hybrid systems must have the following features [196]:

- Have consistent, rigorous, and expressive syntax and semantics for continuous-time, discrete-time, and logic-based components.
- Support hierarchical composition and modular model construction.
- Have the ability to handle uncertainty in the environment and the system state evolution.

Many languages have been proposed [8, 125, 160, 169, 186, 195, 196, 197, 198]. Some of them provide rigorous semantics [8, 197] and allow logic checking and validation, but lack accommodation for hierarchical composition model construction. Others [160, 186, 195]

provide vivid visual representation and powerful simulation engines, but lack rigorous semantics and the power to express uncertainty. HA and Simulink models are two representatives. We will focus on these modeling methods in the following section.

2.3.1 Hybrid automata

HA are a well-studied formal language to model hybrid systems [6, 8, 12]. The language itself is quite expressive. However, it is not compositional and abstraction-friendly. Syntactically, a hybrid automaton extends normal FSMs with a set of continuous variables and uses edges to represent discrete jumps that are instantaneous actions, vertices to describe continuous flows, and time elapsing activity through differential equations [107].

Definition 2.6. (hybrid automaton) A hybrid automaton is defined by a eight-tuple as $H = (Loc, Var, Labels, \Sigma_{Init}, Inv, Flow, Edges, Jump)$, where:

- Loc is a set of vertices, which are called locations or control modes.
- Var is a set of continuous variables. Each variable $x \in Var$ takes values in $\mathbb{R}$. $\dot{Var}$ represents the first derivatives of these variables. We use $V \subseteq \mathbb{R}^n$ to represent the set of all possible valuations of Var where n is the number of variables and also the dimension of the hybrid system. V is also called the data region and is a finite union of convex polyhedra in $\mathbb{R}^n$.
- $Labels$ is a finite set of synchronization labels and is used to define the parallel composition of two or more HA.
- $\Sigma_{Init} \subseteq \Sigma$ is a set of initial states, where $\Sigma = Loc \times V$ denotes the potential hybrid state space of H, normally infinite state space.

- *Inv* : *Loc* $\rightarrow V$ is a set of invariants for each location, also called guard conditions. The system must leave location l if the invariant *Inv*(l) does not hold.

- *Flow* is a set of activity functions which assigns a set of activities to each location $l \in$ *Loc*. Each activity defines the rates at which the values of continuous variables change in control location l. Normally, activities are in the form of differential equations.

- *Edges* is a set of transitions. Each transition $e = (l, a, l')$ where l is the "from" location, a a synchronization symbol in *Labels*, and l' the "to" location.

- *Jump* is a labeling function that assigns to each transition $e \in$ *Edges*, a boolean condition and all variable changes during the transition. Only when the condition is true, the transition is enabled and subsequently executed, and then all values of variables are changed nondeterministically. For example, *Jump*$(e) = (\{x\}, x \geq 25, 30 \leq x' \leq 40)$ states that transition e can happen only if the value of x is at least 25 and the value of x can be nondeterministically chosen between 30 and 40, with all other variables, if any, remaining unchanged. The former part of the labeling function, $x \geq 25$, is a trigger condition while the latter, $30 \leq x' \leq 40$, assigns values to all continuous variables.

Solving verification problems for HA is extremely hard, and most are undecidable even under severe restrictions [8, 14]. Researchers have limited themselves to rectangular hybrid automata (RHA) [108, 110], or linear hybrid automata (LHA) [8, 12]. RHA define initial, invariant, and jump labeling functions and actions in rectangular regions, a cartesian product of intervals on $\mathbb{R}$ all of whose endpoints are rational [108]. Thus, an activity of a vertex flow $5 \leq \dot{x} \leq 10$ defines that x rises at any rate between 5 and 10. This is also called rectangular activity. LHA restrict all these functions further

to boolean combinations of linear combinations of real variables. Also, flow functions only contain free variables in $Var \cup \dot{Var}$ and all flows are defined in the form of $\dot{x} = \mathbb{Z}$. For instance, the invariant condition of a vertex could be $0 \leq x \leq 100$, which indicates that the state stays in this vertex as long as the value of x is in the range from 0 to 100, and the activity of the flow could be $\dot{x} = 5$, which defines the increase rate of x is 5.

A special case of LHA is timed automata [10]. A timed automaton has an implicit time variable x_l for all control locations $l \in Loc$ and a default activity $\dot{x}_l = 1$. Timed automata have shown success in modeling and verifying both hardware [42] and software [40] and are well-supported by many tools [17, 33, 35, 36, 41, 43, 81, 136]. Although the severe restriction makes the analysis easier but less expressive for real problems, the critical problems for nondeterministic timed automata are that, the language itself is not closed under complement, and the universality problem, checking whether the language of a given timed automaton over the alphabet A comprises all timed words over A, is undecidable [10, 137].

2.3.1.1 Hybrid automata semantics

Recall that the state of HA H is a pair $(\mathbf{l}, \mathbf{v})$ which specifies a control location and a valuation for all real variables. H can evolve in two ways. After a nonnegative time duration I, $\mathbf{v}$ changes to $\mathbf{v}'$ according to the location flow functions, the invariants associated with the location should still hold, and the location remains the same. This type of system evolution is named time progress, represented as $(\mathbf{l}, \mathbf{v}) \Rightarrow_t (\mathbf{l}, \mathbf{v}')$. At some time point t, when a flow violates invariants at a location and action predicates are enabled, H will take a transition progress through a transition e resulting in changing the control location and resetting some variables, represented as $(\mathbf{l}, \mathbf{v}) \Rightarrow_e (\mathbf{l}', \mathbf{v}')$. This type of discrete transition is normally assumed to be instantaneous, allowing multiple

discrete transitions to happen theoretically at the same time [151]. After this discrete jump, the continuous flow resumes. Then, we can aggregate both progress and define *simple progress* as $(\mathbf{l}, \mathbf{v}) \Rightarrow (\mathbf{l}_1, \mathbf{v}_1)$ iff there exists $\mathbf{v}', \mathbf{v}_1 \in V$ and $\mathbf{l}_1 \in Loc$ such that $(\mathbf{l}, \mathbf{v}) \Rightarrow_t (\mathbf{l}, \mathbf{v}') \Rightarrow_e (\mathbf{l}_1, \mathbf{v}_1)$. Moreover, the simple progress relation $\Rightarrow$ can be extended to sets of states. For instance, if $(\mathbf{l}, \mathbf{v}), (\mathbf{l}_1, \mathbf{v}_1) \in \Sigma$ and $\Sigma_1, \Sigma_2 \subseteq \Sigma$ define $(l, v) \Rightarrow \Sigma_1$ iff $(\mathbf{l}, \mathbf{v}) \Rightarrow (\mathbf{l}_1, \mathbf{v}_1) \wedge (\mathbf{l}_1, \mathbf{v}_1) \in \Sigma_1$. Also, define $\Sigma_1 \Rightarrow \Sigma_2$ iff $(\mathbf{l}, \mathbf{v}) \Rightarrow \Sigma_2 \wedge (\mathbf{l}, \mathbf{v}) \in \Sigma_1$.

Formally, the evolution of a hybrid system must follow the following conventions:

- The system starts at an initial state in Σ_{Init} at time $t_0 = 0$.
- Only the action predicates enable discrete transitions even if the flow in a location violates the defined invariants.
- Discrete transitions have priority when both discrete transitions and continuous evolution are able to proceed and during the continuous evolution the discrete location remains the same.

The parallel composition of two HA H_1 and H_2 over a common set of variable *Var* can be used to model complex hybrid systems. Similar to CFSMs, H_1 and H_2 communicate with each other through the mutual labels $Labels_1 \cap Labels_2$. $H_1||H_2$ uses special timed semantics and time-abstract semantics to keep things simple: if both share the label a, they must synchronize on a-transitions; otherwise, a transition takes 0 duration for the other HA. Also, both HA share the same duration during the flow transition. The resulting $H_1||H_2$ is still a hybrid system defined on $Loc_1 \times Loc_2$, Var, $Labels_1 \cup Labels_2$, and $\Sigma_{Init_1} \times \Sigma_{Init_2}$. The transition edges contain three cases: H_1 makes transition progress, H_2 makes transition progress, or both make a move on the shared synchronization label. All the other predicates and conditions are the conjunction of those from

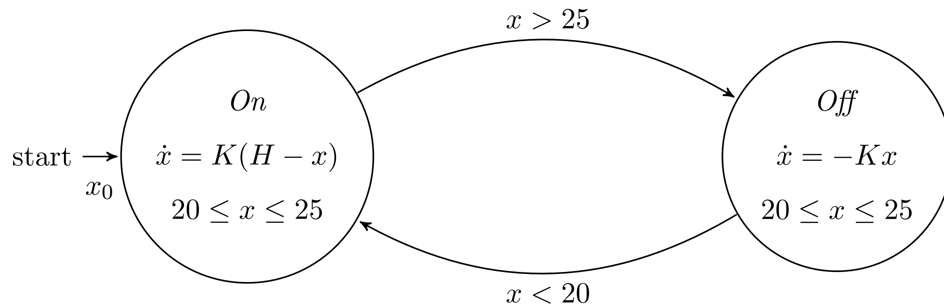

Figure 2.7: The HA for a thermostat controlling the indoor room temperature using a heater which is represented as a continuous variable x. A heating device will be automatically turned on if the room temperature, x, drops below 20°C and be turned off once x reaches 25°C. The temperature will fall according to $\dot{x} = -Kx$ without the heater and rises according to $\dot{x} = K(H - x)$, where K and H are constants for thermodynamics.

H_1 and H_2.

2.3.1.2 A thermostat example

Here, we use a simple example to demonstrate how to model hybrid systems using HA. A thermostat is an automatic control system to regulate indoor temperature using heating devices, shown in Figure 2.7. The system constantly monitors the room temperature and decides to turn a heater on or off. The thermodynamics are governed by differential equations on room temperature represented by a continuous variable x. We simply assume that x decreases according to the exponential function $x(t) = x_0 \cdot e^{-Kt}$ when the heater is off and increases according to $x(t) = x_0 \cdot e^{-Kt} + H(1 - e^{-Kt})$ where x_0 is the initial room temperature, constant H is determined by the power of the heater, and constant K is adjusted according to the dimension of the room [6]. Thus, the flow activity is $\dot{x} = K(H - x)$ for control location *on* and $\dot{x} = -Kx$ for location *off*. The heater will be turned on if the room temperature drops below 20°C and be turned off if the temperature rises above 25°C. We assume the heater is on in the initial state. Thus, the thermostat system is a hybrid automaton with:

- $Loc = \{On, Off\}$;
- $Var = \{x\}$ where $V = \mathbb{R}$;
- $Labels = \emptyset$;
- $\Sigma_{Init} = \{On \times V\}$;
- $Inv(On) = Inv(Off) = \{x \in V : 20 \leq x \leq 25\}$;
- $Flow(On) = \{x \in V : \dot{x} = K(H - x)\}$ and $Flow(Off) = \{x \in V : \dot{x} = -Kx\}$;
- $Edges = \{(On, Off), (Off, On)\}$;
- $Jump(On, Off) = \{x \in V : x > 25, x' = x\}$ and $Jump(Off, On) = \{x \in V : x < 20, x' = x\}$.

Formalisms such as timed automata [10] and HA [8] emphasize the underlying mathematical principles and discrete transition structures of design systems. This provides them with rigorous theoretical background, but less practical problem solving ability. Especially when modeling real hybrid systems, their lack of ability to handle compositional construction and abstraction hinders their adoption in solving real problems [31]. However, if we analyze hybrid systems only by their physical behaviors with respect to time, this eases the burden of analyzing mathematical equations and complex hierarchical composition structures and provides us with visual results of how hybrid systems evolve.

For instance, Figure 2.8 shows a simulation of the thermostat system in Figure 2.7 across 24 hours with initial temperature $x_0 = 21$ at t_0, $H = 32$, and $K = 0.2$. The system has 12 discrete jumps marked as switching points. The heater is on at initial state and stays on during the time interval $I_0 = [t_0, t_1]$. At time t_1, the temperature

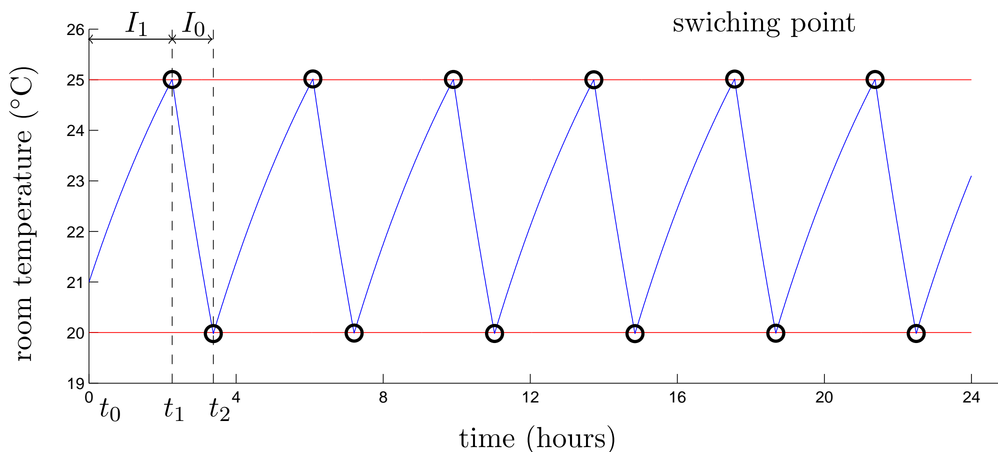

Figure 2.8: One simulation result of the thermostat in Figure 2.7 in the duration of 24 hours with initial room temperature $x_0 = 21$ and the heater is on. The values chosen for the constants are $H = 32$ and $K = 0.2$. All switching points are marked by dark circles.

is over 25°C and the discrete transition switches off the heater. The heater remains off during $I_1 = [t_1, t_2]$, until the temperature drops below 20°C. The system turns the heater back on at time t_2. The fluctuation of temperature is always within the range [19,26] and the system achieves its purpose of temperature regulation. This type of modeling and simulation analysis strategy is explained in the next section.

2.3.2 Signals and hybrid systems

Analyzing hybrid systems based on simulation signals gives us more flexibility to handle and analyze larger or even industrial-sized hybrid systems. This simulation-based modeling method tries to model hybrid systems in a more computationally scientific way and provides a view using numerical algorithms and approximations. This method relies on numerical integration algorithms [179], called *solvers* to compute the system dynamics over time at discrete time points. Between discrete time points, linear or polynomial interpolation will be used to provide the numerical solution within

a predefined precision. Intervals between discrete time points are called steps, the size of which could be fixed or variable according to solvers. In MATLAB [174], Simulink and Stateflow provide continuous and discrete simulations, respectively. The methods integrate well with the MBD trend.

We define a signal, also named *trajectory*, as a function mapping the time domain $\mathbb{T} = \mathbb{R}^{\geq 0}$ to the reals $\mathbb{R}$. Boolean signals, used to represent discrete dynamics, are signals whose values are restricted to false (denoted $\perp$) and true (denoted $\top$). Likewise, a multi-dimensional signal $\mathbf{x}$ is a function from $\mathbb{T}$ to $\mathbb{R}^n$ such that $\forall t \in \mathbb{T}$, $\mathbf{x}(t) = (x_1(t), \cdots, x_n(t))$. A hybrid system $\mathcal{S}$ (such as a Simulink model [186]) is an input-output mapping: it maps a set of initial operating conditions and input signals $\mathbf{u}(t)$ to output signal $\mathbf{x}(t)$. A trace is a collection of output and input signals resulting from the simulation of a system which can be viewed as a multi-dimensional signal. An implicit assumption for analyzing hybrid systems only using signals is that, for each special initial condition and input signals $\mathbf{u}(t)$, the computed output signals $\mathbf{x}(t)$ have to be unique [3, 210]. In other words, the considered system has to be deterministic or the numerical algorithms for simulation have to be deterministic.

2.3.2.1 A four-speed automatic transmission example

Consider as an example a closed-loop model designed for a four-speed automatic transmission controller of a vehicle as shown in Figure 2.9. Although this model is not a real industrial model, it has all the necessary mechanical components: models for an engine controller, a transmission controller, and a plant model for a vehicle. The transmission block computes the transmission ratio (Ti) using the current gear status and computes the output torque from the engine speed (Ne), the gear status, and the

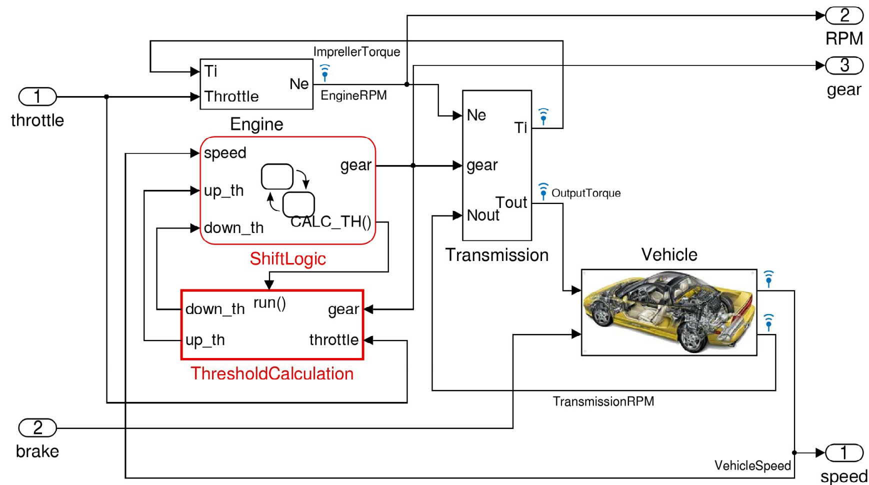

Figure 2.9: The closed-loop Simulink model of an automatic transmission controller. The inputs to the model are the throttle position and the brake torque. The outputs are gear position, engine speed in rpm, and the vehicle speed in mph. The discrete gear shift logic is done using Stateflow. The continuous plant model in *Vehicle* modular is constructed using continuous block in Simulink.

transmission `RPM`. The other two blocks represent the gear shift logic and the related threshold speed calculation. The model takes the percentage of the `throttle` position and the `brake` torque as inputs.

The transmission controller has four gears, and the system switches from gear i up to gear $i+1$ or down to gear $i-1$ based on certain conditions on the current gear i, the current speed of the vehicle, and the applied throttle. Switching gears is necessary for energy efficiency because the engine generates less power at very low or very high speeds. Thus, for efficiency, at a certain gear position the automatic transmission will shift gear up or down when the engine speed reaches the threshold speed. The threshold speed for each gear is specified in a look-up table that is indexed by the current gear and the applied throttle. The interesting signals are the vehicle `speed` measured in miles per hour (mph), the transmission `gear` position, and engine speed measured in rotations per minute (rpm).

The correctness of the design is confirmed if all possible system behaviors are

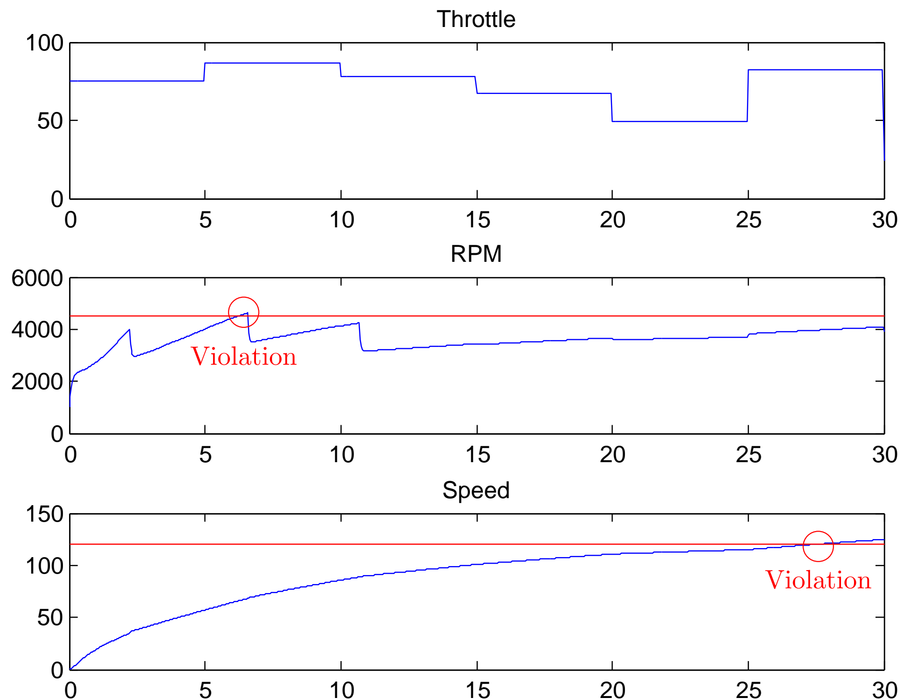

Figure 2.10: Falsifying trace for the automatic transmission controller and the requirement that RPM never goes beyond 4500 rpm or speed beyond 120 mph.

as expected. However, each simulation result is an individual instance of the system behavior. Moreover, the potential state space is normally infinite, so it is impossible to obtain all possible simulation results. Thus, the problem of validation is more practical. Suppose the designer of this transmission system wants to ensure the requirement that the engine speed never exceeds 4500 rpm and that the vehicle never drives faster than 120 mph. After simulating the closed-loop system we can show that these requirements are not met, as illustrated in Figure 2.10.

However, the straightforward result for a human being is not obvious for a machine. Later, in Chapter 4, rigorous temporal logic will be introduced to specify acceptable behaviors and automatically verify whether a trace satisfied the specification. Moreover, this negative result does not provide further insight into the model. If a requirement does not hold, we would like to know what does hold for the controller, and

by how much the controller misses the requirements. Such a characterization would shed more light on the workings of the system, especially in the context of legacy systems and for reverse engineering the behavior of a very complex system. In the context of this example, it would help to know the maximum `speed` and `RPM` that the model can reach, or the minimum dwell time that the transmission enforces to avoid frequent gear shifts. In Chapter 5, we present a technique to automatically obtain such requirements from the model.

2.4 Conclusion

In this chapter, we focused on modeling methods for hybrid systems. Since hybrid systems contain both continuous dynamics and discrete transitions, each modeling method has its own perspective for viewing hybrid systems and has its own advantages and disadvantages. Discretizing the whole system results in a discrete system such as an NCFSM. This simplifies the considered system and reduces the potential state space to be analyzed in the future. ECA rules provide more powerful syntax and semantics to model both the environment and the system due to their expressive power to describe complex events and conditions. However, discretization ignores inherent continuous dynamics and can cause critical errors when analyzing time-sensitive systems. On the other hand, directly modeling hybrid systems can be done using HA, which have rigid theoretical and mathematical foundations. However, the syntax and semantics do not have good support for hierarchical compositional design; also the analysis of HA without severe constraints is NP-hard. All these facts limit their usage in real applications. A promising modeling method that is widely adopted by industry treats hybrid systems as signals. This largely releases the burden of analyzing complex continuous dynamics

and hierarchical composition structures and uses numerical algorithms to compute the trajectories of the system. However, the numerical precision and the algorithm stability are the remaining problems. Also, the deterministic requirement is another restriction of modeling hybrid systems. All the modeling methods introduced in this chapter will be used to explore techniques to analyze hybrid systems in the following chapters.

Chapter 3

Symbolic Verification of Discrete Systems

Traditional techniques to analyze discrete systems can be classified as either explicit or symbolic. Explicit methods enumerate and store each state individually. They normally use graph-based search algorithms such as breadth-first search (BFS) or depth-first search (DFS) to explore the state space and to verify its properties. Symbolic methods, on the other hand, encode states implicitly by using compact data structures, called decision diagrams. Symbolic encoding also provides efficient manipulation algorithms for state sets. Although both techniques still suffer from the state space explosion problem, they also provide different solutions to reduce its impact. Explicit methods exploit symmetries [70, 71] within the model and use partial order reduction [95, 101, 117, 202] to effectively reduce the state space. Symbolic methods naturally have the advantage that they are able to encode large state sets compactly without losing any information. Research efforts focus on how to reduce the peak size of the data structure during the computation [49, 58], how to find a good encoding strategy [24, 38, 98, 59], and how to

increase the convergence speed [168, 177, 190]. In this chapter, we mainly focus on using symbolic techniques to improve both capability and efficiency when verifying discrete systems.

3.1 Decision diagrams

The key ingredient of symbolic approaches is to use decision diagrams, more often referred to as binary decision diagrams (BDDs) [47]. BDDs were originally designed to encode and manipulate binary functions. These methods have been widely adopted and evolved to new forms, such as multi-way decision diagrams (MDDs) [130] and *edge-valued multi-way decision diagrams* (EV^+MDDs) , and successfully used to analyze circuits [48, 50, 114], concurrent systems [78], probabilistic systems [134], and real-time systems [113] for the past few decades. We choose to use MDDs to encode boolean functions for sets and EV^+MDDs [64] to encode partial integer functions where ∞ means "undefined".

Given L domain variables v_l $(1 \leq l \leq L)$ having finite domain $\mathcal{V}_{v_l}$ and a boolean range variable v_0, ordered $v_L \succ \cdots \succ v_1 \succ v_0$, a (quasi-reduced) MDD is a directed acyclic edge-labeled graph where:

- Each nonterminal node p is associated with a domain variable v_l. We write $p.v = v_l$.
- The terminal nodes are $\mathbf{0}$ and $\mathbf{1}$, and are the only nodes with $\mathbf{0}.v = \mathbf{1}.v = v_0$.
- A nonterminal node p with $p.v = v_l$ has, for each $i \in \mathcal{V}_{v_l}$, an edge pointing to node q, with either $q.v = v_{l-1}$ or $q = \mathbf{0}$. We write $p[i] = q$. We must have at least one $p[i] \neq \mathbf{0}$.

- For canonicity, there are no duplicates: given two nonterminal nodes p and q with $p.v = q.v$, there must be at least one $i \in \mathcal{V}_{p.v}$ such that $p[i] \neq q[i]$.

A nonterminal MDD node p with $p.v = v_l$ encodes the set of tuples recursively defined by $\mathcal{B}_p = \bigcup_{i \in \mathcal{V}_{v_l}} \{i\} \cdot \mathcal{B}_{p[i]}$, with terminal cases $\mathcal{B}_{\mathbf{0}} = \emptyset$, the empty set, and $\mathcal{B}_{\mathbf{1}} = \{\epsilon\}$, the empty tuple, where "$\cdot$" indicates tuple concatenation.

To encode partial integer functions, we need a variant of the above. A normalized EV$^+$MDD [64] is a directed acyclic edge-labeled graph where:

- Ω is the only terminal node, with $\Omega.v = v_0$.
- A nonterminal node a with $a.v = v_l$ has, for each $i \in \mathcal{V}_{v_l}$, an edge labeled with $\rho \in \mathbb{N} \cup \{\infty\}$ pointing to node b. We write $a[i] = \langle \rho, b \rangle$, $b = a[i].node$, and $\rho = a[i].val$. We must have $b = \Omega$ if $\rho = \infty$, $b.v = v_{l-1}$ otherwise, and at least one $a[i].val = 0$.
- For canonicity, there are no duplicates: given two nonterminal nodes a and b with $a.v = b.v$, there must be at least one $i \in \mathcal{V}_{p.v}$ such that $a[i] \neq b[i]$, i.e., either $a[i].node \neq b[i].node$, or $a[i].val \neq b[i].val$, or both.

Given EV$^+$MDD node a with $a.v = v_l$ and $n \in \mathbb{N}$, $\langle n,a \rangle$ encodes the function $f_{\langle n,a \rangle} : \mathcal{V}_{v_l} \times \cdots \times \mathcal{V}_{v_1} \rightarrow \mathbb{N} \cup \{\infty\}$ recursively defined by $f_{\langle n,a \rangle} = n + f_{a[v_l]}$, with base case $f_{\langle n,\Omega \rangle} = n$.

Union(a, b) and *Intersection*(a, b) are two operators used to compute the MDD encoding $\mathcal{B}_a \cup \mathcal{B}_b$ and $\mathcal{B}_a \cap \mathcal{B}_b$. Analogously, *Minimum*$(\langle n,a \rangle, \langle m,b \rangle)$, returns the EV$^+$MDD encoding $\min(f_{\langle n,a \rangle}, f_{\langle m,b \rangle})$, and *Normalize* puts an EV$^+$MDD in canonical form [64], i.e., normalizes edge values so that at least one is zero for each node, while still encoding the same function.

To make MDDs more compact and their manipulation more efficient, edges can skip variables under various reduction rules [47, 204]. These rules ensure canonicity and implicitly define the meaning of skipping edges. We associate a reduction rule $RR(v_l) \in \{F, I, Q\}$ to each variable v_l with the following interpretations:

- $RR(v_l) = F$. The fully-reduced rule forbids redundant nodes. Node p, associated with v_l, is a redundant node, if and only if for all $i \in \mathcal{V}_{v_l}$, $p[i]$ lead to the same node.
- $RR(v_l) = I$. The identity-reduced rule forbids singular nodes. If node q has the only outgoing edge $q[i_q] \neq \mathbf{0}$ and edge $p[i_q]$ points to q, node q is a singular node.
- $RR(v_l) = Q$. The quasi-reduced rule forbids edges skipping variables. For any $i \in \mathcal{V}_{p_v}$, if $p[i] = q \neq \mathbf{0}$ we have $p.v \succ q.v$ in the predefined variable order list.

The same technique can be applied to EV$^+$MDDs.

3.2 LTL and CTL

Temporal logics provide a formalism to describe the properties of discrete systems. It extends first-order logic with temporal operators. Computational tree logic (CTL) [69, 74] and linear temporal logic (LTL) [170] are two widely used logics. CTL considers the system evolution as a branching tree with potentially different future branches, while LTL treats it as a set of evolution paths, so that for each of these path the future lies in one direction. Both are defined on a set of atomic propositions AP with a labeling function which maps each state to a subset of AP. Temporal formulas are formed using the following temporal operators: “always” (denoted as G), “eventually” (denoted as F), “next” (denoted as X), and “until” (denoted as U).

Definition 3.1. (LTL syntax) An LTL formula over a set of atomic propositions AP is inductively defined using the following grammar:

$$\varphi := \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\,\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

where a is an atomic proposition in set AP. □

Both the "eventually" operator F and the "always" operator G can be derived from the "until" operator U, as follows:

$$\mathsf{F}\,\varphi = \top \,\mathsf{U}\, \varphi \tag{3.1}$$

$$\mathsf{G}\,\varphi = \neg \mathsf{F}\, \neg\varphi \tag{3.2}$$

An LTL formula describes the property of paths, infinite behaviors. The semantics of LTL formula φ is defined as a language that contains all infinite words over 2^{AP} satisfying φ. Here, given an infinite word $\sigma = A_0A_1A_2\ldots \in (2^{AP})^\omega$, $\sigma^i = A_iA_{i+1}A_{i+2}\ldots$ denotes the suffix of σ starting from symbol A_i.

Definition 3.2. (LTL semantics) Given an infinite word $\sigma = A_0A_1A_2\ldots$ over a set of atomic propositions AP, the LTL semantics of formula φ are defined as follows:

$$\begin{array}{lll}
\sigma \models a & \text{iff} & A_0 \models a \;\; (a \in A_0) \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists\, j \geq 0 \;\text{ s.t. } \sigma^j \models \varphi_2 \text{ and } \forall\, 0 \leq i < j, \sigma^i \models \varphi_1
\end{array}$$

where $a \in AP$. □

The derived operators have then the following semantics:

$$\begin{array}{lll}
\sigma \models \mathsf{F}\varphi & \text{iff} & \exists\, j \geq 0 \;\text{ s.t. } \sigma^j \models \varphi \\
\sigma \models \mathsf{G}\varphi & \text{iff} & \forall\, j \geq 0 \;\text{ s.t. } \sigma^j \models \varphi
\end{array}$$

The LTL is linear due to its qualitative interpretation of time: at each point in time there is only one possible successor state. Thus, the interpretation of an LTL formula can be also defined on an infinite sequence of states. Each state s is associated with a set of atomic propositions that is satisfied in that state. We define label function $L(s)$ which returns the satisfied atomic proposition set in state s. Instead of an infinite path, CTL considers a branching notion of time [26] at each state. Thus, there may exist many possible successor states for each state, and we define function $Path(s)$ which returns all paths starting from s. Then, an infinite word equals to one possible infinite path starting from s and we have $\sigma = s_0 s_1 s_2 \ldots$. We use function $\sigma[i] = s_i$ to denote the ith state of a path. CTL defines path quantifier "existential" (denoted as E) and "universal" (denoted as A) to allow CTL formulas to express "some" or "all" computations from a state in an infinitely branching tree.

Definition 3.3. (CTL syntax) A CTL state formula is inductively defined using the following grammar:

$$\phi := \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{A}\,\varphi \mid \mathsf{E}\,\varphi$$

where $a \in AP$ and φ is a *path formula* formed using the following grammar:

$$\varphi := \mathsf{X}\,\phi \mid \phi_1\,\mathsf{U}\,\phi_2$$

where ϕ, ϕ_1, ϕ_2 are state formulas. $\Box$

Definition 3.4. (CTL semantics) Given a set of atomic propositions AP, label function L, and function $Path(s)$ for generating all paths starting from s, the CTL semantics

of a state formula ϕ at a state s are defined as follows:

$$
\begin{array}{lll}
s \models a & \text{iff} & a \in L(s) \\
s \models \neg\phi & \text{iff} & s \not\models \phi \\
s \models \phi_1 \wedge \phi_2 & \text{iff} & s \models \phi_1 \text{ and } s \models \phi_2 \\
s \models \mathsf{E}\,\varphi & \text{iff} & \exists\,\sigma \in \mathit{Path}(s) \;\; \text{s.t. } \sigma \models \varphi \\
s \models \mathsf{A}\,\varphi & \text{iff} & \forall\,\sigma \in \mathit{Path}(s) \;\; \text{s.t. } \sigma \models \varphi
\end{array}
$$

and

$$
\begin{array}{lll}
\sigma \models \mathsf{X}\phi & \text{iff} & \sigma[1] \models \phi \\
\sigma \models \phi_1 \,\mathsf{U}\, \phi_2 & \text{iff} & \exists\, j \geq 0 \;\; \text{s.t. } \sigma[j] \models \phi_2 \text{ and } \forall\, 0 \leq i < j, \sigma[i] \models \phi_1
\end{array}
$$

where σ is a path from s; φ is a path formula; ϕ, ϕ_1, and ϕ_2 are state formulas. □

Also, from the definitions of F and G operators in (3.1) and (3.2), we can derive the following operators:

$$\mathsf{AF}\,\phi = \mathsf{A}(\top\,\mathsf{U}\,\phi) \tag{3.3}$$

$$\mathsf{EF}\,\phi = \mathsf{E}(\top\,\mathsf{U}\,\phi) \tag{3.4}$$

$$\mathsf{AG}\,\phi = \neg\mathsf{EF}\neg\phi \tag{3.5}$$

$$\mathsf{EG}\,\phi = \neg\mathsf{AG}\neg\phi \tag{3.6}$$

Both CTL and LTL have shown their success in software and hardware verification [52, 94, 105]. Many properties can be specified using either CTL or LTL, but their expressiveness is incomparable [68]. Some CTL formulas have no equivalent LTL formulas and vice versa.

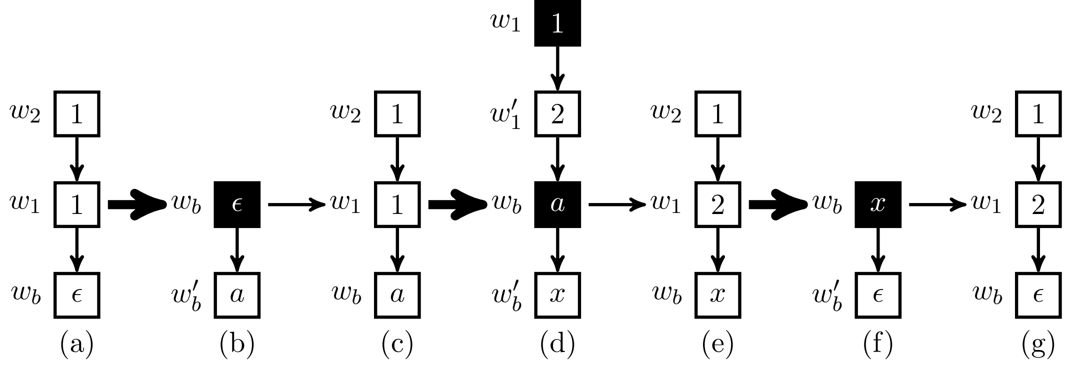

Figure 3.1: A simple example of system evolution for M_{ex} in Figure 2.3. Figures (a), (c), (e), and (g) are MDDs encoding states while (b), (d), and (f) are MDDs encoding transitions. Shaded arrows mean to apply transitions to states and slim arrows point to the result.

3.3 Symbolic verification of a network of communicating FSMs

3.3.1 Symbolic encoding of an NCFSM

Given an NCFSM with K component CFSMs and a system buffer, variables $(w_K,\ldots,w_1,w_b)$ describe the global state. The first K variables individually store the local state components of each CFSM, while w_b stores the current buffer content. In the following, we use the quasi-reduced rule for MDDs encoding the set of global states $\mathcal{S}$. The initial global state $\mathbf{s}_{init}$ of M_{ex} in Figure 2.3 is shown in Figure 3.1(a). For all MDD figures, we only display paths leading to terminal **1** and omit the terminal nodes.

We capture the global state transition function δ and output function λ with a next-state function $\mathcal{T} : \mathcal{S} \times (\mathcal{X} \cup \{\epsilon\}) \to \mathcal{S} \times (\mathcal{Y} \cup \{\epsilon\})$ encoded using MDDs on $2(K+1)$ variables $(w_K, w'_K, ..., w_1, w'_1, w_b, w'_b)$, so that $\mathcal{T}(x, y) = (x', y')$ iff $\delta(x, y) = x'$ and $\lambda(x, y) = y'$, where x and x' are stable or unstable global states and y and y' are symbols in $\mathcal{X} \cup \mathcal{Y} \cup \{\epsilon\}$. The unprimed and primed variables represent the interleaved

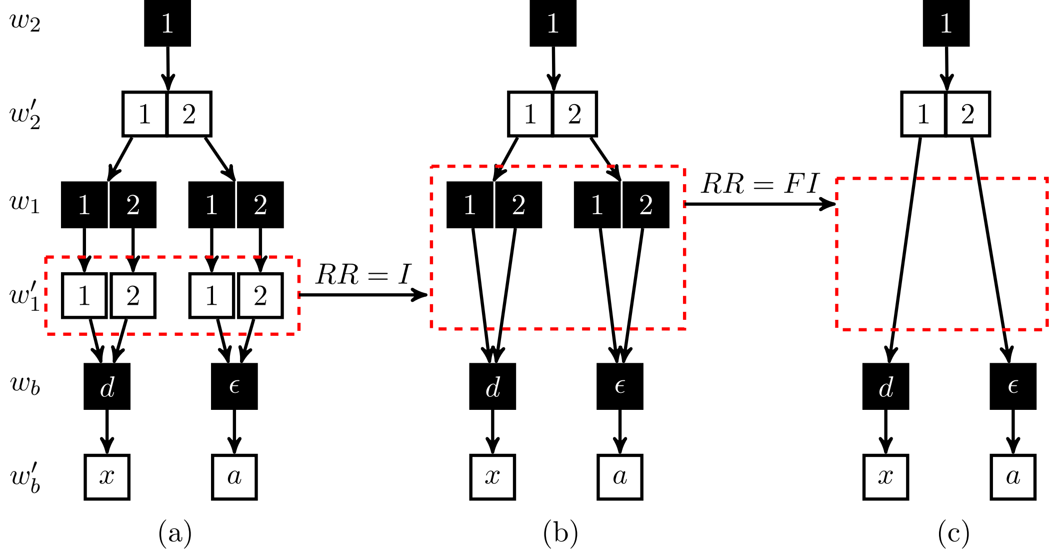

Figure 3.2: A demonstration of applying fully-reduced and identity-reduced rules. Figure (a) encodes transitions in quasi-reduced rule. Applying identity-reduced rule on variable w_1' in (a) results in (b) and applying fully-reduced rule on variable w_1 in (b) results in (c).

"from" and "to" global states, distinguished by their background color (black for "from" and white for "to"). w_k and w_k' describe states of the same component. Thus, they share the same variable domain $\mathcal{V}_{w_k}$ and variable order position. w_b and w_b' correspond to input and output symbols in buffer β. In conclusion, we can symbolically encode $\mathcal{S}$ and $\mathcal{T}$ using $(K+1)$-variable and $2(K+1)$-variable MDDs, respectively (types *mdd* and *mdd*2 in the following algorithms).

Transitions that affect a given component only change the corresponding variable (we call this *locality*). Thus, we store the MDDs encoding $\mathcal{T}$ in disjunctive form as $\mathcal{T} = (\bigcup_{1 \leq k \leq K} \mathcal{T}_k) \cup \mathcal{T}_\beta$, where $\mathcal{T}_k$ encodes the next-state function of M_k, and $\mathcal{T}_\beta$ encodes the interaction between the system and the environment. For further efficiency, we adopt the quasi-fully-identity-reduced (QFI) rule [205] to more compactly encode each disjunct. Take partial transitions for example. The quasi-reduced encoded MDD is shown in Figure 3.2(a). Nodes at variable w_1 and w_1' qualify for identity reduction since edges from w_1 to w_1' have the same indexes and all the edges point to the same

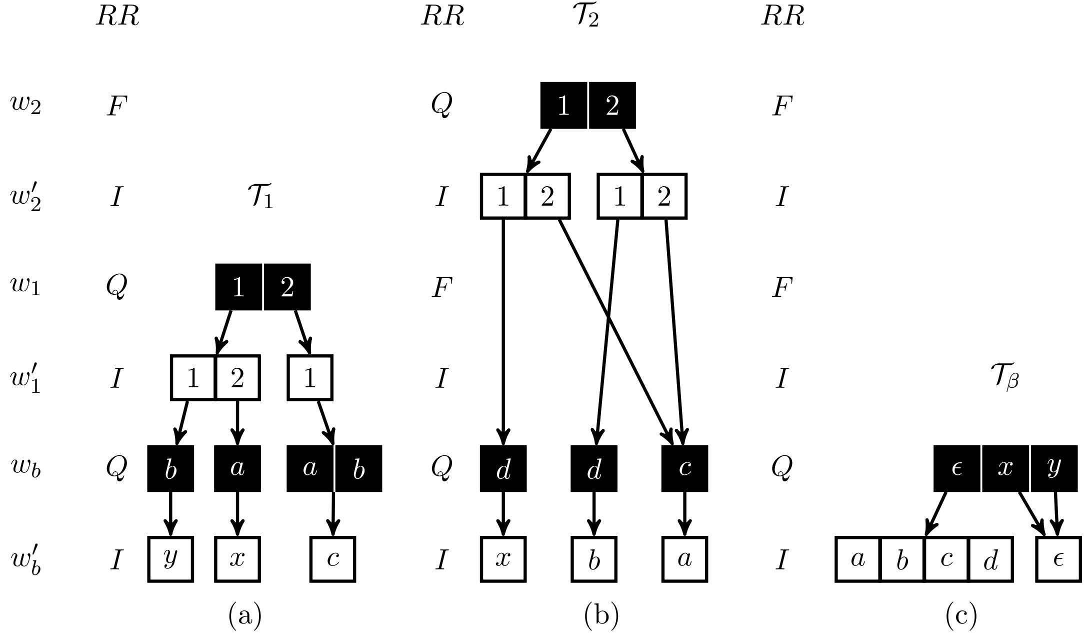

Figure 3.3: QFI-reduced MDDs encoding transition $\mathcal{T}$ of M_{ex} in Figure 2.3, where (a) for $\mathcal{T}_1$, (b) for $\mathcal{T}_2$, and (c) for $\mathcal{T}_\beta$.

node. The resulting I-reduced MDD in (b), is amenable to fully-reduced rule on variable w_1 since the domain of w_1 is $\{1, 2\}$, all nodes for w_1 have full edges, and all edges for each node lead to the same child. The final QFI-reduced MDDs in (c) encode the same information as Q-reduced MDDs but use fewer nodes. Thus, if the interpretation of the missing nodes along these long jumping edges is well-defined, applying the appropriate reduction rule can save memory while encoding the same amount of information. Applying the QFI-rule to $\mathcal{T}$ of M_{ex} in Figure 2.3, the results are shown in Figure 3.3, where (a) for $\mathcal{T}_1$, (b) for $\mathcal{T}_2$, and (c) for $\mathcal{T}_\beta$. MDDs following the QFI-reduced rule have a compact encoding of transitions and are easy to build. QFI-rule will be extensively used in advanced algorithms to achieve great improvement, as discussed in the next section.

Next, we use Figure 3.1 to show how to apply $\mathcal{T}$ to $\mathcal{S}$ symbolically. Here, shaded arrows signify an application of $\mathcal{T}$ and slim arrows point to the result. The system starts from $\mathbf{s}_{init}$ of NCFSM M_{ex} in (a). Applying (b), the next-state function encoding the

arrival of external input symbol a, results in a new global state (c), with two unchanged local states and symbol a in buffer β. Then, next-state function (d) becomes enabled, which changes the local state of M_1 and outputs symbol x, as the result state in (e). Due to x is an external output symbol, it is absorbed by the environment (f), and the system reaches another stable global state (g). Thus, (c)-(e) together correspond to the step $a.(1,1)\,[\underline{a}/x\rangle\,(1,2)$ and (a)-(e) represent stable global transition $(1,1)\,[\![a/x\rangle\!\rangle\,(1,2)$. While the sets manipulated in this illustrative example contain only a single state, symbolic algorithms normally manipulate very large sets of states when applied to real models.

3.3.2 Reachability analysis for an NCFSM

Reachability analysis is the fundamental step in the study of discrete systems. With symbolic encoding, applying transition relations to a set of reachable states obtains the next set of reachable states. The simplest algorithm for reachability analysis is based on BFS [157], which starts from an initial global state set $\mathbf{s}_{init}$ and repeatedly applies next-state function $\mathcal{T}$ on currently reachable states to search successive reachable states, until no more can be found. Thus, the reachable state set is $\mathcal{S}_{rch} = \{\mathbf{s}_{init}\} \cup \mathcal{T}(\mathbf{s}_{init}) \cup \mathcal{T}^2(\mathbf{s}_{init}) \cup \cdots$. Procedure *BFS* contains these heavyweight global fixed-point iterations at Lines 2-6 of Figure 3.4. Procedure *RelProd* applies transition to states to generate the next reachable states. Procedure *BFS* calls *RelProd* with Line 14b.

Due to locality, the saturation algorithm [60], shown as Procedure *Saturate* in Figure 3.4, tends to be much more efficient than BFS for asynchronous discrete-state systems. The MDD roots of the disjuncts in the QFI encoding of $\mathcal{T} = (\bigcup_{1 \leq k \leq K} \mathcal{T}_k) \cup \mathcal{T}_\beta$ are assigned to distinct variables. Saturation exploits the property that $\mathcal{T}_k$ cannot change any state variable above w_k, and proceeds in phases, from the bottom variable w_b up to

```
mdd  BFS(mdd s_init, mdd2 r)
 1   mdd s ← s_init;
 2   repeat
 3     mdd s_pre ← s;
 4     mdd s_next ← RelProd(s_pre, r);      • Use 14b to generate the next reachable
     state set
 5     s ← Union(s_pre, s_next);
 6   until s = s_pre;                        • Reach the fixed-point
 7   return s;
mdd  RelProd(mdd s, mdd2 r)
 8   if s = 1 then return s ∧ r;
 9   mdd t ← 0;
10   if CacheLookUp(RelProdCode, s, r, t) then return t;
11   foreach i ∈ V_{s.v} s.t. s[i], r[i] ≠ 0 do
12     foreach i' ∈ V_{r[i].v} s.t. r[i][i'] ≠ 0 do
13       t[i'] ← Union(t[i'], RelProd(s[i], r[i][i']));
14b  t ← UniqueTableInsert(t);                     • for BFS
14s  t ← Saturate(UniqueTableInsert(t));          • for Saturation
15   CacheInsert(RelProdCode, s, r, t);
16   return t;
mdd  Saturate(mdd s)
17   if s.v = v_0 then return s;
18   mdd t ← 0;
19   if CacheLookUp(SaturateCode, s, t) then return t;
20   foreach i ∈ V_{s.v} s.t. s[i] ≠ 0 do
21     t[i] ← Saturate(s[i]);                   • saturate its children
22   repeat
23     foreach r ∈ T s.t. r.v = s.v, t[i] ≠ 0 and r[i] ≠ 0 do
24       foreach i' ∈ V_{r[i].v} s.t. r[i][i'] ≠ 0 do
25         t[i'] ← Union(t[i'], RelProd(t[i], t[i][i']));   • use 14s
26   until t does not change;
27   t ← UniqueTableInsert(t);
28   CacheInsert(SaturateCode, s, t);
29   return t;
```

Figure 3.4: The BFS and saturation algorithms for reachability analysis.

w_K. At the phase considering a particular variable w_k, it "saturates" all MDD nodes associated with it by applying $\mathcal{T}_k$ to them until no more new states can be found. If this creates a new node associated to a lower variable, it recursively saturates this new node first, by applying the appropriate disjuncts for lower level variables, before continuing the saturation of the current node associated with w_k. In this way, the saturation algorithm greedily exploits event locality and consists of exhaustive lightweight local

fixed-point iterations. Lines 20-21 first saturate all children of s and then Lines 22-26 generate the saturated next reachable states by calling Procedure *RelProd* with Line 14s. As a result, the algorithm significantly reduces the number of intermediate nodes which disappear in the final MDDs and is often orders of magnitude more efficient in memory and runtime than BFS. It is often able to generate state spaces with 10^{20} or even 10^{100} states in a matter of seconds or minutes.

All MDD algorithms fall in the dynamic programming category, and caches are heavily used to save intermediate results for efficiency. Procedures *CacheLookUp* and *CacheInsert* are for this purpose. Procedure *UniqueTableInsert* uses a hash table to ensure the canonicity of MDDs. For more details, interested readers can refer to [61].

Procedure *Saturate* stops when the root, associated with w_K, is saturated and the saturation result encodes the desired $\mathcal{S}_{rch}$. Then, it is easy to split $\mathcal{S}_{rch}$ into $\mathcal{S}_{st}$ and $\mathcal{S}_{unst}$ by checking the content of w_b. If $w_b = \epsilon$, the global state is stable, otherwise it is unstable. This is symbolically done through the *Intersection* operation mentioned in Section 3.1.

3.3.3 Symbolic observable product machine generation

In order to symbolically generate the observable product machine for an NCFSM, defined in Section 2.2.1.2, the next step is to develop an algorithm to compute the stable next-state function $\mathcal{T}_{obs}$ encoding both δ_{obs} and λ_{obs}. We build it from the perspective of transition relations, instead of from reachable global states. First, we define the unstable transitive closure (UTC):

Definition 3.5. (transition transitive closure) Given the next-state function $\mathcal{T}$ of NCFSM M, UTC is the smallest relation containing $\mathcal{T}$ and satisfying:

$$(c.\mathbf{j}, \epsilon) \in \mathcal{T}(b.\mathbf{i}, \epsilon) \wedge (b.\mathbf{i}, \epsilon) \in UTC(a.\mathbf{h}, \epsilon) \Rightarrow (c.\mathbf{j}, \epsilon) \in UTC(a.\mathbf{h}, \epsilon),$$

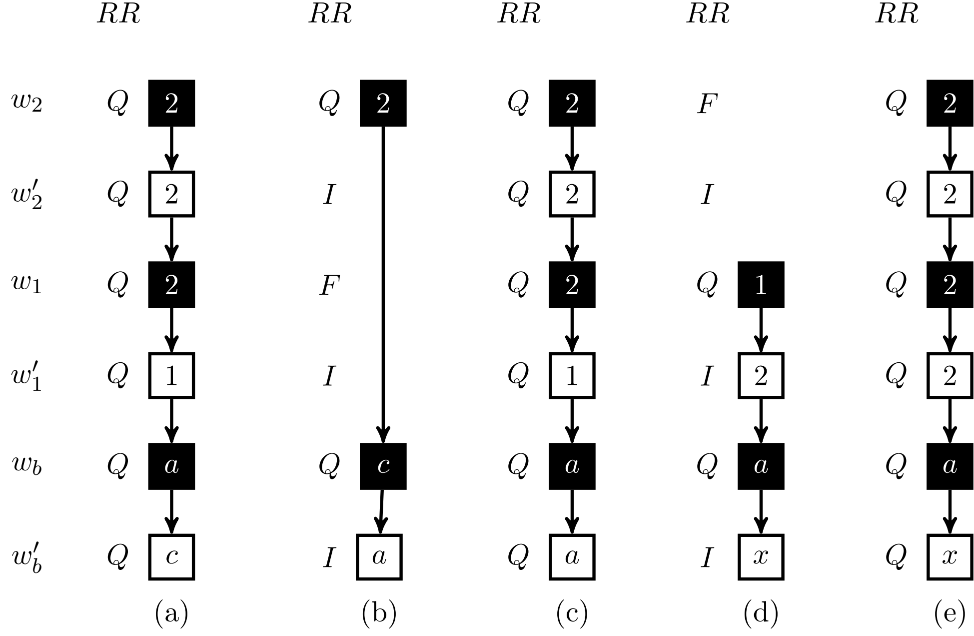

Figure 3.5: Composition operation on the transition relation

$$(c.\mathbf{j}, \epsilon) \in \mathcal{T}(b.\mathbf{i}, \epsilon) \wedge (b.\mathbf{i}, \epsilon) \in UTC(\mathbf{h}, a) \Rightarrow (c.\mathbf{j}, \epsilon) \in UTC(\mathbf{h}, a),$$

$$(\mathbf{j}, c) \in \mathcal{T}(b.\mathbf{i}, \epsilon) \wedge (b.\mathbf{i}, \epsilon) \in UTC(a.\mathbf{h}, \epsilon) \Rightarrow (\mathbf{j}, c) \in UTC(a.\mathbf{h}, \epsilon),$$

$$(\mathbf{j}, c) \in \mathcal{T}(b.\mathbf{i}, \epsilon) \wedge (b.\mathbf{i}, \epsilon) \in UTC(\mathbf{h}, a) \Rightarrow (\mathbf{j}, c) \in UTC(\mathbf{h}, a). \quad \square$$

Thus, UTC captures all transition sequences in M between global states that start from stable or unstable states to stable or unstable states and do not pass through stable global states. As with $\mathcal{T}$, UTC can be encoded using $2(K+1)$-variable MDDs. After building UTC, we can obtain its restriction to global stable transitions by applying the *Intersection* operator to select the elements with $w_b \in \mathcal{X}_{ext}$ and $w'_b \in \mathcal{Y}_{ext}$ as

$$\mathcal{T}_{obs} = \{(a.\mathbf{i}, b.\mathbf{j}) \in UTC : \mathbf{i}, \mathbf{j} \in \mathcal{S}_{st}, a \in \mathcal{X}_{ext}, b \in \mathcal{Y}_{ext}\}.$$

The calculation of UTC is based on combining the effect of sequentially firing several transitions into one transition. Figure 3.5 illustrates this operation on M_{ex}. Again, we assume that each set contains only one transition for better explanation. In

```
mdd  ComposeRelProd(mdd2  p, mdd2  r)
 1  if r = 1 or p = 1 then return p;
 2  mdd t ← 0, s ← 0;
 3  if CacheLookUp(ComposeRelProdCode, p, r, t) then return t;
 4  if p.v = r.v then
 5    foreach i, i' ∈ V_{p.v} s.t. p[i][i'] ≠ 0, r[i'] ≠ 0 do
 6      if r.v = r[i'].v then
 7        foreach j ∈ V_{r.v} do
 8          s ← ComposeRelProd(p[i][i'], r[i'][j]);
 9          t[i][j] ← Union(t[i][j], s);
10      else                                              • r[i'].v is I-reduced
11        s ← ComposeRelProd(p[i][i'], r[i']);
12        t[i][i'] ← Union(t[i][i'], s);
13  else                                                  • r.v is FI-reduced
14    foreach i, i' ∈ V_{p.v} s.t. p[i][i'] ≠ 0 do
15      s ← ComposeRelProd(p[i][i'], r);
16      t[i][i'] ← Union(t[i][i'], s);
17b t ← UniqueTableInsert(t);                              • for UTC_BFS
17s t ← UTC_Saturate(UniqueTableInsert(t));               • for UTC_Saturate
18  CacheInsert(ComposeRelProdCode, p, r, t);
19  return t;
```

Figure 3.6: The algorithm for the operation to combine the effect of sequential firing transitions into one transition.

(a), "to" state $\mathbf{i'} = (2, 1, c)$ is identical to one of the "from" states of (b). Recall that we are using the QFI-rule, so arcs skipping variables imply that the value of those variables is unchanged. We can combine the effect of (a) and (b), resulting in (c), which directly acts on $\mathbf{i} = (2, 2, a)$, the "from" state of (a), and moves to $\mathbf{j} = (2, 1, a)$, the "to" state of (b). Similarly, transition (e) is the result of further combination of transitions (c) and (d). With respect to the *UTC* definition, transitions (c) and (e) are in the *UTC* for (a). Procedure *ComposeRelProd* in Figure 3.6 is proposed to compute this composition effect of two next-state functions encoding in two $2(K + 1)$-variable MDDs. Compared with Procedure *RelProd* in Figure 3.4 which works on state sets, *ComposeRelProd* deals with two variables at a time instead of one variable.

As for reachability, *UTC* can be obtained by repeatedly applying Procedure

```
mdd2 UTC_BFS(mdd2 T)
 1  mdd2 z ← T, s ← 0, z_pre ← 0;
 2  repeat
 3    z_pre ← z;
 4    z_next ← ComposeRelProd(z_pre, T);                              • use 17b
 5    z ← Union(z_pre, z_next);
 6  until z = z_p;
 7  return z;

mdd2 UTC_Saturate(mdd2 z)
 8  if p.v = v_0 then return z;
 9  mdd2 t ← 0, s ← 0;
10  if CacheLookUp(UTC_SaturateCode, z, t) then return t;
11  foreach i, i' ∈ V_{p.v} s.t. p[i][i'] ≠ 0 do                     • V_{p.v} = V_{p[i].v}
12      t[i][i'] ← UTC_Saturate(p[i][i']);
13  repeat
14    foreach i, i' ∈ V_{p.v}, r ∈ T s.t. t.v = r.v, p[i][i'] ≠ 0, r[i'] ≠ 0 do
15        if r.v = r[i'].v then                                      • r: QFI-reduced
16          foreach j ∈ V_{r.v} s.t. r[i'][j] ≠ 0 do
17            s ← ComposeRelProd(p[i][i'], r[i'][j]);                 • use 17s
18            t[i][j] ← Union(t[i][j], s);
19        else                                                       • r[i'].v is I-reduced
20          s ← ComposeRelProd(p[i][i'], r[i']);                      • use 17s
21          t[i][i'] ← Union(t[i][i'], s);
22  until t does not change;
23  t ← UniqueTableInsert(t);
24  CacheInsert(UTC_SaturationCode, z, t);
25  return t;
```

Figure 3.7: The BFS and saturation algorithms for computing transition transitive closure.

ComposeRelProd to $\mathcal{T}$ until all possible merged next-state functions are found, as in the BFS algorithm, shown in Procedure *UTC_BFS* in Figure 3.7. Again, the global fixed-point computation is located at Lines 2-6 and uses *ComposeRelProd* on Line 17b. On the other hand, the saturation algorithm can be employed too, as Procedure *UTC_Saturate* in Figure 3.7. It first saturates the grandchildren, (the children's "to" nodes), at Line 12 and then at the current "from" node. *UTC_Saturate* operates on two successive variables in each recursion at Line 13-22 and uses *ComposeRelProd* with Lines 17s. Our experience shows that the more components an NCFSM has, the greater improvement saturation achieves.

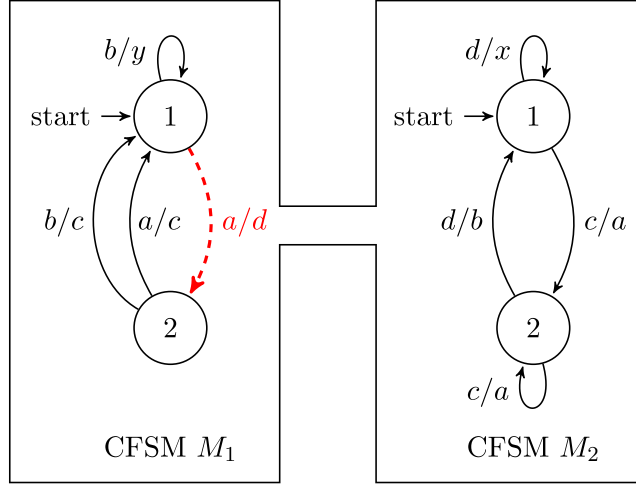

Figure 3.8: A simple variation of NCFSM M_{ex} in Figure 2.3. The small modification is that local transition $1\,[M_1, a/x\rangle\, 2$ is changed to $1\,[M_1, a/d\rangle\, 2$. The resulting new NCFSM contains a livelock when giving external input c at the initial state.

3.3.4 Symbolic livelocks verification

The difference between a livelock and a deadlock is that, in the former, the system keeps running instead of halting. Researchers [89] provided a sound, but incomplete solution for the livelock identification problem using integer programming. Explicit model checking tools, such as SPIN [118], solve this problem by enumerating non-progress cycles using reachability analysis.

An NCFSM does not terminate on an input if it reaches a livelock (a cycle of transitions on unstable global states):

$$a^{(1)}.\mathbf{i}^{(1)}\,[\underline{a^{(1)}}/\underline{a^{(2)}}\rangle \cdots a^{(n)}.\mathbf{i}^{(n)}\,[\underline{a^{(n)}}/\underline{a^{(1)}}\rangle\, a^{(1)}.\mathbf{i}^{(1)}.$$

Figure 3.8 shows a variation of M_{ex} in Figure 2.3. It changes local transition $1\,[M_1, a/x\rangle\, 2$ to $1\,[M_1, a/d\rangle\, 2$. This new NCFSM enters the livelock given external input c in global state $(1,1)$ as

$$(1,1)\,[c/\underline{a}\rangle\, a.(1,2)\,[\underline{a}/\underline{d}\rangle\, d.(2,2)\,[\underline{d}/\underline{b}\rangle\, b.(2,1)\,[\underline{b}/\underline{c}\rangle\, c.(1,1)\,[\underline{c}/\underline{a}\rangle\, a.(1,2).$$

If an NCFSM has a livelock, it will indefinitely delay the consumption of the next input

symbol from the environment. Thus, a livelock is a fatal flaw in the design and is normally the first property to be verified.

If an NCFSM contains livelocks, the MDD encoding $\mathcal{T}_{UTC}$ contains transitions where:

- the "from" global state is the same as the "to" global state.
- the input symbol is identical to the output symbol and belongs to $\mathcal{Z}_{int}$.

The filter for these livelock conditions is easy to build using QFI reduction rules. Then, using the *Intersection* operator on the resulting set from the filter with the set UTC yields all potential livelocks as a $2L$-variable MDD. Next, all "from" global states $\mathcal{S}_{lock}$ can be extracted from the resulting MDD. Finally, we check if any state in $\mathcal{S}_{lock}$ is actually reachable which can be done through an *Intersection* with $\mathcal{S}_{rch}$, the result of reachability analysis. In conclusion, this algorithm not only verifies livelock freeness, when the resulting MDD encodes the empty set, but also finds all reachable livelock-originating states in the design, when the check fails. In the latter case, traces from $\mathbf{s}_{init}$ to each $\mathbf{s} \in \mathcal{S}_{lock}$ can be generated efficiently [63] using a CTL model checker to help designers eliminate the livelock. Trace generation is similar to distinguishing sequence generation, discussed later. Generating a shortest trace, however, is essentially the minimal EG witness generation problem for CTL model checking. Fortunately, our symbolic model checker has an efficient solution [214]. Generating a shortest trace from $\mathbf{s}_{init}$ to the error-inducing state (ignoring the length of the livelock itself), on the other hand, is not difficult, and can be done by reverse reachability analysis used in the next section.

3.3.5 Symbolic strong connectedness verification

Many traditional FSM analysis algorithms require that the observable product machine is strongly connected. This property is satisfied if all reachable stable global states are mutually reachable.

However, we can also define this property in the following way: an NCFSM is strongly connected iff the initial state $\mathbf{s}_{init}$ is reachable from every reachable state $\mathbf{i} \in \mathcal{S}_{st}$. To check this property, we build the MDD for $\mathcal{T}^{-1}$, the inverse of $\mathcal{T}$: if $\mathbf{i}\,[a/x\rangle\,\mathbf{j} \in \mathcal{T}$ then $\mathbf{j}\,[x/a\rangle\,\mathbf{i} \in \mathcal{T}^{-1}$. The MDD encoding $\mathcal{T}^{-1}$ can be built by changing the variable order of $\mathcal{T}$: each w'_k appears before, instead of after, its corresponding w_k by swapping two adjacent variables. This is a standard and efficient MDD operation. Then, we perform a backward state-space search from $\mathbf{s}_{init}$ along $\mathcal{T}^{-1}$, and build the set of reachable states $\mathcal{S}_{st}^{-1}$ using BFS or saturation. Finally, the global state space is strongly connected iff $\mathcal{S}_{st} = \mathcal{S}_{st}^{-1} \cap \mathcal{S}_{rch}$.Note that $\mathcal{T}^{-1}$ may be non-deterministic even if $\mathcal{T}$ is deterministic, though this does not hinder the applicability of symbolic methods.

3.3.6 Symbolic dead transition verification

For a system modeled by an NCFSM, dead transitions mean wasteful designs or useless functions, thus it is important to detect and eliminate them. Formally, transition $i_k\,[M_k, a/b\rangle\,j_k$ is dead if it is never used in the product machine of the NCFSM, i.e., if $\mathcal{T}$ does not contain any transition $\mathbf{i}\,[a/b\rangle\,\mathbf{j}$, $a.\mathbf{i}\,[\underline{a}/b\rangle\,\mathbf{j}$, $\mathbf{i}\,[a/\underline{b}\rangle\,b.\mathbf{j}$, or $a.\mathbf{i}\,[\underline{a}/\underline{b}\rangle\,b.\mathbf{j}$, where i_k is the kth component of $\mathbf{i}$, which then implies that $\mathbf{j} = \mathbf{i}|_{k:j_k}$.

As we have already built $\mathcal{S}_{rch} = \mathcal{S}_{st} \cup \mathcal{S}_{unst}$, dead transitions can be easily detected through MDDs operations. Specifically, for each $i_k\,[M_k, a/b\rangle\,j_k$, we can first check if $\mathcal{S}_{unst}$ contains an unstable state with $w_k = i_k$ and $w_b = a$; if it does, then

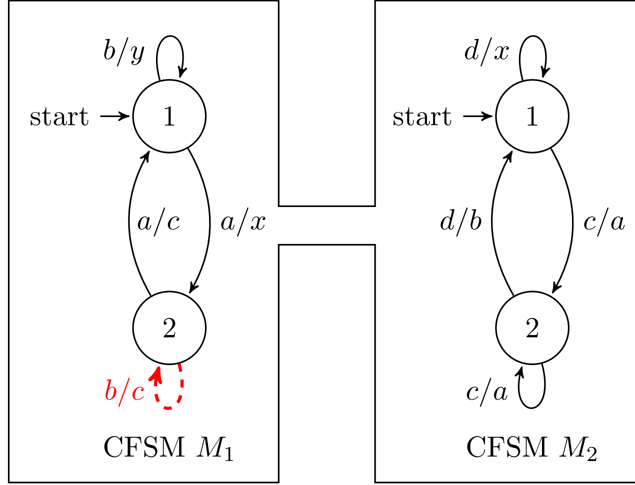

Figure 3.9: A fully masked mutant of M_{ex} in Figure 2.3, $\overline{M_{ex}}^{(1)} = (M_1, M_2)$. The introduced modification is that local transition $2\,{\scriptstyle [M_1,\, b/c\rangle}\, 1$ of M_{ex} is changed to $2\,{\scriptstyle [M_1,\, b/c\rangle}\, 2$ of $\overline{M_{ex}}^{(1)}$.

$i_k\,{\scriptstyle [M_k,\, a/b\rangle}\, j_k$ is not dead. Otherwise, if $a \in \mathcal{X}_{ext}$, we check if $\mathcal{S}_{rch}$ contains a stable state with $w_k = i_k$ (and, obviously, $w_b = \epsilon$); if it does, then $i_k\,{\scriptstyle [M_k,\, a/b\rangle}\, j_k$ is not dead, since i_k is the kth component of a global stable state, and a can be received from the environment in that state. Otherwise, $i_k\,{\scriptstyle [M_k,\, a/b\rangle}\, j_k$ is dead.

3.3.7 Symbolic equivalence verification and its application

Equivalence checking is introduced by mutation analysis for fault-based testing [155]. Given an NCFSM requirement, there are many different ways to implement the design. Among these implementations, there are correct and incorrect ones. In order to assure the conformity between the implementation and the design, fault-based testing is proposed by actively introducing one or more errors to construct mutants from the original requirement [154]. Among these mutants, some of them are equivalent to the original requirement. They share the same behavior as the requirement and are indistinguishable. For example, Figure 3.9 is a mutant of M_{ex} in Figure 2.3 where local

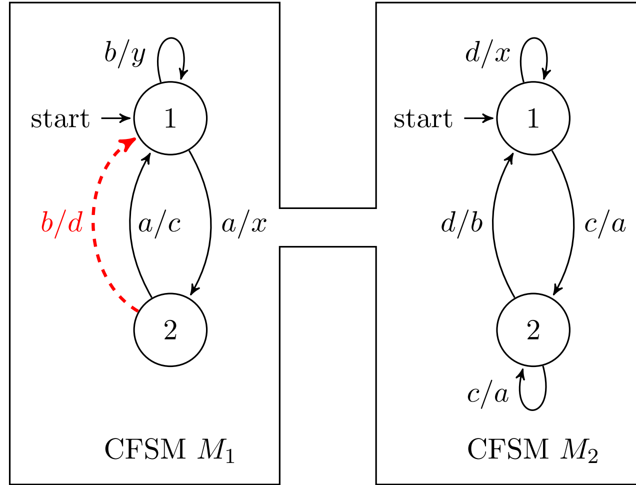

Figure 3.10: A partially masked mutant of M_{ex} in Figure 2.3, $\overline{M_{ex}}^{(2)} = (M_1, M_2)$. The introduced modification is that local transition $2\,{\scriptstyle [M_1,b/c\rangle}\,1$ of M_{ex} is changed to $1\,{\scriptstyle [M_1,b/d\rangle}\,2$ of $\overline{M_{ex}}^{(2)}$.

transition $2\,{\scriptstyle [M_1,b/c\rangle}\,1$ is changed to $2\,{\scriptstyle [M_1,b/c\rangle}\,2$. The error is fully masked, due to local transitions $1\,{\scriptstyle [M_2,c/a\rangle}\,2$, $2\,{\scriptstyle [M_2,c/a\rangle}\,2$, and $2\,{\scriptstyle [M_1,a/c\rangle}\,1$. Instead, consider Figure 3.10, with an introduced error on local transition $2\,{\scriptstyle [M_1,b/d\rangle}\,1$. This error is partially masked if M_2 is at state 1, due to local transition $1\,{\scriptstyle [M_2,d/x\rangle}\,1$. However, the sequence $c/x, b/x \in Tr(\mathbf{s}_{init})$ is able to distinguish this mutant from M_{ex}, also called killing the mutant, since its corresponding resulting sequence from $\overline{M_{ex}}^{(2)}$ is $c/x, b/y$ instead. Thus, if a mutant is not equivalent to the original NCFSM, there must exist an input sequence to distinguish them. This distinguishing sequence can be contributed to a test suite which is able to kill all non-equivalent mutants. The final test suite can then be used to test the real implementation. These fault-based testing methods have achieved great success in web applications and other collaborative systems [173] due to their scalability. The key of these methods is to verify the equivalence between an NCFSM and its mutants.

Structural-based methods are another way to generate test cases for an NCFSM. These methods can be mainly categorized into two classes. One is to is to transform an NCFSM into M_{obs} [150]. Standard test derivation techniques for an FSM, such as the

W-method, Wp-method, and UIO-method, can then be employed. Although these algorithms are well-studied, the transformation limits the applicability of this approach. For instance, even if all component CFSMs are deterministic, minimal, completely specified, and strongly connected, the resulting M_{obs} might not enjoy these properties. However, all above traditional testing generation methods have these strong restrictions on the generated M_{obs}. M_{obs} is deterministic and completely specified only if the NCFSM is livelock-free, otherwise, it is not completely specified. Minimality and strong connectedness are not guaranteed either. Since $\mathcal{Y}_{ext} \subseteq \mathcal{Y}$, the limited number of external output symbols could reduce the ability to distinguish global states of M_{obs}. For example, in Figure 2.4, states $(2, 1)$ and $(2, 2)$ in M_{obs} are state equivalent, thus M_{obs} is not minimal. For strong connectedness, there could exist a subset of stable global states in M_{obs} that only has outgoing global transitions to themselves; the state sets $\{(2, 2)\}$ and $\{(2, 1), (2, 2)\}$ in M_{obs}. Since none of these properties being minimal, strong connected, and completely specified can be guaranteed in M_{obs}, standard structural FSM-based test derivation algorithms cannot be directly applied.

Another approach of generating test cases from an NCFSM is to try to obtain test cases without building M_{obs} [104, 116, 143]. so it avoids an expensive full reachability analysis and instead uses branching coverage [143] or heuristic techniques [104]. Researchers [116] provides a method to check local transitions instead of global transitions and is able to reduces testing efforts under the assumption that the system only has one fault. It extends the notion of unique input/output (UIO) sequences to a constrained identification sequences (CIS) that distinguishes a global state from others in which only one local state is different, under a set of constraints about other local state restrictions. The CISs are generated through BFS and then combined to form sequences that check global states. A dependency digraph is provided to prevent dependency cir-

cularity caused by constraint conflicts. However, if the dependency digraph contains cycles, the algorithm may not find the appropriate order of local CISs to check global states. An order graph is suggested to reduce the required number of resetting the system. However, if the order graph contains cycles, finding the appropriate CIS sets to be sequenced or the minimal sequencing order are exhaustive search problems as well, for which the author proposes greedy or heuristic algorithms.

Both structure-based and fault-based approaches have advantages and limitations. Although a fault-based approach is more applicable and scalable and having fewer constraints on the requirement model, the major challenge of equivalence checking is the ability to handle a large number of mutants and to generate a distinguishing sequence for each non-equivalent mutant [155] efficiently. All distinguishing sequences together will form a test suite. Note that all mutants have a similar structure to the requirement system. Thus, symbolic strategies will have great advantages in terms of both storage and computation.

Before going into more detail of test generation algorithms, we first introduce the mutant operators used to generate a set of first-order mutants $\mathcal{U}$ given specification NCFSM M, as follows:

- Alter the initial state. Create a mutant by changing one of the local states in the initial state $\mathbf{s}_{init}$. This generates $\sum_{1\leq k\leq K}(|\mathcal{S}_k|-1)$ mutants.
- Alter the output of a local transition. Create a mutant by changing local transition $i\,[M_k, a/b\rangle\, j$ to $i\,[M_k, a/b'\rangle\, j$, for $b' \in \mathcal{Y}_k \setminus \{b\}$. This generates $\sum_{1\leq k\leq K}|\delta_k|(|\mathcal{Y}_k|-1)$ mutants, where $|\delta_k|$ is the number of local transitions in M_k, thus $|\delta_k| = |\mathcal{X}_k|\cdot|\mathcal{S}_k|$ if the model is completely specified.
- Alter the destination state of a local transition. Create a mutant by changing

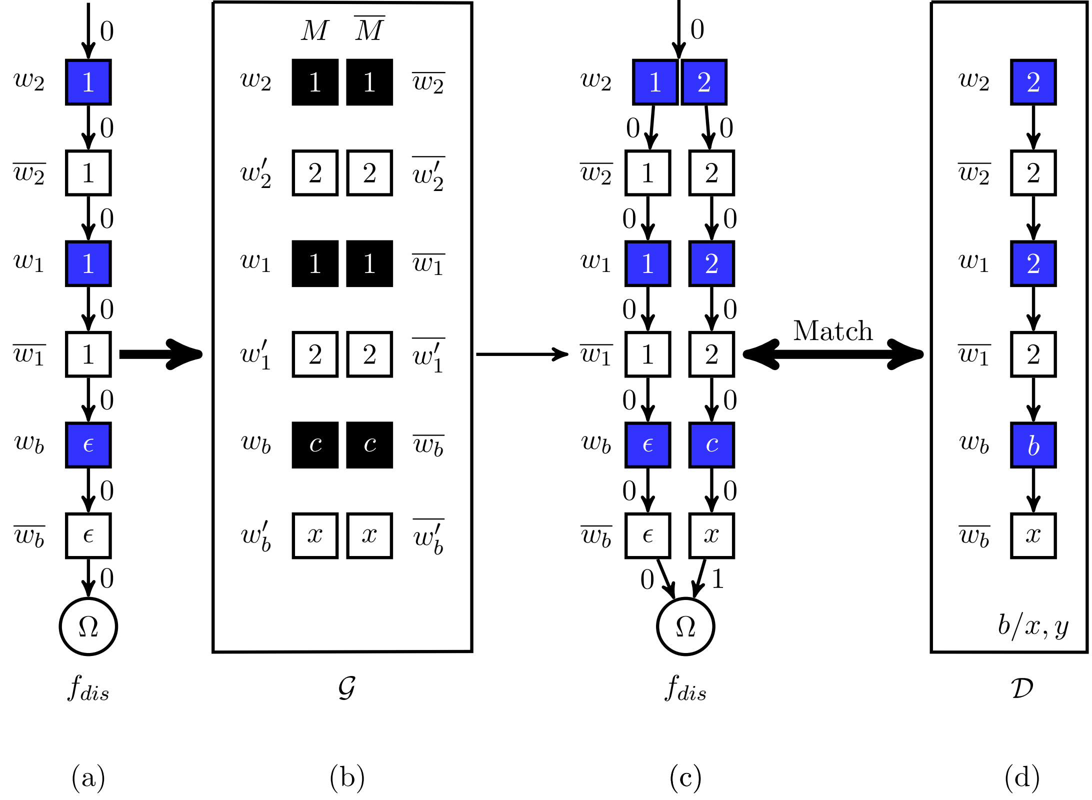

Figure 3.11: Encoding distance function f_{dis}, next-state-pair function $\mathcal{G}$, and distinguishable-state-pair $\mathcal{D}$ of M_{ex} in Figure 2.3 and $\overline{M_{ex}}^{(2)}$ in Figure 3.10.

local transition $i\,[M_k, a/b\rangle\, j$ to $i\,[M_k, a/b\rangle\, j'$, where $j' \in \mathcal{S}_k \setminus \{j\}$. This generates $\sum_{1\leq k\leq K} |\delta_k|(|\mathcal{S}_k| - 1)$ mutants.

Given a mutant $\overline{M}$ of specification M ("$\overline{}$" indicates quantities related to the mutant), our objective is to find a sequence $a_1/b_1,\ldots, a_n/b_n$ as a test case that kills this mutant if it is not equivalent, where each a_i/b_i pair corresponds to an input symbol and the corresponding expected output of the specification M. Let $\alpha = a_1a_2\cdots a_{n-1}$ and $\beta = b_1b_2\cdots b_{n-1}$, then $\lambda_{obs}(s,\alpha) = \beta = \overline{\lambda_{obs}}(s,\alpha)$ and $\lambda_{obs}(\delta_{obs}(s,\alpha), a_n) = b_n \neq \overline{\lambda_{obs}}(\overline{\delta_{obs}}(s,\alpha), a_n)$. The general idea is to simultaneously process both M_{obs} and $\overline{M_{obs}}$ from their initial state with same α, until we reach a pair of states that are distinguishable by introducing a_n.

```
evmdd PairRelProd(evmdd ⟨μ,p⟩, mdd g1, mdd g2)
 1  if g1.v = wb or g2.v = wb then
 2    return ⟨μ, MDD2EV(g1)⟩;                                  • We should have g1 = g2
 3  if CacheLookUp(PairRelProdCode, p, g1, g2, ⟨λ,r⟩) then return⟨λ + μ,r⟩;
 4  evmdd ⟨λ,t⟩;
 5  t ← 0;
 6  λ ← 0;
 7  foreach i, i′ ∈ V_{p.v}, s.t. p[i].val ≠ ∞ ∧ g1[i][i′] ≠ 0 do
 8    foreach j, j′ ∈ V_{p.v} s.t. g2[j][j′] ≠ 0 ∧ p[i][j].val ≠ ∞ do
 9      evmdd ⟨η,u⟩ ← PairRelProd(p[i][j], g1[i][i′], g2[j][j′]);
10      t[i′][j′] ← Minimum(t[i′][j′], ⟨η,u⟩);
11  ⟨λ,t⟩ ← Normalize(t);
12  UniqueTableInsert(t);
13  CacheInsert(PairRelProdCode, p, g1, g2, ⟨λ,t⟩);
14  return ⟨λ + μ,t⟩;
```

```
seq TCGen(evmdd r, mdd G, evmdd f_dis, seq a/x)
15  seq tr ← a/x;
16  while r.val > 0 do
17    foreach G_{b/y} ∈ G do
18      if t ∈ f_dis^{-1}(f_dis(r) − 1) ∧ r = G_{b/y}(t) then
19        r ← t;                                                • predecessor
20        tr ← b/y · tr;
21        break;
22  return tr
```

Figure 3.12: The algorithms for the *PairRelProd* operator and the test generation.

If the state pair set is $\mathcal{P} = \mathcal{S}_{st} \times \overline{\mathcal{S}}_{st}$, define the next-state-pair function $\mathcal{G} = \{\mathcal{G}_{a/b} : a \in \mathcal{X}, b \in \mathcal{Y}\}$ and the distinguishable-state-pairs $\mathcal{D} = \{\mathcal{D}_{a/b} : a \in \mathcal{X}, b \in \mathcal{Y}\}$:

$$\mathcal{G}_{a/b} = \{\big((a.\mathbf{i}, b.\mathbf{j}), (a.\bar{\mathbf{i}}, b.\bar{\mathbf{j}})\big) : (a.\mathbf{i}, b.\mathbf{j}) \in \mathcal{T}_{obs} \wedge (a.\bar{\mathbf{i}}, b.\bar{\mathbf{j}}) \in \overline{\mathcal{T}_{obs}}\},$$

$$\mathcal{D}_{a/b} = \{(a.\mathbf{i}, b.\bar{\mathbf{i}}) : \exists (a.\mathbf{i}, b.\mathbf{j}) \in \mathcal{T}_{obs} \wedge \exists (a.\bar{\mathbf{i}}, \bar{b}.\bar{\mathbf{j}}) \in \overline{\mathcal{T}_{obs}} \wedge b \neq \bar{b}\}.$$

These sets can be built through symbolic operations on the MDDs for $\mathcal{S}_{st}$, $\mathcal{T}_{obs}$, and $\overline{\mathcal{T}_{obs}}$. Consider M_{ex} in Figure 2.3 and $\overline{M_{ex}}^{(2)}$ in Figure 3.10. Figure 3.11(b) shows an element in $\mathcal{G}$, encoding $(1,1) \llbracket M_{ex}, c/x \rangle\!\rangle (2,2)$ and $(1,1) \llbracket \overline{M_{ex}}, c/x \rangle\!\rangle (2,2)$ as a pair of transitions. Figure 3.11(d) shows an element in $\mathcal{D}$, the pair of states $((b,2,2),(b,2,2))$, which are distinguishable because $(2,2) \llbracket M_{ex}, b/x \rangle\!\rangle (2,2)$ while $(2,2) \llbracket \overline{M_{ex}}, b/y \rangle\!\rangle (1,1)$.

Our test derivation algorithm takes in input the set $\mathcal{U}$ of mutants, the stable

next-state function $\mathcal{T}_{obs}$, and $v(\mathbf{p})_{init} = (\mathbf{i}_{init}, \overline{\mathbf{i}_{init}})$. For each mutant $\overline{M}$, we first run the current test suite to check whether an existing test kills $\overline{M}$. If not, we build $\overline{\mathcal{T}_{obs}}$ for $\overline{M}$ and encode the next-state-pair function $\mathcal{G}$ and the distinguishable-state-pairs $\mathcal{D}$. The algorithm is analogous to state-space exploration, except that we explore pairs of states (one from M and one from $\overline{M}$), and keep track of the distance of each such pair from $v(\mathbf{p})_{init}$ by using a $2(K+1)$-variable EV$^+$MDD instead of $(K+1)$-variable MDD. This EV$^+$MDD encodes the distance function:

$$f_{dis} : \mathcal{P} \to \mathbb{N} \cup \{\infty\} \;\; \text{s.t.} \;\; f_{dis}(v(\mathbf{p})) = \min\{d : v(\mathbf{p}) \in \mathcal{G}^d(v(\mathbf{p})_{init})\},$$

so that $f_{dis}(v(\mathbf{p})) = \infty$ iff $v(\mathbf{p})$ has not yet been reached in the exploration, starting with the initialization $f_{dis}(v(\mathbf{p})_{init}) = 0$ and $f_{dis}(v(\mathbf{p})) = \infty$ for $v(\mathbf{p}) \neq v(\mathbf{p})_{init}$. We also define its reverse function: $f_{dis}^{-1}(d) = \{v(\mathbf{p}) : f_{dis}(v(\mathbf{p})) = d\}$, where $d \in \mathbb{N}$.

The algorithm uses a BFS algorithm to generate the distance function for reachable state pairs until the search reaches a distinguishing state pair $v(\mathbf{p})_{err}$ in $\mathcal{D}$. If no such pair is reachable, $\overline{M}$ is equivalent to M, and the algorithm builds a fixed-point $\mathcal{P}_{rch}$ containing all the pairs of states that can be reached from $v(\mathbf{p})_{init}$ by providing the same input sequence to both M and $\overline{M}$.

Procedure *PairRP*, shown in Figure 3.12, applies $\mathcal{G}$ to the distance function f_{dis} and returns a set of state pairs with an incremented value of f_{dis}. To help with test-case generation, we observe that $v(\mathbf{p})$ consists of a pair of stable global states, thus $i_\beta = \overline{i_\beta} = \epsilon$ by definition. For efficiency, we then use i_β and $\overline{i_\beta}$ to store instead an input/output symbol pair leading to this state pair. For example, Figure 3.11c encodes $f_{dis}((c, 2, 2), (x, 2, 2)) = 1$ instead of $f_{dis}((\epsilon, 2, 2), (\epsilon, 2, 2)) = 1$. We can achieve this through a minor change when dealing with terminal cases in Procedure *PairRP* Line 2. Procedure *MDD2EV* transforms an MDD into an EV$^+$MDD with value 0 for

elements encoded by the MDD, ∞ to the rest.

We again use M_{ex} and $\overline{M_{ex}}^{(2)}$ to illustrate. Figure 3.11(a) encodes $v(\mathbf{p})_{int}$ with a distance of 0, where nodes with the same color correspond to one of the stable states in the pair. Applying Figure 3.11(b) to Figure 3.11(a) results in the $f_{dis} \leq 1$ of Figure 3.11(c), containing $v(\mathbf{p})_{err} = \big((2,2),(2,2)\big)$, which can be found in $\mathcal{D}$, explained above. $v(\mathbf{p})_{err}$ is used to generate a sequence as a new test case which is added to the test suite $\mathcal{C}$. Procedure *TCGen* in Figure 3.12 uses the distance function to generate a sequence leading M and $\overline{M}$ from $v(\mathbf{p})_{init}$ to $v(\mathbf{p})_{err}$. Starting from $v(\mathbf{p})$ at distance n, there must exist a predecessor $\mathbf{q}$, i.e., satisfying $v(\mathbf{p}) = \mathcal{G}(\mathbf{q})$, at distance $n-1$. Thus, we keep reducing the distance value until reaching $v(\mathbf{p})_{init}$, at distance 0. In the above example, sequence $(c/x, b/x)$ is built and added to the test suite. Finally, Procedure *Minimize* eliminates test cases subsumed by other test cases, to form a minimal test suite.

3.3.8 Experimental results

We implement the proposed framework using our MDD library [204], and report experimental results on an Intel Xeon 2.53GHz workstation with 36GB RAM running Linux. The main metrics of our comparison are runtime (in seconds) and peak memory (in MB). For BFS and saturation, we compare the cumulative time to compute the transition transitive closure on all mutants (UTC_{bfs} and UTC_{sat}), and the total runtime and peak memory (Tot_{bfs} and Tot_{sat}). For each model, we list the number of components (K), of mutants (Tot), of non-equivalent mutants (NE), of test cases (Num), as well as the average length of the tests in the suite (Avg). The total time includes runtime for preprocessing, livelock checking, and test suite generation.

Model		Mutants		Test Suite		UTC_{bfs}	UTC_{sat}	Tot_{bfs}		Tot_{sat}	
M	K	Tot	NE	Num	Avg	time	time	mem	time	mem	time
Completely specified models											
M_{ex}	2	26	20	6	2.33	0.002	0.003	2.23	0.003	2.19	0.004
M_{se}	3	96	96	27	3.19	0.053	0.021	3.63	0.427	3.42	0.235
M_{hs}	3	81	81	16	2.81	0.044	0.018	3.60	0.301	3.30	0.158
Incompletely specified models											
M'_{ex}	2	17	8	3	1.67	<0.001	<0.001	2.12	<0.001	2.10	<0.001
M'_{se}	3	90	90	23	3.30	0.039	0.014	3.28	0.245	3.10	0.192
M'_{hs}	3	77	77	13	3.08	0.041	0.012	3.24	0.252	3.00	0.126
Control systems											
M_{hcs}	4	179	157	12	11.17	0.17	0.03	7.01	1.00	6.54	0.48
M_{tr}	4	177	150	12	2.50	0.17	0.09	6.72	1.00	5.33	0.20
M_{tr3}	5	1024	849	43	3.14	4.69	3.35	69.75	9.25	50.72	7.74
Communication protocols											
ABP	2	96	81	12	3.83	0.005	0.004	2.90	0.49	2.90	0.32
BGP	2	4898	1613	79	5.16	344.26	24.10	96.48	1230.0	82.41	438.6
EGP	3	69066	27883	3501	9.24	38928	14631	7085.84	63384	5183.89	25904

Table 3.1: Test case generation results (time in seconds and memory in MB).

Table 3.1 presents results for four sets of models. The first set, shown under "completely specified models", consists of M_{ex} in Figure 2.3, our running example, M_{se} [115], and M_{hs} [116]. All components in these models are completely specified, minimized, strongly connected, and deterministic. We obtain the second set by modifying these three models, eliminating some selfloops to derive three corresponding incompletely specified models: M'_{ex}, M'_{se}, and M'_{hs}. The third set consists of several control systems: a heating controller system [90] and a train gate controller [32] with two trains,

M_{tr}, or three trains, M_{tr3}. The fourth set includes three communication protocols: the alternating bit protocol (ABP) [194], the border gateway protocol (BGP) [180], and the exterior gateway protocol (EGP) [159]. ABP is used to guarantee correct data delivery between a sender and a receiver connected by an error-prone channel. BGP and EGP are two important TCP/IP exterior routing protocols. BGP is currently used on fully decentralized inter-autonomous systems. Both EGP and BGP contain some signal events undefined in many local states, so they are incompletely specified. We only consider mandatory events.

We first discuss the results on the first two sets of models. As these are relatively small and have only up to three components, the runtime in the result is the average time over 50 runs. For M_{ex} and $M_{ex}^{'}$, the saturation algorithm is less efficient. As stated before, the saturation algorithm works better for large models, because of its complex recursive structure. The overhead is not paid off if not enough locality can be exploited. However, the memory consumption is still less. For the other models, the saturation algorithm works better in both time and memory, although only minor improvements are observable for M_{se}, M_{hs}, and their corresponding incompletely specified models.

Consider the set of control system models. The runtime and memory consumption are all reduced with only minor improvements. Turning to communication protocols, *BGP* models a connection between two routers and contains two components, both of which, according to the specification [180], maintain a separate CFSM for each configured peer (for this model, we only consider mandatory events). Thus, including the component for the environment, we encode the model with 5 variables. There are 1,153 reachable global states and 40,939 global transitions. Saturation is clearly superior, 14 times faster than BFS when computing the *UTC* for all mutants.

Similar trends can be observed for *EGP* with three peers: saturation saves more than 10 hours and almost 2GB over BFS. There are about 1.2×10^5 global states, 6.7×10^5 local transitions, and 1.5×10^6 global transitions for the considered EGP model.

The benefit of symbolic encodings can be clearly seen in our results, as the memory consumption remains stable even if the number of generated mutants increases by an order of magnitude when growing the number of components. Also, we can observe that the number of generated test cases and the average length of the test suite are stable even if the number of mutants increases dramatically. This is important for complex models in practice, as it reduces testing efforts.

A further advantage of our symbolic approach is that no constraints are required of investigating models. Most state-of-the-art test generation methods assume requirement models to be completely specified and strongly connected, which can hardly be met in many real models. Moreover, our framework can be extended to non-deterministic NCFSMs by returning instead of distinguishing sequences, test cases as pairs, each consisting of an input string and a set of all correct output strings.

3.4 Petri nets

In this section, we will introduce a self-modifying Petri net [201] (PN) with priorities and inhibitor arcs, described by a seven-tuple $(\mathcal{P}, \mathcal{T}, \pi, \mathbf{D}^-, \mathbf{D}^+, \mathbf{D}^\circ, \mathbf{s}^{init})$, where:

- $\mathcal{P}$ is a finite set of places, drawn as circles, and $\mathcal{T}$ is a finite set of transitions, drawn as rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$.
- $\pi : \mathcal{T} \to \mathbb{N}$ assigns a priority to each transition.
- $\mathbf{D}^- : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, $\mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, and $\mathbf{D}^\circ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N} \cup \{\infty\}$ are the marking-dependent cardinalities of the input, output, and inhibitor arcs.

- $\mathbf{s}^{init} \in \mathbb{N}^{\mathcal{P}}$ is the initial marking, the number of tokens initially in each place.

Transition t has concession in marking $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$ if, for each $p \in \mathcal{P}$, the input arc cardinality is satisfied, i.e., $\mathbf{m}_p \geq \mathbf{D}^{-}(p, t, \mathbf{m})$, and the inhibitor arc cardinality is not, i.e., $\mathbf{m}_p < \mathbf{D}^{\circ}(p, t, \mathbf{m})$. If t has concession in $\mathbf{m}$ and no other transition t' with priority $\pi(t') > \pi(t)$ has concession, then t is enabled in $\mathbf{m}$ and can fire and lead to marking $\mathbf{m}'$, where $\mathbf{m}'_p = \mathbf{m}_p - \mathbf{D}^{-}(p, t, \mathbf{m}) + \mathbf{D}^{+}(p, t, \mathbf{m})$, for all places p (arc cardinalities are evaluated in the current marking $\mathbf{m}$ to determine the enabling of t and the new marking $\mathbf{m}'$). In our figures, $tk(p)$ indicates the number of tokens in p for the current marking, a thick input arc from p to t signifies a cardinality $tk(p)$, i.e., a reset arc, and we omit arc cardinalities 1, input or output arcs with cardinality 0, and inhibitor arcs with cardinality ∞.

The PN defines a discrete-state model $(\mathcal{S}_{pot}, \mathcal{S}_{init}, \mathcal{A}, \{\mathcal{N}_t : t \in \mathcal{A}\})$. The potential state space is $\mathcal{S}_{pot} = \mathbb{N}^{\mathcal{P}}$ (in practice we assume that the reachable set of markings is finite or, equivalently, that there is a finite bound on the number of tokens in each place, but we do not require to know this bound a priori). $\mathcal{S}_{init} \subseteq \mathcal{S}_{pot}$ is the set of initial states, $\{\mathbf{s}^{init}\}$ in our case (assuming an arbitrary finite initial set of markings is not a problem). The set of (asynchronous) model events is $\mathcal{A} = \mathcal{T}$. The next-state function for transition t is $\mathcal{N}_t$, such that $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{m}'\}$, where $\mathbf{m}'$ is as defined above if transition t is enabled in marking $\mathbf{m}$, and $\mathcal{N}_t(\mathbf{m}) = \emptyset$ otherwise. Thus, the next-state function for a particular PN transition is deterministic, although the overall behavior remains nondeterministic due to the choice of which transition should fire when multiple transitions are enabled.

PNs are a great candidate to express ECA rules. The nondeterministic interleaving execution semantics of PNs can naturally model unforeseen interactions between

ECA rule executions and environmental changes. Also, we can conveniently map priorities of PNs to rule priorities. Thus, an intermediate step in verifying ECA rules, discussed in the next section, is to translate them into a self-modifying PN with priorities and inhibitor arcs.

3.5 Symbolic verification of a set of ECA rules

Given a set of ECA rules, termination and confluence are two fundamental properties that assure the correctness of the design. Termination guarantees that the system does not remain "busy" internally forever without responding to external events so that the designed system fulfills the constantly reactive requirement. Confluence, on the other hand, ensures that any possible interleaving of a set of triggered rules yields the same final result so that the designed system behaves consistently regardless of the nondeterministic nature in both the hybrid system and the environment. While termination has been studied extensively and many algorithms have been proposed to verify it, in real applications, confluence is particularly challenging due to a potentially large number of rule interleavings [4].

Researchers began studying these properties for active databases in the early 90's [4, 29, 145, 162], by transforming ECA rules into some form of graphs and applying various static analysis techniques on it to verify properties. These approaches based on a static methodology worked well to detect redundancy, inconsistency, incompleteness, and circularity. However, since static approaches may not explore the whole state space, they could easily miss some errors. Also, they could find scenarios that did not actually result in errors due to the fact that found error states that may not be reachable. Moreover, they had poor support to provide concrete counterexamples and analyze ECA

rules with priorities. Researchers [4] looked for cycles in the rule-triggering graph to disprove termination, but the cycle-triggering conditions may be unsatisfiable. Later, this work was improved [29] with an activation graph describing when rules are activated; while its analysis detects termination where previous works failed, it may still report false positives when rules have priorities. Another work [30] proposed an algebraic approach emphasizing the condition portion of rules, but did not consider priorities. Other researchers [145, 162] chose to translate ECA rules into a Petri Net (PN), whose non-deterministic interleaving execution semantics naturally models unforeseen interactions between rule executions. However, as the analysis of the set of ECA rules was through structural PN techniques based on the incidence matrix of the net, false positives were again possible.

To overcome these limitations, dynamic analysis approaches using model checking tools such as SMV [178] and SPIN [56] have been proposed to verify termination. While closer to our work, these approaches require manually transforming ECA rules into an input script, assume a priori bounds for all variables, provide no support for priorities, and require the initial system state to be known; our approach does not have these limitations. Researchers [75] analyzes both termination and confluence by transforming ECA rules into Datalog rules through a "transformation diagram"; this supports rule priority and execution semantics, but requires the graph to be commutative and restricts event composition. However, most of these works show limited results, and none of them properly addresses confluence; we present detailed experimental results for both termination and confluence. Statecharts defined in unified modeling language (UML) [203] provide visual diagrams to describe the dynamic behavior of reactive systems and can verify these properties, but event dispatching and execution semantics are not as flexible as for PNs [161].

Our approach transforms a set of ECA rules into a PN, then dynamically verifies termination and confluence and, if errors are found, provides concrete counterexamples to help with debugging. It uses our tool SMART, which supports PNs with priorities to model rule priorities. Moreover, a single PN can naturally describe both the ECA rules as well as their nondeterministic concurrent environment and, while our MDD-based symbolic model-checking algorithms [212] require a finite state space, they do not require knowing the variable bounds a priori(i.e., the maximum number of tokens each place may contain). Finally, our approach is not restricted to termination and confluence, but can be easily extended to verify a broader set of properties.

3.5.1 Transforming a set of ECA rules into a PN

We now explain the procedure to transform a set of ECA rules into a PN. First, we put each ECA rule into a regular form where both events and condition are disjunctions of conjunctions of events and relational expressions, respectively. All rules of the smart home example in Figure 2.6 are in this form. While this transformation may in principle cause the expressions for events and conditions to grow exponentially large, each ECA rule usually contains a small number of events and conditions, hence this is not a problem in practice. Based on the immediate event-condition checking assumption, a rule is triggered iff "$trigger \equiv events \wedge condition$" holds.

Next, we map variables and events into places, and use PN transitions to model event testing, condition evaluation, and action execution. Any change of variable values is achieved through input and output arcs with appropriate cardinalities. Additional control places and transitions allow the PN behavior to be organized into "phases", as shown next. ECA rules r_1 through r_{10} of Figure 2.6 are transformed into the PN of

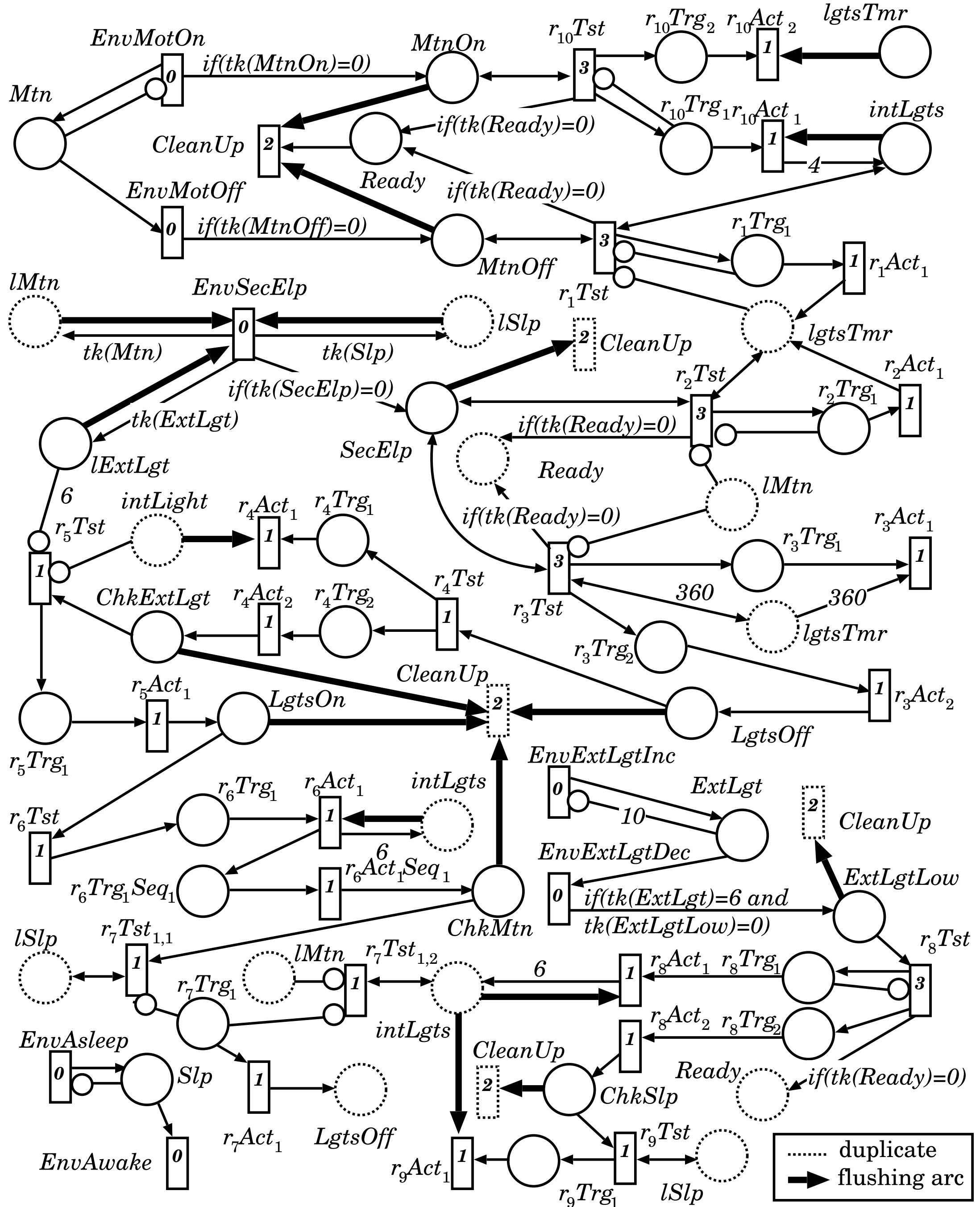

Figure 3.13: The PN for ECA rules in Figure 2.6.

Figure 3.13 (dotted transitions and places are duplicated and arcs labeled "*if*(*cond*)" are present only if *cond* holds).

3.5.1.1 Occurring phase

This phase models the occurrence of external events, due to environment changes over which the system has no control. The PN firing semantics perfectly matches the nondeterministic asynchronous nature of these changes. For example, in Figure 3.13, transitions *EnvMotOn* and *EnvMotOff* can add or remove the token in place *Mtn*, to nondeterministically model the presence or absence of people in the room (the inhibitor arc from place *Mtn* back to transition *EnvMotOn* ensures that at most one token resides in *Mtn*). Firing these environmental transitions might nondeterministically enable the corresponding external events. Here, firing *EnvMotOn* generates the external event *MtnOn* by placing a token in the place of the same name, while firing *EnvMotOff* generates event *MtnOff*, consistent with the change in *Mtn*. To ensure that environmental transitions only fire if the system is in a stable state (when no rule is being processed) we assign the lowest priority, namely 0, to these transitions. As the system does not directly affect environmental variables, rule execution does not modify them. However, we can take snapshots of these variables by copying the current number of tokens into their corresponding local variables using marking-dependent arcs. For example, transition *EnvSecElp* has an output arc to generate event *SecElp*, and arcs connected to local variables to perform the snapshots, e.g., all tokens in *lMtn* are removed by a reset arc (an input arc that removes all tokens from its place), while the output arc with cardinality $tk(Mtn)$ copies the value of *Mtn* into *lMtn*.

3.5.1.2 Triggering phase

This phase starts when $trigger \equiv events \wedge condition$ holds for at least one external ECA rule. If, for rule r_k, events and condition consist of n_d and n_c disjuncts,

respectively, we define $n_d \cdot n_c$ test transitions $r_k Tst_{i,j}$ with priority $P + 2$, where i and j are the index of a conjunct in events and one in condition, respectively, while $P \geq 1$ is the highest priority used for internal rules (in our example, all internal rules have default priority $P = 1$). Then, to trigger rule r_k, only one of these transitions, e.g., $r_7 Tst_{1,1}$ or $r_7 Tst_{1,2}$, needs to be fired (we omit i and j if $n_d = n_c = 1$). Firing a test transition means that the corresponding events and conditions are satisfied and results in placing a token in each of the triggered places $r_k Trg_1, \ldots, r_k Trg_N$, to indicate that Rule r_k is triggered, where N is the number of outermost parallel actions (recall that **par** and **seq** model parallel and sequential actions). Thus, $N = 1$ if r_k contains only one action, or an outermost sequential series of actions. Inhibitor arcs from $r_k Trg_1$ to test transitions $r_k Tst_{i,j}$ ensure that, even if multiple conjuncts are satisfied, only one test transition fires. The firing of test transitions does not "consume" external events, thus we use double-headed arrows between them. This allows one batch to trigger multiple rules, conceptually "at the same time". After all enabled test transitions for external rules have fired, place *Ready* contains one token, indicating that the current batch of external events can be cleared: transition *CleanUp*, with priority $P+1$, fires and removes all tokens from external and internal event places using reset arcs, since all the rules that can be triggered have been marked. This ends the triggering phase and closes the current batch of events.

3.5.1.3 Performing phase

This phase executes all actions of external rules marked in the previous phase. It may further result in triggering and executing internal rules. Transitions in this phase correspond to the actions of rules with priority in $[1, P]$, the same as that of the

TransformECAintoPN $(\mathbf{R}_{ext}, \mathbf{R}_{int}, \mathbf{V}_{env}, \mathbf{V}_{loc}, \mathbf{E}_{ext}, \mathbf{E}_{int})$

1 normalize $\mathbf{R}_{ext}$ and $\mathbf{R}_{int}$ into regular form and set P to the highest rule priority
2 create a place $Ready$ • *to control "phases" of the net*
3 create transition $CleanUp$ with priority $P + 1$ and $Ready \multimap[1]\rightarrow CleanUp$
4 foreach event $e \in \mathbf{E}_{ext} \cup \mathbf{E}_{int}$ do
5 create place p_e and $p_e -[tk(p_e)]\rightarrow CleanUp$
6 create place p_v, for each variable $v \in \mathbf{V}_{loc}$
7 foreach variable $v \in \mathbf{V}_{env}$ with range $[v_{min}, v_{max}]$ do
8 create place p_v and transitions t_{vInc} and t_{vDec} with priority 0
9 create $t_{vDec} -[\text{if}(tk(p_v) > v_{min})1 \text{ else } 0]\rightarrow p_v$
10 create $t_{vInc} -[1]\rightarrow p_v$ and $p_v -[v_{max}]\!\!-\!\!\circ\, t_{vInc}$
11 foreach event $e \in \mathbf{E}_{ext}$ **activated when** $v\ op\ val$, for $op \in \{\geq \mid =\}$ do
12 create $t_{vInc} -[\text{if}(tk(p_v)\ op\ val)1 \text{ else } 0]\rightarrow p_e$
13 if e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ then
14 create $p_{v'} -[\text{if}(tk(p_v)\ op\ val)tk(p_{v'}) \text{ else } 0]\rightarrow t_{vInc}$
15 create $t_{vInc} -[\text{if}(tk(p_v)\ op\ val)tk(p_v) \text{ else } 0]\rightarrow p_{v'}$
16 foreach event $e \in \mathbf{E}_{ext}$ **activated when** $v\ op\ val$, for $op \in \{\leq \mid =\}$ do
17 create $t_{vDec} -[\text{if}(tk(p_v)\ op\ val)1 \text{ else } 0]\rightarrow p_e$
18 if e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ then
19 create $p_{v'} -[\text{if}(tk(p_v)\ op\ val)tk(p_{v'}) \text{ else } 0]\rightarrow t_{vDec}$
20 create $t_{vDec} -[\text{if}(tk(p_v)\ op\ val)tk(p_v) \text{ else } 0]\rightarrow p_{v'}$
21 foreach event $e \in \mathbf{E}_{ext}$ without an **activated when** portion do
22 create t_e and $t_e -[\text{if}(tk(p_e) = 0)1 \text{ else } 0]\rightarrow p_e$
23 if e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ then
24 create $p_{v'} -[tk(p_{v'})]\rightarrow t_e$ and $t_e -[tk(p_v)]\rightarrow p_{v'}$
25 foreach rule $r_k \in \mathbf{R}_{ext} \cup \mathbf{R}_{int}$ with n_d event disjuncts, n_c condition disjuncts, actions A, and priority $p \in [1, P]$ do
26 create trans. $r_k Tst_{i,j}$,$i \in [1, n_d], j \in [1, n_c]$,w/priority $P + 2$ if $r_k \in \mathbf{R}_{ext}$, else p
27 foreach event e in disjunct i do
28 create $p_e -[1]\rightarrow r_k Tst_{i,j}$
29 create $r_k Tst_{i,j} -[1]\rightarrow p_e$, if $e \in \mathbf{E}_{ext}$
30 foreach conjunct $v \leq val$ or $v = val$ in disjunct j do
31 create $p_v -[val + 1]\!\!-\!\!\circ\, r_k Tst_{i,j}$
32 foreach conjunct $v \geq val$ or $v = val$ in disjunct j do
33 create $p_v -[val]\rightarrow r_k Tst_{i,j}$ and $r_k Tst_{i,j} -[val]\rightarrow p_v$
34 if actions A is "$(A_1$ *par* $A_2)$" then $n_a = 2$;
35 else $n_a = 1$, $A_1 = A$;
36 foreach $l \in [1, n_a]$ do
37 create places $r_k Trg_l$ and transitions $r_k Act_l$ with priority p
38 create $r_k Trg_l -[1]\rightarrow r_k Act_l$ and $r_k Tst_{i,j} -[1]\rightarrow r_k Trg_l$
39 $SeqSubGraph(A_l, \text{"}r_k Acl_l\text{"}, l, p)$
40 foreach $r_k \in \mathbf{R}_{ext}, i \in [1, n_d], j \in [1, n_c]$ do
41 create $r_k Tst_{i,j} -[\text{if}(tk(Ready) = 0)1 \text{ else } 0]\rightarrow Ready$ and $r_k Trg_1 -[1]\!\!-\!\!\circ\, r_k Tst_{i,j}$

Figure 3.14: Transforming ECA rules into a PN: $a -[k]\rightarrow b$ means "an arc from a to b with cardinality k; $a -[k]\!\!-\!\!\circ\, b$ means "an inhibitor arc from a to b with cardinality k.

```
ParSubGraph(Pars, Pre, p)                • Pars: parallel actions, Pre: prefix
 1  foreach l ∈ {1,2} do          • according to the syntax Pars_2 has two components
 2    create place PreAct_l Trg_l Seq_l and transition PreAct_l w/priority p
 3    create Pre –[1]→ PreAct_l and PreAct_l –[1]→ PreAct_l Trg_l Seq_l
 4    SeqSubGraph(Pars_l, "PreAct_l Trg_l Seq_l", l, p);
```

Figure 3.15: Processing **par**.

```
SeqSubGraph(Seqs, Pre, i, p)             • Seqs: sequential actions, Pre: prefix
 1  if Seqs sets variable v to val then
 2    create p_v –[tk(p_v)]→ Pre and Pre –[val]→ p_v
 3  else if Seqs increases variable v by val then
 4    create Pre –[val]→ p_v
 5  else if Seqs decreases variable v by val then
 6    create p_v –[val]→ Pre
 7  else if Seqs activates an internal event e then
 8    create Pre –[1]→ p_e
 9  else if the outermost operator of Seqs is par then
10    ParSubGraph(Seqs, "Pre", p)                    • Recursion on parallel part
11  else if the outermost operator of Seqs is seq then
12    SeqSubGraph(Seqs_1, "Pre", 1, p)              • Seqs_1 is the first part of Seq
13    create place PreTrg_i Seq_1 and transition PreAct_i Seq_1
14    create Pre –[1]→ PreTrg_i Seq_1 and PreTrg_i Seq_1 –[1]→ PreAct_i Seq_1
15    SeqSubGraph(Seqs_2, "PreTrg_i Seq_1", 2, p)   • Seqs_2 is the second part of Seq
```

Figure 3.16: Processing **seq**.

corresponding rule. An action activates an internal event by adding a token to its place. This token is consumed as soon as a test transition of any internal rule related to this event fires. This is different from the way external rules "use" external events. Internal events not consumed in this phase are cleared when transition *CleanUp* fires in the next batch. When all enabled transitions of the performing phase have fired, the system is in a stable state where environmental changes (transitions with priority 0) can again happen and the next batch starts.

3.5.1.4 ECA rules to PN translation algorithms

The algorithm in Figure 3.14 takes external and internal ECA rules $\mathbf{R}_{ext}$, $\mathbf{R}_{int}$, with priorities in $[1, P]$, environmental and local variables $\mathbf{V}_{env}$, $\mathbf{V}_{loc}$, and external

and internal events $\mathbf{E}_{ext}$, $\mathbf{E}_{int}$, and generates a PN. After normalizing the rules and setting P to the highest priority among the rule priorities in $\mathbf{R}_{int}$, it maps environmental variables $\mathbf{V}_{env}$, local variables $\mathbf{V}_{loc}$, external events $\mathbf{E}_{ext}$, and internal events $\mathbf{E}_{int}$, into the corresponding places (Lines 5, 6, and 8). Then, it creates phase control place *Ready*, transition *CleanUp*, and reset arcs for *CleanUp* (Lines 4-5). We use arcs with marking-dependent cardinalities to model expressions. For example, together with inhibitor arcs, these arcs ensure that each variable $v \in \mathbf{V}_{env}$ remains in its range $[v_{min}, v_{max}]$ (Lines 8-10). These arcs also model the **activated when** portion of external events (Line 17), rule conditions (Line 33), and assignments of environmental variables to local variables (Lines 19-20 and lines 14-15). The algorithm also models external events and environmental changes (Lines 11-24); it connects environmental transitions such as t_{vInc} and t_{vDec} to their corresponding external event places, if any, with an arc whose cardinality evaluates to 1 if the corresponding condition becomes true upon the firing of the transition and the event place does not contain a token already, 0 otherwise (e.g., the arcs from *EnvExtLigDec* to *ExtLgtLow*).

Next, rules are considered (Lines 25-41). A rule with n_d event disjuncts and n_c condition disjuncts generates $n_d \cdot n_c$ testing transitions. To model the parallel-sequential action graph of a rule, we use mutually recursive procedures, one for parallel actions in Figure 3.15 and the other for sequential actions in Figure 3.16. Procedure *SeqSubGraph* first tests all atomic actions, such as "set", "increase", "decrease", and "activate". Then, it recursively calls *ParSubGraph* at Line 10 if it encounters parallel actions. Otherwise, it calls itself to unwind another layer of sequential actions at Line 12 and Line 15 for the two portions of the sequence. Procedure *ParSubGraph* creates control places and transitions for the two branches of a parallel action and calls *SeqSubGraph* at Line 4.

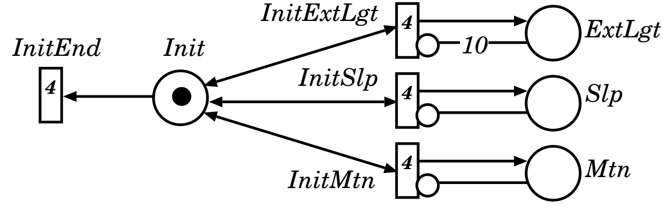

Figure 3.17: The initialization phase for the smart home example.

3.5.2 Verifying properties

The first step towards verifying correctness properties is to define $\mathcal{S}_{init}$, the set of initial states, corresponding to all the possible initial combinations of system variables (e.g., *ExtLgt* can initially have any value in $[0, 10]$). One could consider all these possible values by enumerating all legal stable states corresponding to possible initial combinations of the environmental variables, then start the analysis from each of these states, one at a time. However, in addition to requiring the user to explicitly provide the set of initial states, this approach may require enormous runtime, also because many computations are repeated in different runs. Our approach instead computes the initial states symbolically, thanks to the nondeterministic semantics of PNs, so that the analysis is performed once starting from a single, but very large, set $\mathcal{S}_{init}$.

To this end, we add an initialization phase that puts a nondeterministically chosen legal number of tokens in each place corresponding to an environmental variable. This phase is described by a subnet consisting of a transition *InitEnd* with priority $P+3$, a place *Init* with one initial token, and an initializing transition with priority $P+3$ for every environmental variable, to initialize the number of tokens in the corresponding place. Figure 3.17 shows this subnet for our running example. We initialize the PN by assigning the minimum number of tokens to every environmental variable place and

leaving all other places empty, then we let the initializing transitions nondeterministically add a token at a time, possibly up to the maximum legal number of tokens in each corresponding place. When *InitEnd* fires, it disables the initializing transitions, freezes the nondeterministic choices, and starts the system's normal execution.

This builds the set of initial states, ensuring that the PN will explore all possible initial states, and avoids the overhead of manually starting the PN from one legal initial marking at a time. Even though the overall state space might be larger (it equals the union of all the state spaces that would be built starting from each individual marking), this is normally not the case. Having to perform just one state space generation is obviously enormously better.

After the initialization step, we proceed with verifying termination and confluence using our tool SMART, which provides symbolic reachability analysis and CTL model checking with counterexample generation [57].

3.5.2.1 Termination

Reactive systems constantly respond to external events. However, if the system has a livelock, a finite number of external events can trigger an infinite number of rule executions (i.e, activate a cycle of internal events), causing the system to remain "busy" internally, a fatal design error. When generating the state space, all legal batches of events are considered. Due to the PN execution semantics, we can again avoid the need for an explicit enumeration, this time, of event batches.

Proposition 3.6. A set $\mathcal{G}$ of ECA rules satisfies termination if no infinite sequence of internal events can be triggered in any possible execution of $\mathcal{G}$. This can be expressed in CTL as $\neg\mathsf{EF}(\mathsf{EG}(\mathit{unstable}))$, stating that there is no cycle of unstable states reachable from an initial, thus stable, state.

```
bool Term(mdd S_init, mdd2 N_int)
 1  mdd S_rch ← StateSpaceGen(S_init, N_ext ∪ N_int);
 2  mdd S_unst ← Intersection(S_rch, ExtractUnprimed(N_int));
 3  mdd S_p ← EF(EG(S_unst));
 4  if S_p ≠ ∅ then return false                        • provide error trace
 5  else return true;
```

```
mdd ExtractUnprimed(mdd2 p)                              • p unprimed
 6  if p = 1 then return 1;
 7  if CacheLookUp(ExtractUnprimedCode, p, r) return r;
 8  foreach i ∈ V_{p.v} do
 9    mdd r_i ← 0;
10    if p[i] ≠ 0 then                • p[i] is the node pointed edge i of node p
11      foreach j ∈ V_{p.v} s.t. p[i][j] ≠ 0 do
12        r_i ← Union(r_i, ExtractUnprimed(p[i][j]))
13  mdd r ← UniqueTableInsert({r_i : i ∈ V_{p.v}});
14  CacheInsert(ExtractUnprimedCode, p, r);
15  return r;
```

Figure 3.18: Algorithms to verify the termination property.

Both traditional breadth-first-search (BFS) and saturation-based [213] algorithms are suitable to compute the EG operator. Figure 3.18 uses saturation, which tends to perform much better in both time and memory consumption when analyzing large asynchronous systems. We encode transitions related to external events and environmental variable changes into $\mathcal{N}_{ext}$. Thus, the internal transitions are $\mathcal{N}_{int} = \mathcal{N} \setminus \mathcal{N}_{ext}$. After generating the state space $\mathcal{S}_{rch}$ using constrained saturation [212], we build the set of states $\mathcal{S}_{unst}$ by symbolically intersecting $\mathcal{S}_{rch}$ with the unprimed, or "from", states extracted from $\mathcal{N}_{int}$. Then, we use the CTL operators EG and EF to identify any non-terminating path (i.e., cycle).

3.5.2.2 Confluence

Confluence is another desirable property to ensure consistency in systems exhibiting highly concurrent behavior.

Proposition 3.7. A set $\mathcal{G}$ of ECA rules satisfying termination also satisfies confluence

if, for any legal batch b of external events and starting from any particular stable state s, the system eventually reaches a unique stable state.

We stress that what constitutes a legal batch b of events depends on state s, since the condition portion of one or more rules might affect whether b (or a subset of b) can trigger a rule (thus close a batch). Given a legal batch b occurring in stable state s, the system satisfies confluence if it progresses from s by traversing some (nondeterministically chosen) sequence of unstable states, eventually reaching a stable state uniquely determined by b and s. Checking confluence is therefore expensive [4], as it requires verifying the combinations of all stable states reachable from $\mathcal{S}_{init}$ with all legal batches of external events when the system is in that stable state. A straightforward approach enumerates all legal batches of events for each stable state, runs the model, and checks that the set of reachable stable states has cardinality one. We instead only check that, from each reachable unstable state, exactly one stable state is reachable; this avoids enumerating all legal batches of events for each stable state. Since nondeterministic execution in the performing phase is the main reason for a system violates confluence, checking the evolution starting from unstable states will fulfill the purpose.

The brute force algorithm *ConfExplicit* in Figure 3.19 enumerates unstable states and generates reachable states only from unstable states using constrained saturation [212]. Then, it counts the stable states in the obtained set. We observe that, starting from an unstable state u, the system may traverse a large set of unstable states before reaching a stable state. If unstable state u is reachable, so are the unstable states reachable from it. Thus, the improved version *ConfExplicitImproved* first picks an unstable state $\mathbf{i}$ and, after generating the states reachable from $\mathbf{i}$ and verifying that they include only one stable state, it excludes all visited unstable states (Line 11). Further-

```
bool ConfExplicit(mdd S_st, mdd S_unst, mdd2 N_int)
 1  foreach i ∈ S_unst
 2    mdd S_i ← StateSpaceGen(i, N_int);
 3    if Cardinality(Intersection(S_i, S_st)) > 1 then
 4      return false;                                   • provide error trace
 5  return true;
```

```
bool ConfExplicitImproved(mdd S_st, mdd S_unst, mdd2 N_int, mdd2 N)
 5  mdd S_frontier ← Intersection(RelProd(S_st, N), S_unst);
 6  while S_frontier ≠ ∅ do              • if S_frontier is empty, it explores all S_unst
 7    pick i ∈ S_frontier;
 8    mdd S_i ← StateSpaceGen(i, N_int);
 9    if Cardinality(Intersection(S_i, S_st)) > 1 then
10      return false;                                   • provide error trace
11    else                                 • exclude all unstable states reached by i
12      S_frontier ← S_frontier \ Intersection(S_i, S_unst);
13  return true;
```

Figure 3.19: Explicit algorithms to verify the confluence property.

more, it starts only from states $\mathbf{i}$ in the frontier, i.e., unstable states reachable in one step from stable states (all other unstable reachable states are by definition reachable from this frontier). However, we stress that these, as most symbolic algorithms, are heuristics, so they are not guaranteed to work better than the simpler approaches.

Next, we introduce a fully symbolic algorithm to check confluence in Figure 3.20. It first generates the transition transitive closure (TC) set from $\mathcal{N}_{int}$ using constrained saturation [213], where the "from" states of the closure are in $\mathcal{S}_{unst}$ (Line 1). The resulting set encodes the reachability relation from any reachable unstable state without going through any stable state. Then, it filters this relation to obtain the relation from reachable unstable states to stable states by constraining the "to" states to set $\mathcal{S}_{st}$. Thus, checking confluence reduces to verifying whether there exist two different pairs $(\mathbf{i}, \mathbf{j})$ and $(\mathbf{i}, \mathbf{j}')$ in the relation: Procedure *CheckConf* implements this check symbolically. While computing TC is an expensive operation [213], this approach avoids separate searches from distinct unstable states and is particularly appropriate when $\mathcal{S}_{unst}$ scales to large sizes.

```
bool ConfSymbolic(mdd S_st, mdd S_unst, mdd2 N_int)
 1  mdd2 TC ← ConstraintedTransitiveClosure(N_int, S_unst);
 2  mdd2 TC_u2s ← FilterPrimed(TC, S_st);
 3  return CheckConf(TC_u2s);
```

```
bool CheckConf(mdd2 p)
 4  if p = 1 then return true
 5  if CacheLookUp(CheckConfCode, p, r) return r;
 6  foreach i ∈ V_p.v, s.t. exist j, j' ∈ V_p.v, j ≠ j', p[i][j] ≠ 0, p[i][j'] ≠ 0 do
 7    foreach j, j' ∈ V_p.v, j ≠ j' s.t. p[i][j] ≠ 0, p[i][j'] ≠ 0 do
 8      if p[i][j] = p[i][j'] return false;              • Confluence does not hold
 9      mdd f_j ← ExtractUnprimed(p[i][j]);                 • Result will be cached
10      mdd f_j' ← ExtractUnprimed(p[i][j']);             • No duplicate computation
11      if Intersection(f_i, f_j') ≠ 0 then return false;
12  foreach i, j ∈ V_p.v s.t. p[i][j] ≠ 0 do
13    if CheckConf(p[i][j]) = false return false;
14  CacheInsert(CheckConfCode, p, true);
15  return true;
```

Figure 3.20: Fully symbolic algorithm to verify the confluence property.

3.5.3 Experimental results

Table 3.2 reports results for a set of models run on an Intel Xeon 2.53GHz workstation with 36GB RAM under Linux. For each model, it shows the state space size ($|\mathcal{S}_{rch}|$), the peak memory (M_p), and the final memory (M_f). For termination, it shows the time used to verify the property (T_t) and to find the shortest counterexample (T_c). For confluence, it reports the best runtime between our two explicit algorithms (T_{be}) and for our symbolic algorithm (T_s). Memory consumption accounts for both decision diagrams and operation caches.

Net PN_t is the model corresponding to our running example in Figure 2.6, and fails the termination check. Even though the state space is not very large, counterexample generation is computationally expensive [214] and consumes most of the runtime. The shortest counterexample generated by SMART has a long tail consisting of 1885 states and leads to the 10-state cycle of Figure 3.21 (only the non-empty places

Termination (time: sec, memory: MB)					
Model	$\lvert\mathcal{S}_{rch}\rvert$	T_t	T_c	M_p	M_f
PN_t	$2.66 \cdot 10^6$	0.009	9.665	358.68	88.36
PN_c	$2.61 \cdot 10^6$	0.005	9.497	344.42	87.88
PN_1	$8.99 \cdot 10^6$	0.010	11.559	391.52	89.78
PN_2	$1.78 \cdot 10^7$	0.010	24.477	673.33	158.66
PN_3	$2.61 \cdot 10^7$	0.010	85.171	1686.46	559.52
PN_4	$5.02 \cdot 10^7$	0.010	14.541	491.80	105.90

Confluence (time: min, memory: GB, –: out of memory)							
Model	$\lvert\mathcal{S}_{rch}\rvert$	Best Explicit			Symbolic		
		T_{be}	M_p	M_f	T_s	M_p	M_f
PN_c	$2.38 \cdot 10^6$	4.51	4.24	4.10	5.11	2.02	0.22
PN_1	$8.12 \cdot 10^6$	40.25	14.53	14.33	6.40	2.31	0.27
PN_2	$1.61 \cdot 10^7$	34.40	0.85	0.08	10.11	2.59	0.25
PN_3	$2.33 \cdot 10^7$	> 120.00	–	–	60.09	2.59	0.25
PN_4	$4.55 \cdot 10^7$	> 120.00	–	–	23.33	4.66	0.52

Table 3.2: Results of verifying the ECA rules for a smart home in Figure 2.6.

are listed for each state, and edges are labeled with the corresponding PN transition). Analyzing the trace, we can clearly see (in bold) that, when lights are about to be turned off due to the timeout, $lMtn = 0$, and the external light is low, $ExtLgt \leq 5$, the infinite sequence of internal events $(LgtsOff, ChkExtLgt, LgtsOn, ChkMtn)^\omega$ prevents the system from terminating. Thus, rules r_4, r_5, r_6, and r_7 need to be investigated to fix the error. Among the possible modifications, we choose to replace rule r_5 with r'_5 : **on** *ChkExtLgt* **if** $((intLgts = 0$ **and** $lExtLgt \leq 5)$ **and** $\underline{lMtn = 1}$) **do activate** (*LgtsOn*), resulting in

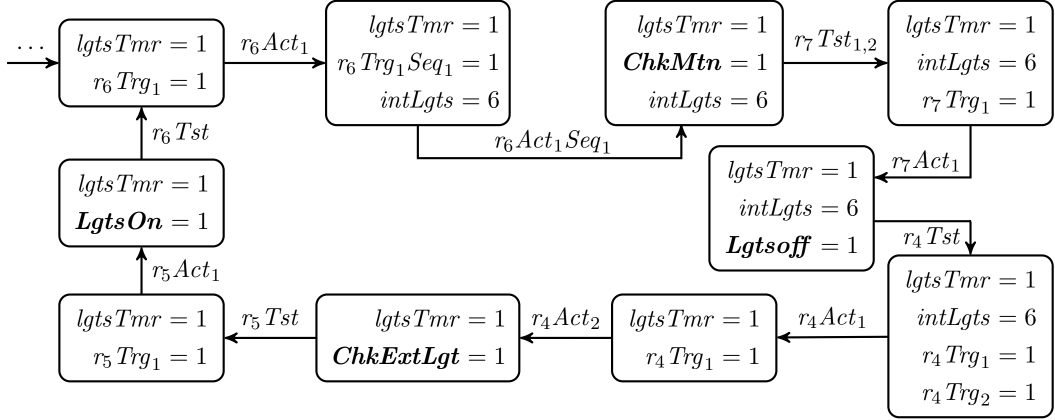

Figure 3.21: A termination counterexample (related to rules r_4 to r_7).

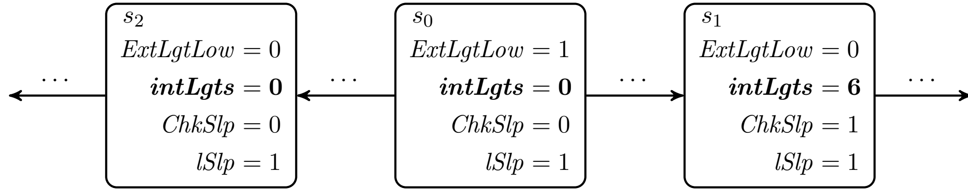

Figure 3.22: A confluence counterexample (related to rules r_8 and r_9).

the addition of an input arc from *lMtn* to r_5Tst. The new corrected model is called PN_c in Table 3.2, and SMART verifies it holds the termination property.

We then run SMART on PN_c to verify confluence, and found 72,644 bad states. Figure 3.22 shows one of these unstable states, s_0, reaching two stables states, s_1 and s_2. External event *ExtLgtLow* closes the batch in s_0 and triggers rule r_8, which sets *intLgt* to 6 and activates internal event *ChkSlp*, which in turn sets *intLgt* to 0 (we omit intermediate unstable states from s_0 to s_1 and to s_2). Recall rule r_8: **on** *ExtLgtLow* **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*) and rule r_9: **on** *ChkSlp* **if** (*lSlp* = 1) **do set** (*intLgts*, 0), in Figure 2.6. We correct and replace them with r_8': **on** *ExtLgtLow* **if** *lSlp*=0 **do set** (*intLgt*, 6) and r_9': **on** *ExtLgtLow* **if** *lSlp* = 1 **do set** (*intLgt*, 0); resulting in model PN_{fc}. Checking this new model for confluence, we find that the number of bad states decreases from 72,644 to 24,420. After investigation, we determine that the remaining problem is related to rules r_2 and r_3. After changing rule r_2 to

on *SecElp* **if** (($lgtsTmr \geq 1$ **and** $lgtsTmr \leq 359$) **and** $lMtn = 0$) **do increase** ($lgtsTmr$, 1), the model passes the check. This demonstrates the effectiveness of counterexamples to help a designer debug a set of ECA rules.

We then turn our attention to larger models, which extend our original model by introducing four additional rules and increasing variable ranges. In PN_1 and PN_2, the external light variable *ExtLgt* ranges in $[0, 20]$ instead of $[0, 10]$; for PN_4, it ranges in $[0, 50]$. PN_2 also extends the range of the light timer variable *lgtTmr* to $[0, 720]$; PN_3 to $[0, 3600]$. We observe that, when verifying termination or confluence, the time and memory consumption tends to increase as the model grows; also, our symbolic algorithm scales much better than the best explicit approach when verifying confluence. For the relatively small state space of PN_t, enumeration is effective, since computing TC is quite computationally expensive. However, as the state space grows, enumerating the unstable states consumes excessive resources. We also observe that the supposedly improved explicit confluence algorithm sometimes makes things worse. The reason may lie in the fact that a random selection of a state from the frontier has different statistical properties than for the original explicit approach, and also in the fact that operation caches save many intermediate results. However, both explicit algorithms run out of memory on PN_3 and PN_4. Comparing the results for PN_3 and PN_4, we also observe that larger state spaces might require fewer resources. With symbolic encodings, this might happen because the corresponding MDD is more regular than the one for a smaller state space.

3.6 Conclusion

This chapter presented a way to use symbolic techniques to verify discrete systems. We mainly focused on discrete systems specified using an NCFSM or ECA rules. For an NCFSM system, the details of how to encode an NCFSM using symbolic encoding give the foundation for applying advanced saturation algorithms. Symbolic algorithms are derived to check for livelocks, dead transitions, and strong connectedness. Then, we explain one of our main contributions: designing and using a symbolic equivalence checker to generate a test suite consisting of input sequences which distinguish all non-equivalent mutants of an NCFSM requirement. The challenge lies in storing and computing a large amount of similarly structured mutants, along with the well-known state space explosion problem for verification. The experimental results demonstrate the effectiveness of symbolic algorithms. For hybrid systems specified using ECA rules, we tackle two critical problems to verify both termination and confluence properties. The challenge comes from the systems' highly concurrent and nondeterministic nature. We propose an approach to verify these properties using a self-modifying PN with inhibitor arcs and priorities. Our approach is general enough to give precise answers to questions about other properties, certainly those that can be expressed in CTL. The important contribution is to develop a fully symbolic algorithm to verify the confluence property for a set of ECA rules and provide the first practical experimental results. The counterexample provided by the algorithm leads us to design flaws and helps designers to fix errors. We show this whole debugging process through a light control subsystem from a smart home for senior housing. In this chapter, we use discretization to analyze hybrid systems; the next chapter will explore techniques without discretization.

Chapter 4

Verification and Validation of Hybrid Systems

Analyzing hybrid systems in a discrete manner simplifies the problem and allows us to use powerful tools we would not otherwise be able to apply. However, handling the continuous nature of hybrid systems requires many workarounds in our analysis, for instance, discretizing continuous time into integers used in analyzing ECA rules. The choice of granularity and precision in such instances is often critical to the analysis. Too large of steps results in the oversight of time sensitive errors while too fine of steps results in wasting computational resources unnecessarily. In this chapter, we will analyze hybrid systems directly as opposed to discretizing continuous dynamics. On the positive side, we have powerful modeling languages such as HA to model physical phenomena accurately with mathematical formulas, which also provide a way to verify the properties of hybrid systems by checking the execution trajectories of given HA. On the negative side, reachability analysis for HA is undecidable [108] even under severe limitations. Thus, modeling and analyzing real applications with HA is not practical.

Instead, simulation-based approaches, i.e., modeling hybrid systems as signals, are considered a reasonable approach, and have been widely adopted in industry due to their scalability and expressiveness. Moreover, the temporal logic designed for analyzing HA can be directly transplanted to this approach to analyze signals. In this chapter, we will cover both approaches.

4.1 Verification of hybrid automata

Recall that a hybrid automaton $H = (Loc, Var, Labels, \Sigma_{Init}, Inv, Flow, Edges, Jump)$ generally has an infinite potential state space Σ as defined in Section 2.3.1. As in Chapter 3, reachability analysis is the foundation to verify HA. If it is able to compute a set of hybrid states Σ_{rch} reached by executions of H starting from Σ_{Init} through both discrete jumps and continuous flow, a large portion of verification problems, for instance safety verification, can be done through checking whether $\Sigma_{rch} \cap \Sigma_{bad}$ is empty or not, where Σ_{bad} denotes the unsafe hybrid states. However, it is well known that the exact computation of Σ_{rch} is undecidable for general HA [6, 108]. Even for our simple thermostat example shown in Figure 2.7, the reachability problem is still undecidable since the flow activities $\dot{x} = K(H - x)$ and $\dot{x} = -Kx$ are nonlinear.

Thus, researchers restrict to LHA and initialized RHA for verification purposes because both lie on the boundary between decidable and undecidable problems. If LHA are simple enough to be transfered into timed automata, the reachability problem can be solved by an algorithm designed for timed automata [5]. Normally, the simplicity condition restricts all functions in *Inv* and *Jump* to be of the form $x \leq k$ or $x \geq k$, where $x \in Var$ and $k \in \mathbb{Z}$ [6]. However, the reachability problem is undecidable for 2-rate timed systems or simple integrator systems [6]. For initialized RHA, there are

```
bool ForwardAnalysis(Σ_bad, Σ_Init)
1  Σ_old ← ∅;
2  Σ_cur ← Σ_Init;
3  while Σ_cur ∩ Σ_bad = ∅ ∧ Σ_cur ≠ Σ_old do
4    Σ_next ← Post(Σ_cur);
5    Σ_old ← Σ_old ∪ Σ_cur;
6    Σ_cur ← Σ_next;
7  end while
8  return Σ_curr ∩ Σ_bad ≠ ∅;
```

```
bool BackwardAnalysis(Σ_bad, Σ_Init)
1  Σ_old ← ∅;
2  Σ_cur ← Σ_bad;
3  while Σ_cur ∩ Σ_init = ∅ ∧ Σ_cur ≠ Σ_old do
4    Σ_prev ← Prev(Σ_cur);
5    Σ_old ← Σ_old ∪ Σ_cur;
6    Σ_cur ← Σ_prev;
7  end while
8  return Σ_curr ∩ Σ_Init ≠ ∅;
```

Figure 4.1: Fixed-point algorithms for the forward and backward analysis on state regions of HA.

two important limitations: (1) all initial hybrid states, *Inv*, and (2) *Flow* are defined in rectangular regions and any continuous variable has to be reinitialized whenever it takes a discrete jump transition [108]. These limitations strongly limit real applications that can be modeled and analyzed using HA.

Assuming the reachability problem of the inspected HA is decidable, the ordinary fixed-point algorithms operating on state sets do not work properly. Thus, instead of directly analyzing state sets, researchers define state regions that are finite quotients of the state space [8, 12]. A state region contains a hybrid state (l, v) as the kernel and other states that are reachable from the time transition $(l, v) \Rightarrow_t (l, v')$. Since all the functions and conditions are linear, the values of all real variables change linearly. Thus, a set of possible valuations can be defined by a set of linear formulas. Intuitively, if *Var* has n real variables, a set of evaluations can be graphically represented by the union of polyhedra in $\mathbb{R}^n$. Then, a state region can be represented by a location indicator as a set of linear formulas. State region representation is the cornerstone of the *Prev* and *Post* computations, which are defined as:

$$Prev(\Sigma_{cur}) = \{(\mathbf{l}, \mathbf{v}) \in \Sigma | \Sigma_{cur} \Rightarrow (\mathbf{l}, \mathbf{v})\}$$

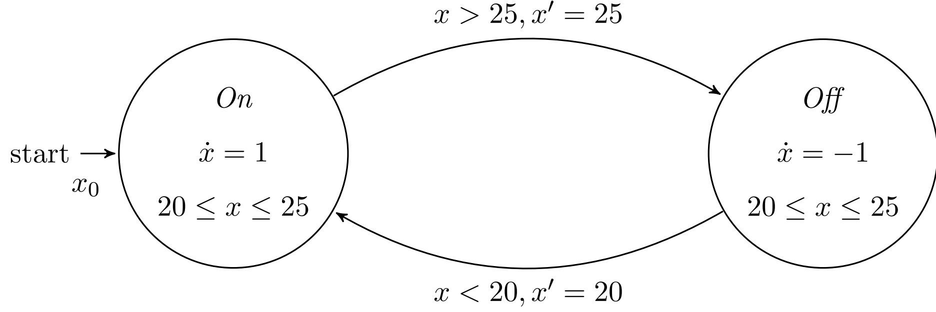

Figure 4.2: The initialized RHA for the thermostat in Figure 2.7.

and

$$Post(\Sigma_{cur}) = \{(\mathbf{l}, \mathbf{v}) \in \Sigma | (\mathbf{l}, \mathbf{v}) \Rightarrow \Sigma_{cur}\},$$

where $\mathbf{l} \in Loc$, $\mathbf{v} \in V$, and the simple progress $\Rightarrow$ is defined in Section 2.3.1.1.

Figure 4.1 shows the algorithms for forward and backward fixed-point computations on state regions [6]. These algorithms provide a semi-decision procedure for reachability analysis of linear hybrid systems. If the final number of state regions is infinite, these algorithms do not terminate. Otherwise, they would verify whether the system can reach any of the states in the bad state region Σ_{bad} from the initial state region Σ_{init}. If we further simplify our thermostat example in Figure 2.7 to the initialized RLA shown in Figure 4.2 with $\Sigma_{Init} = \{(On, 20 \leq x \leq 25)\}$, we obtain that $\Sigma_{rch} = \{(On, 20 \leq x \leq 25), (Off, 20 \leq x \leq 25)\}$. Thus, the safety property with $\Sigma_{bad} = \{Off \vee On, x < 20 \wedge x > 25\}$ holds.

Researchers extended CTL, which was originally defined to reason about the temporal behaviors of discrete reactive systems, to temporal logics that are able to reason both continuous and discrete behaviors of a hybrid system and express properties related to time, such as real-time computational tree logic (TCTL) [7] and integrator computational tree logic (ICTL) [12]. TCTL was originally proposed for timed automata [109]

and introduces an extra set of clock variables and a reset quantifier. ICTL enhances TCTL by allowing the integrator, a stop watch that can be stopped and restarted. The integrator is useful when expressing duration of properties. Both logics are implemented in the symbolic model checking tool HYTECH [111] based on the above two fixed-point algorithms in Figure 4.1. The exact analysis of LHA is undecidable for both TCTL and ICTL [7].

Similar extensions from LTL are timed propositional temporal logic (TPTL) [13] and metric temporal logic (MTL) [132]. MTL introduces time-bounded temporal operations and has been widely adopted by validating hybrid systems modeled as signals, since linear time based logic is more natural than branching time based logic to describe signals. Again, the satisfiability problem for MTL is undecidable [11].

4.2 STL and MTL

MBD promotes the use of executable block diagrams to represent and simulate a hybrid system, such as the model designed using Simulink in Figure 2.9. This model and design method seamlessly combines visualization and simulation in a manner well-suited for hierarchical and compositional design. Graphical block diagrams, which represent mathematical equations, are supported by a simulation engine to generate signals as outputs. Also, MBD provides the ability to directly generate production code from models which greatly reduces the possibility of introducing errors during the implementation. Then, early validation can be applied to models, and increase the probability of discovering errors even at the design phase. Since the verification of hybrid systems is extremely challenging, inevitably expensive, and possibly non-terminating, even for severely limited hybrid systems, researchers have shifted their focus from rigorous veri-

fication to more relaxed simulation-based validation [92, 66, 171, 199].

Researchers have adopted fruitful results from verification to develop, metric temporal logic (MTL) [132] and, more recently, signal temporal logic (STL) [152] to analyze systems modeled as input and output signals. STL and MTL extend the temporal operators G, F, and U from LTL. Each temporal operator is indexed by an interval of the form (a, b), $(a, b]$, $[a, b)$, $[a, b]$, (a, ∞), or $[a, \infty)$, where both a and b are non-negative real-valued constants with $a \leq b$.

The subtle difference between STL and MTL is in how they to map signals to atomic propositions. STL is more specific than MTL and uses predicates to express constraints on signals. These predicates can then be reduced to inequalities of the form

$$\mu = f(\mathbf{x}) \sim \pi_{const}, \tag{4.1}$$

where f is a scalar-valued function over the signal $\mathbf{x}$, $\sim \in \{<, \leq, \geq, >, =, \neq\}$, and π_{const} is a real number. In the following sections, we mainly focus on STL.

Definition 4.1. (STL syntax) Given an interval I, an STL formula is inductively defined by the following grammar:

$$\varphi := \top \mid \mu \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathsf{U}_I \, \varphi_2$$

where μ is a predicate defined on signal $\mathbf{x}$ as in Equation (4.1). □

Similar to LTL, the always and eventually operators are derived from the until operator as follows:

$$\mathsf{F}_I\varphi = \top \, \mathsf{U}_I \, \varphi \qquad \mathsf{G}_I\varphi = \neg\mathsf{F}_I\neg\varphi.$$

When the interval I is omitted, the default interval $[0, +\infty)$ is implied.

Intuitively, the semantics of STL formulas are defined as follows. The signal $\mathbf{x}$ satisfies $f(\mathbf{x}) > 10$ at time t, where $t \geq 0$, if $f(\mathbf{x}(t)) > 10$. It satisfies $\varphi = \mathsf{G}_{[0,2)} \, (x > -1)$

if for all time $0 \leq t < 2$, $x(t) > -1$. The signal $\mathbf{x}_1$ satisfies $\varphi = \mathsf{F}_{[1,2)}\mathbf{x}_1 > 0.4$ iff there exists time t such that $1 \leq t < 2$ and $\mathbf{x}_1(t) > 0.4$. The two-dimensional signal $\mathbf{x} = (x_1, x_2)$ satisfies the formula $\varphi = (x_1 > 10)\ \mathsf{U}_{[2.3,4.5]}\ (x_2 < 1)$ iff there is some time u where $2.3 \leq u \leq 4.5$ and $x_2(u) < 1$, and for all time v in $[2.3, u)$, $x_1(v)$ is greater than 10.

Definition 4.2. (STL boolean semantics) Given a signal $\mathbf{x}$ at time t, the boolean semantics of an STL formula φ is inductively defined as follows:

$(\mathbf{x}, t) \models \mu$	iff	$\mathbf{x}$ satisfies μ at time t
$(\mathbf{x}, t) \models \neg\varphi$	iff	$(\mathbf{x}, t) \not\models \varphi$
$(\mathbf{x}, t) \models \varphi_1 \wedge \varphi_2$	iff	$(\mathbf{x}, t) \models \varphi_1$ and $(\mathbf{x}, t) \models \varphi_2$
$(\mathbf{x}, t) \models \varphi_1\ \mathsf{U}_{[a,b]}\ \varphi_2$	iff	$\exists\, t' \in [t+a, t+b]$ s.t. $(\mathbf{x}, t') \models \varphi_2$ and $\forall\, t'' \in [t, t'], (\mathbf{x}, t'') \models \varphi_1$ □

Extending the above semantics to other kinds of intervals (open, open-closed, and closed-open) is straightforward. We write $\mathbf{x} \models \varphi$ as a shorthand for $(\mathbf{x}, 0) \models \varphi$.

4.3 Quantitative semantics

The boolean semantics combined with the definition of an atomic proposition μ abstract the real value signal $\mathbf{x}$ to a set of boolean signals based on inequalities. This abstraction keeps the information of timing, but largely loses information such as amplitude, which is sometimes critical when designing hybrid systems. For instance, given an STL formula $\varphi = \mathsf{G}(\mathbf{x} < 15 \wedge \mathbf{x} > -15)$, two signals x_1 and x_2 shown in Figure 4.3 will be abstracted to the same boolean signal, straight true through the whole time domain. However, it is obvious that x_1 is closer to the threshold than x_2. A little perturbation on the system generating x_1, introduced by environmental noise

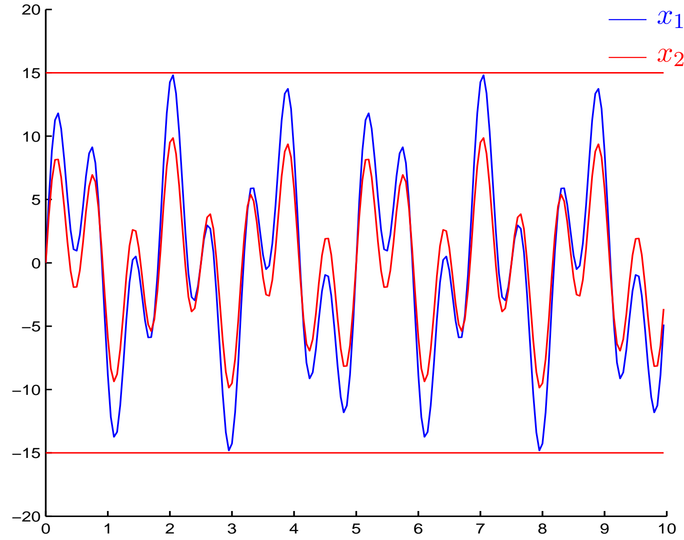

Figure 4.3: With respect to STL formula $\varphi = \mathsf{G}(\mathbf{x} < 15 \wedge \mathbf{x} > -15)$, using Boolean semantics, signals x_1 and x_2 will be abstracted to the same Boolean signal, straight true for the whole time domain. Thus, both of them satisfy φ. However, it is straightforward that x_1 is closer to the threshold to violate φ, comparing to x_2. Unfortunately, Boolean semantics is unable to recognize this difference.

or the measurement precision of devices, will cause the resulting signal to violate φ, as in Figure 4.4. Unfortunately, the boolean semantics is unable to distinguish whether x_2 is more tolerant than x_1 with respect to φ. Thus, instead of qualitative semantics, we need an advanced semantics based on quantitative measurement to differentiate the tolerance of signals against perturbation, also referred as robustness.

The quantitative semantics of STL can be defined using a real-valued function ρ of a signal $\mathbf{x}$, a formula φ, and time t satisfying the following property:

$$\rho(\varphi, \mathbf{x}, t) \geq 0 \text{ iff } (\mathbf{x}, t) \models \varphi. \tag{4.2}$$

This quantitative semantics preserves the satisfaction of boolean semantics and also perfectly captures the notion of robustness satisfaction of φ by a signal $\mathbf{x}$. For example, whenever the absolute value of $\rho(\varphi, \mathbf{x}, t)$ is large, a little perturbation on the system is less likely to affect the boolean satisfaction (or violation) of φ by $\mathbf{x}$. In [85], different quantitative semantics for STL have been proposed. The most commonly-used semantics

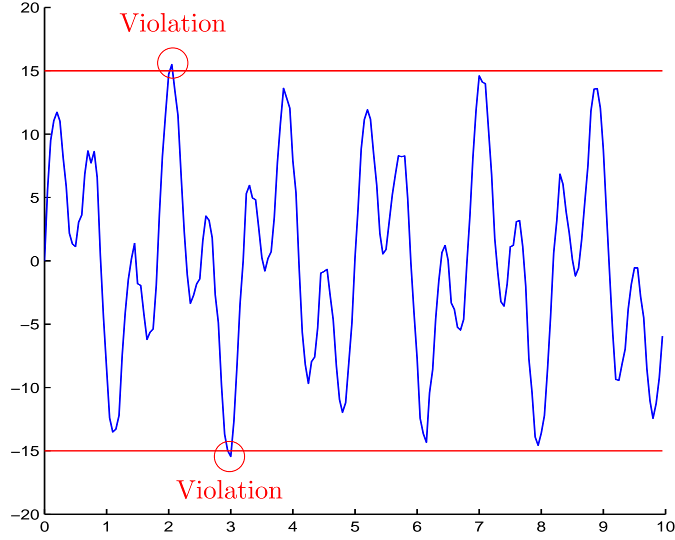

Figure 4.4: After introduced some random noise to the system generating x_1 in Figure 4.3, the resulting signal violates the same STL formula $\varphi = \mathsf{G}(\mathbf{x} < 15 \wedge \mathbf{x} > -15)$.

is defined inductively from the quantitative semantics for predicates and inductive rules for each STL operator. The quantitative semantic is similar to the robust semantic of MTL which is based on the definition of a metric on the state space of signals [92]. This metric should be able to identify each predicate with the set in which the predicate holds.

Definition 4.3. (metric) Let d be a metric on $\mathbb{R}^n$ with the usual extension to the signed distance from a point $p \in \mathcal{P}$ to a set $\mathcal{P}' \subseteq \mathcal{P}$:

$$d(x, \mathcal{P}') = \begin{cases} -\inf\limits_{p' \in \mathcal{P}} d(p, p') \text{ if } x \notin \mathcal{P}' \\ \inf\limits_{p' \in \mathcal{P} \setminus \mathcal{P}'} d(p, p') \text{ otherwise} \end{cases}$$

where $d(p, p')$ is a normal distance function which satisfies the following conditions:

- non-negativity: $d(p, p') \geq 0$;
- identity of indiscernibles: $d(p, p') = 0$ iff $p = p'$;
- symmetry: $d(p, p') = d(p', p)$;

- triangle inequality: $d(p,p'') \leq d(p,p') + d(p',p'')$, where $p'' \in \mathcal{P}$. $\square$

The signed distance provides a way to measure whether a point belongs to a set and how far it is from the set. For example, when the point belongs to the set, the signed distance is the shortest non-negative distance from the point to the boundary of the set. For each MTL predicate μ, first define its truth set $\mathcal{O}(\mu)$ as: $\mathbf{x}, t \models \mu$ iff $\mathbf{x}(t) \in \mathcal{O}(\mu)$ and let $\rho_d(\mu, \mathbf{x}, t) = d(\mathbf{x}(t), \mathcal{O}(\mu))$.

Without loss of generality, an STL predicate μ can be identified by an inequality of the form $f(\mathbf{x}) \geq 0$. Other forms of inequalities can be transformed into this form. From this form, a straightforward quantitative semantics for predicate μ is defined as

$$\rho(\mu, \mathbf{x}, t) = f(\mathbf{x}(t)). \tag{4.3}$$

Thus, we can conclude that the robust semantics of MTL is a special case of the quantitative semantics of STL. When the associated function f for each predicate in STL is defined as the signed distance d, ρ_d and ρ coincide.

Definition 4.4. (STL quantitative semantics) Given signal $\mathbf{x}$, the quantitative semantics of STL formula φ is defined inductively as follows:

$$\rho(\neg\varphi, \mathbf{x}, t) = -\rho(\varphi, \mathbf{x}) \tag{4.4}$$

$$\rho(\varphi_1 \wedge \varphi_2, \mathbf{x}, t) = \min(\rho(\varphi_1, \mathbf{x}, t), \rho(\varphi_2, \mathbf{x}, t)) \tag{4.5}$$

$$\rho(\varphi_1 \mathsf{U}_I \varphi_2, \mathbf{x}, t) = \sup_{t' \in t+I} \big(\min(\rho(\varphi_2, \mathbf{x}, t'), \inf_{t'' \in [t,t')} \rho(\varphi_1, \mathbf{x}, t'')\big), \tag{4.6}$$

where sup and inf are the supremum and infimum functions. $\square$

Additionally, by combining Equation (4.6), and the derived definition of $\mathsf{F}_I\varphi$

and $\mathsf{G}_I\varphi$, we can obtain:

$$\rho(\mathsf{F}_I\varphi, \mathbf{x}, t) = \sup_{t' \in t+I} \rho(\varphi, \mathbf{x}, t') \tag{4.7}$$

$$\rho(\mathsf{G}_I\varphi, \mathbf{x}, t) = \inf_{t' \in t+I} \rho(\varphi, \mathbf{x}, t'). \tag{4.8}$$

Example 4.5. For the automatic transmission controller model from Section 2.3.2.1, suppose we want to specify that the `speed` should never exceed 120 mph and `RPM` should never exceed 4500 rpm. The predicate specifying the former constraint is $\texttt{speed} > 120$ and the latter is $\texttt{RPM} > 4500$. The STL formula expressing these to be always false,

$$\varphi = \mathsf{G}(\texttt{speed} \leq 120) \wedge \mathsf{G}(\texttt{RPM} \leq 4500), \tag{4.9}$$

has two predicates $\mu_1 : \texttt{speed} \leq 120$ and $\mu_2 : \texttt{RPM} \leq 4500$. We put them into the standard form $\mu_i : f_i(\mathbf{x}) \geq 0$, and define $\mathbf{x} = (\texttt{speed}, \texttt{RPM})$, $f_1(\mathbf{x}) = 120 - \texttt{speed}$ and $f_2(\mathbf{x}) = 4500 - \texttt{RPM}$. From (4.3), we get the quantitative semantics:

$$\rho(\texttt{speed} \leq 120, \mathbf{x}, t) = 120 - \texttt{speed}(t).$$

Applying rule (4.8) for the semantics of G, we get:

$$\rho(\mathsf{G}(\texttt{speed} \leq 120), \mathbf{x}, t) = \inf_{t \in \mathbb{T}}(120 - \texttt{speed}(t)).$$

Similarly for μ_2,

$$\rho(\mathsf{G}(\texttt{RPM} \leq 4500), \mathbf{x}, t) = \inf_{t \in \mathbb{T}}(4500 - \texttt{RPM}(t)).$$

Finally, by applying rule (4.5):

$$\rho(\varphi, \mathbf{x}, t) = \min(\inf_{t \in \mathbb{T}}(120 - \texttt{speed}(t)), \inf_{t \in \mathbb{T}}(4500 - \texttt{RPM}(t)). \tag{4.10}$$

In other words, the resulting satisfaction function ρ looks for the maximum `speed` and `RPM` over time and returns the minimum of the differences with the thresholds 120 and 4500.

4.4 Falsification of hybrid systems

Unlike formal verification, which tries to study a hybrid system based on its fundamental mathematical representation and complex structure, validation only focuses on observable behaviors, namely input and output signals from simulation results. Validation and falsification treat a hybrid system as a black box. Moreover, they require no knowledge of its internal theory and complex hierarchical structure and no advanced or sophisticated techniques to track internal system state changes. For some cases, such as intellectual property (IP) protection, it is mandatory to validate the system design as a black box. Another advantage of this approach is its scalability, as it is able to handle industrial models without requiring excessive effort and puts no requirements on the dynamics of the considered hybrid system.

Given an STL formula φ and signal $\mathbf{x}$, the validation result can be directly obtained from quantitative semantics, or called robustness estimation, of the system with respect to the requirement formula. If we look at the robustness estimation from another point of view, it can be used in an entirely new application. For instance, it can be used to guide the search of input signals that may cause undesired system behaviors. This is known as the falsification problem:

Problem 4.6. (the falsification problem) Given a system $\mathcal{S}$ and an STL formula φ, the falsification problem is to find a signal $\mathbf{u}$ such that $\mathcal{S}(\mathbf{u}) \not\models \varphi$. □

Using quantitative semantics, this is equivalent to finding a trace $\mathbf{x} = \mathcal{S}(\mathbf{u})$ such that $\rho(\varphi, \mathbf{x}, 0) < 0$. The problem can be solved by

$$\text{Solve } \rho^* = \min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0) \tag{4.11}$$

Then, if $\rho^* < 0$, we return $\mathbf{u}^* = \arg\min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$, otherwise, $\mathcal{S} \models \varphi$.

Unfortunately, the resulting minimization (4.11) is a non-linear and non-convex optimization problem for general hybrid systems [210]. No solver can guarantee convergence, uniqueness, or even existence of a solution. Formally, letting φ be a simple safety property establishes a reduction from the reachability problem for general hybrid systems, which is undecidable [108]. Thus, the falsification problem is undecidable.

On the other hand, many heuristics can be used to find an approximate solution for MTL and STL formulas. Researchers proposed and implemented different strategies for MTL in S-TALIRO, namely Monte-Carlo [163], ant-colony optimization [19], and the cross entropy method [182]. STL uses Nelder-Mead non-linear optimization in BREACH [128]. Although different optimization algorithms might achieve various performance, they all use the same general framework:

1. Define the space of permissible input signals with the help of m input control parameters $\mathbf{k} = (k_1, \ldots, k_m)$ which take values from a set $\mathcal{P}_u$, and a generator function g such that $\mathbf{u}(t) = g(v(\mathbf{k}))(t)$ is a permissible input signal for $\mathcal{S}$ for any valuation $v(\mathbf{k}) \in \mathcal{P}_u$.

2. Sample the space of the control parameters in a uniform, random fashion to obtain N_{init} distinct valuations $v_i(\mathbf{k}) \in \mathcal{P}_u$.

3. For $i \leq N_{\text{init}}$, solve $\rho_i = \min\limits_{v(\mathbf{k}) \in \mathcal{P}_u} \rho(\varphi, \mathcal{S}(g(v(k))), 0)$ using different optimization algorithms and $v_i(\mathbf{k})$ as an initial guess.

4. Return the corresponding $\mathbf{u}$ with the minimal ρ value.

For example, if permissible input signals are step functions, the input parameters would characterize the amplitude of the step and the time at which the step input is applied. Note that g does not necessarily generate all possible inputs to the system.

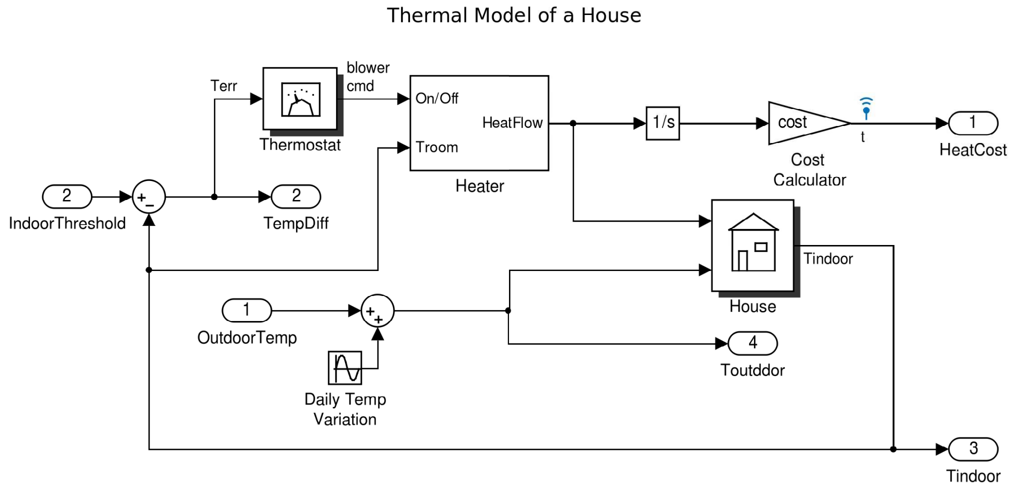

Figure 4.5: The closed-loop Simulink model of a thermostat system. This model is more advanced than the one in Section 2.3.1.2. It details the physical plant in the House block that allows designers to specify thermal properties, such as the materials of walls and windows, the geometry of the house, the temperature of the heater flow. Moreover, the model also estimates the heating cost for this specific house. The inputs of the model are the indoor temperature threshold for the thermostat and the initial outdoor temperature that is able to affect the thermal resistance of the whole system.

However, it is useful in a very generic way to restrict the search space of possible input signals. The falsification engine in Breach is more flexible in the definition of input parameters than that in S-Taliro.

4.5 Experimental results

We revisit the thermostat system that we attempted to verify using HA in Section 2.3.1.2. Here, we provide an advanced model using Simulink, shown in Figure 4.5, which originally comes from the Simulink tutorial demo [187] with some modifications. This model details the physical plant in the House block that allows designers to specify thermal properties, such as the materials of walls and windows, the geometry of the house, and the temperature of the heater flow. Moreover, the model also estimates the heating cost for this specific house. The inputs of the model are the indoor temperature threshold for the thermostat and the initial outdoor temperature that is able to affect

	S-TALIRO falsification			BREACH falsification		
Formula	Time	#Sim	Rob./$\mathbf{x}$	Time	#Sim	Rob./$\mathbf{x}$
$\varphi_{\texttt{eff}}^{50}$	9.635	17	0.566	0.362	10	0.036
$\varphi_{\texttt{eff}}^{80}$	18.342	43	0.432	0.929	12	0.036
$\varphi_{\texttt{cost}}$	> 3.977	> 1000	0.003	0.135	27	0.005

Table 4.1: The falsification results for the thermostat system in Figure 4.5.

the thermal resistance of the whole system. All properties discussed later are tried on the two falsification engines BREACH and S-TALIRO. The comparison results are shown in Table 4.1, including the total time for falsification, the number of simulations, and the time to compute the robustness value.

The first interesting property to verify is the efficiency of the designed heating system. One goal is to make sure that, when the room temperature drops 2.5°C below the threshold, the system is able to raise the room temperature to at least 1°C lower than the threshold within 15 minutes (since the simulation time unit is hour, 15 minutes equals 0.25 hour) and maintain the status for at least 15 minutes. The difference between the room temperature and the threshold is marked by the output signal *TempDiff*. Thus, the property can be formalized as

$$\varphi_{\texttt{eff}} = \mathsf{G}((\mathit{TempDiff} > 2.5) \Rightarrow (\mathsf{F}_{[0,0.25]}(\mathsf{G}_{[0,0.25]}(\mathit{TempDiff} < 1)))).$$

The counterexample provided by the falsification engine is shown in Figure 4.6. It shows that the heating system is not strong enough to raise the room temperature to 26°C, the user defined temperature threshold. Increasing the heat flow temperature from 50°C to 80°C solves this problem. Thus, in the results table, formulas $\varphi_{\texttt{eff}}^{50}$ and $\varphi_{\texttt{eff}}^{80}$ are used to distinguish this predefined heat flow temperature.

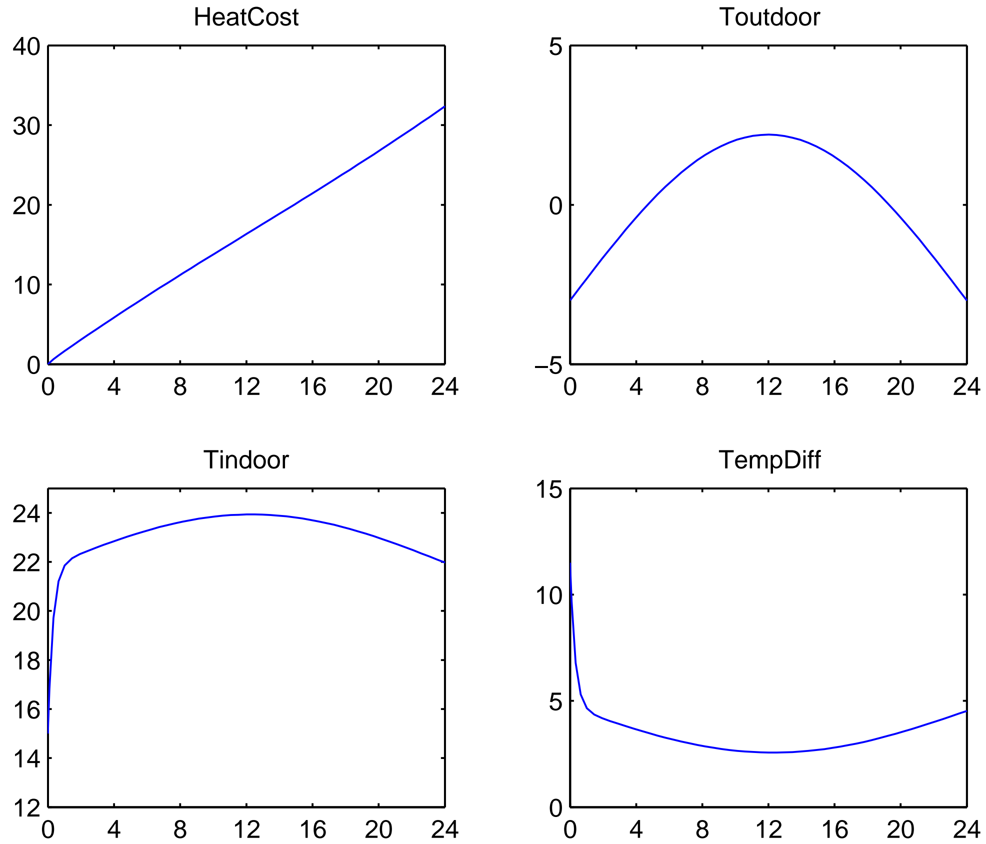

Figure 4.6: A counterexample of the formula with the condition that the predefined heat flow temperature is 50°C, $\varphi^{50}_{\texttt{eff}} = \mathsf{G}((\mathit{TempDiff} > 2.5) \Rightarrow (\mathsf{F}_{[0,0.25]}(\mathsf{G}_{[0,0.25]}(\mathit{TempDiff} < 1))))$.

However, the falsification engine returns another counterexample shown in Figure 4.7. It informs us that the system is capable of raising the room temperature fast enough, but the sealing and insulating of the house is not energy efficient enough to maintain the temperature. Thus, we double the thickness of windows and walls and reduce the total window area to increase the house insulation.

The next property is related to the daily cost. Since the heating power is boosted, the cost for electricity will increase. However, we also improve the insulation which in turn decreases the cost. The requirement specifying that the daily heating cost

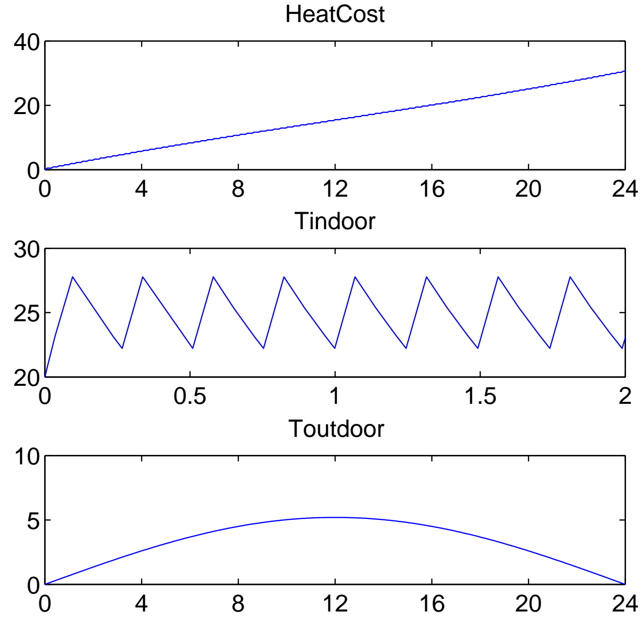

Figure 4.7: A counterexample of the formula with the condition that the predefined heat flow temperature is 80°C, $\varphi^{80}_{\texttt{eff}} = \mathsf{G}((\mathit{TempDiff} > 2.5) \Rightarrow (\mathsf{F}_{[0,0.25]}(\mathsf{G}_{[0,0.25]}(\mathit{TempDiff} < 1))))$.

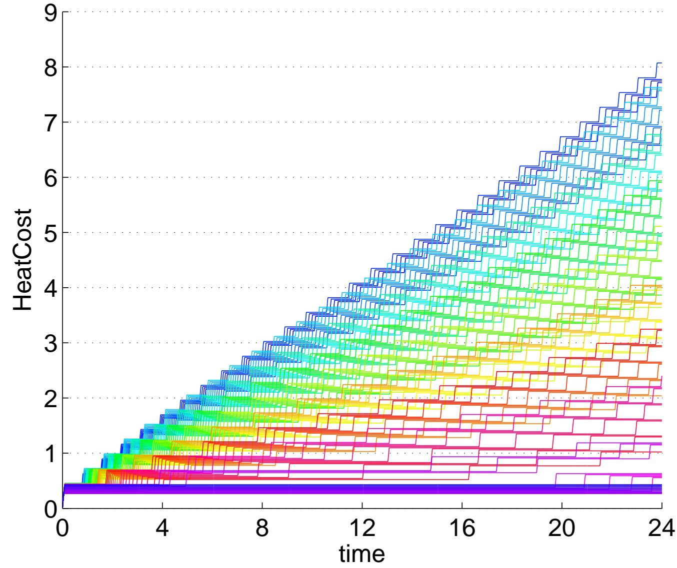

Figure 4.8: Simulation results of 100 runs against $\varphi_{\texttt{cost}} = \mathsf{G}(\mathit{HeatCost} < 8)$.

should be less then $8 can be formalized as

$$\varphi_{\texttt{cost}} = \mathsf{G}(\mathit{HeatCost} < 8).$$

Figure 4.8 shows the simulation results with 100 runs using Breach, which finds a

counterexample with the settings: the threshold temperature as 26.5°C and the initial outdoor temperature as −10°C. Unfortunately, S-TALIRO fails to find any counterexample after over 1000 attempts.

We observe that the performance of S-TALIRO is not stable. S-TALIRO uses convex stochastic optimization techniques. However, the dynamics of most hybrid systems is non-convex which causes its unstable performance. Thus, in Table 4.1, we use the average time with several runs to falsify the formula for S-TALIRO. For instance, for $\varphi^{80}_{\texttt{eff}}$, the longest run takes 46.495 seconds after 110 attempts but the most successful run costs only 3.8181 seconds with 8 attempts. As stated earlier, for $\varphi_{\texttt{cost}}$, S-TALIRO fails all runs even with 1000 attempts. If more runs are allowed, S-TALIRO may be able to falsify the formula. In comparison, BREACH is relatively stable even for relatively complex formulas. The results demonstrate that the Nelder-Mead algorithm provides a good trade-off between global randomized exploration and local optimization. Also, S-TALIRO computes the robustness value using the monitoring algorithm [93]. As for BREACH, signals are computed before computing the robustness value. Thus, the downgrade in performance of S-TALIRO with the complexity of the formula may be a result of some hidden costs due to merging robustness computation and signal generation [84].

4.6 Conclusion

This chapter illustrated verification and validation techniques for hybrid systems. Although formal modeling languages such as HA offer a solid mathematical foundation, the majority of verification problems are undecidable, even for the basic reachability problem under severe limitations. Thus, analyzing real applications using HA is not realistic. The fact that development and even maintenance of the famous HA

verification tool HYTECH [111] has been stopped for a long time is a consideration.

On the other hand, the fruitful result from the combination of simulation and formal methods, validation, gives hope of tackling this problem by ignoring the internal mathematical theory and complex hierarchical structure. The boolean semantics of validation is enhanced into multi-valued logics [79], and researchers use quantitative semantics to describe the robustness of the model with respect to a temporal logic formula. Falsification is the direct result of this extension. It uses quantitative semantics to search the input space for a counterexample that falsifies the formula. The valuable counterexample provides both input and output signals for designers to debug the designed hybrid system. These methods have been highly acknowledged as an important breakthrough. In this chapter, we also compared the performance of the two falsification engines from S-Taliro and Breach. Breach is more stable and flexible and gives better results in general, compared with S-Taliro. On the other hand, S-Taliro uses stochastic optimization techniques, and it can sometimes generate good quality counterexamples within a short time. However, the random nature of stochastic optimization occasionally frustrates design engineers because of its unstable performance. The falsification engines used in this chapter will be part of a requirement mining framework to mine formal specifications from a closed-loop model developed in the next chapter.

Chapter 5

Requirement Mining for Hybrid Systems

Industrial-scale controllers used in automobiles and avionics are now commonly developed using a MBD paradigm [188, 164]. The MBD process consists of a sequence of steps. In the first step, the designer captures the plant model, i.e., the dynamic characteristics of the physical parts of the system, using differential, logic, and algebraic equations. The next step is to design a controller that employs some specific control law to regulate the behavior of the physical system. The closed-loop model consists of at least one plant and one controller. Then, the designer may perform extensive simulations on the closed-loop model.

In an ideal world, all requirements, including high-level specifications, are well-documented and can later be used in system testing and property validation. Thus, the objective is to analyze the controller design by observing the time-varying behavior of the signals of interest. These signals result from exciting the exogenous time-varying inputs of the closed-loop model. An important aspect of this step is to use validation and

falsification, discussed in the previous chapter, to check if the time-varying behavior of the closed-loop system matches a set of requirements. Unfortunately, in practice, these requirements are often high-level and vague. Examples of requirements include, “better fuel-efficiency”, “signal should eventually settle”, and “resistance to turbulence”. Different designers may have various interpretations of the same requirements. If designers are not satisfied with the simulation results, they will refine or tune the controller and repeat the validation process.

In the formal methods literature, a requirement (also called a specification) is a mathematical expression of the design goals or desirable design properties, expressed in a suitable logic. In an industrial setting, requirements are rarely expressed formally, and it is common to find them written in natural language. Control designers then validate their design manually by comparing experimental time traces to these informal requirements. In some cases, they simply use simulation-data and their domain expertise to determine the quality of the design. Moreover, to date, formal validation tools have been unable to digest the format or scale of industrial-scale requirements. As a result, widespread adoption of formal tools has been restricted to testing syntactic coverage of the controller code, with the hope that higher coverage implies better chances of discovering bugs. It is clear that even simulation-based tools would benefit from the more semantic notions of coverage offered by formal requirements.

In this chapter, we propose a scalable technique to systematically mine requirements from the closed-loop model of an industrial-scale control system from observations of the system behavior. In addition to the closed-loop model, our technique takes as input a template requirement. The final output is a synthesized requirement matching the template. We assume that the model is specified in Simulink [186], an industry-wide standard that is able to:

- Express complex dynamics such as differential and algebraic equations.
- Capture discrete state-machine behavior by allowing both boolean and real-valued variables.
- Allow a layered design approach through modularity and hierarchical composition.
- Perform high-fidelity real-time simulations.

5.1 Parametric STL

Parametric Signal Temporal Logic (PSTL) is an extension of STL introduced in [22] to define template formulas containing unknown parameters. Syntactically speaking, a PSTL formula is an STL formula where numeric constants, either in the constraints given by the predicates μ or in the time intervals of the temporal operators, can be replaced by symbolic parameters. These parameters are divided into two types:

- A Scale parameter π is a parameter appearing in a predicate of the form $\mu = f(\mathbf{x}) \sim \pi$.
- A Time parameter τ is a parameter appearing in the interval of a temporal operator.

An STL formula is obtained from a PSTL formula by using a valuation function that assigns a value to each symbolic parameter. Consider the PSTL formula $\varphi(\tau, \pi) = \mathsf{G}_{[0,\tau]} x > \pi$, with a scale parameter π in the comparison with the signal value x and a time parameter τ to specify the uncertainty of the duration for the G operator. The STL formula $\mathsf{G}_{[0,10]} x > 1.2$ is an instance of φ obtained with the valuation $v = \{\tau \mapsto 10,\ \pi \mapsto 1.2\}$.

Example 5.1. Consider again the STL property in Formula (4.9):

$$\varphi = \mathsf{G}(\texttt{speed} \leq 120) \wedge \mathsf{G}(\texttt{RPM} \leq 4500).$$

To turn this into a PSTL formula, we rewrite it by introducing parameters π_{speed} and π_{RPM}:

$$\varphi(\pi_{speed}, \pi_{rpm}) = \mathsf{G}(\texttt{speed} \leq \pi_{speed}) \wedge \mathsf{G}(\texttt{RPM} \leq \pi_{rpm}). \tag{5.1}$$

The STL formula φ is then obtained by using the valuation $v = (\pi_{\texttt{speed}} \mapsto 120, \pi_{rpm} \mapsto 4500)$ in the PSTL Formula (5.1).

On the other hand, if we want to know how long the system will maintain that `speed` never exceeds 120 mph and `RPM` never exceeds 4500 rpm, the following PSTL can be used:

$$\varphi = \mathsf{G}_{[0,\tau_d]}(\texttt{speed} \leq 120 \wedge \texttt{RPM} \leq 4500) \tag{5.2}$$

where time parameter τ_d is used to specify the unknown duration. The STL requirement that within the first 10 seconds `speed` never exceeds 120 mph and `RPM` never exceeds 4500 rpm can be derived from Formula (5.2) by using the valuation $v = (\tau_d \mapsto 10)$.

5.2 Weighted STL and parametric weighted STL

Formalisms such as MTL and STL are adept at capturing both the real-valued and time-varying behaviors of hybrid control systems. PSTL is particularly well-suited to express template requirements to be mined. Thus, STL and PSTL are perfect candidates for the requirement mining framework.

However, we observe that STL and PSTL do not distinguish the different contributions associated with each predicate when computing the quantitative robustness value. Consider Example 5.1 for the automatic transmission model in Figure 2.9. Predicates $\mathsf{G}(\texttt{speed} \leq 120)$ and $\mathsf{G}(\texttt{RPM} \leq 4500)$ contribute the same even though `speed` and

RPM are measured in different units. For example, speed $= 150$ has the same robustness value contribution as RPM $= 4530$ according to the semantics defined in Section 4.3 because both values are 30 over their thresholds. Even so, a counterexample with a 30 mph increase on speed is much more meaningful than one with a 30 RPM increase. To better describe the desirable and valuable behaviors of the system, we propose new weighted temporal logics: weighted STL (WSTL) and parametric weighted STL (PWSTL).

Definition 5.2. (WSTL syntax) Similar to STL, given an interval I, a WSTL formula is inductively defined using the following grammar:

$$\varphi := \top \mid \mu.\omega \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \, \mathsf{U}_I \, \varphi_2$$

where $\mu.\omega$ is a predicate defined on signal $\mathbf{x}$ and its associated weight $\omega \in \mathbb{R}^{>0}$ to express the interest or the importance of the predicate. □

Since $\omega \in \mathbb{R}^{>0}$, the definition of WSTL preserves the boolean semantics of its corresponding STL formula which assumes all weight parameters are 1. However, this definition affects the quantitative semantics by allowing more contribution from more important predicates, while still capturing the robustness of satisfaction. Thus, the quantitative semantics of WSTL is defined using a real-valued function ρ_w satisfying the property:

$$\rho_w(\varphi.\boldsymbol{\omega}, \mathbf{x}, t) \geq 0 \text{ iff } (\mathbf{x}, t) \models \varphi.\boldsymbol{\omega}, \tag{5.3}$$

where $\boldsymbol{\omega} = \{\omega_1, \omega_2, \ldots, \omega_m\}, \omega_i \in \mathbb{R}^{>0}$, and m is the number of predicates in φ. The quantitative semantics for each predicate μ with a weight parameter ω in WSTL is defined as:

$$\rho_w(\mu.\omega, \mathbf{x}, t) = \omega \cdot f(\mathbf{x}(t)). \tag{5.4}$$

Similar to STL, ρ_w can be inductively defined using the same rules (4.4-4.6).

As for PSTL, we equip WSTL with scale and time parameters to obtain PW-STL, where the scalar parameter π in predicates is in the form of

$$\mu.\omega = \omega \cdot (f(\mathbf{x}) \sim \pi). \tag{5.5}$$

STL and PSTL are the special case for WSTL and PWSTL when $\boldsymbol{\omega}$ is a vector of ones.

Consider the PWSTL formula $\varphi(\pi, \tau).\boldsymbol{\omega} = \mathsf{G}_{[0,\tau]}((x > \pi).\omega)$. Applying the G rule (4.8) and the predicate semantics (5.4) and choosing $f(x) = \pi - x$,

$$\begin{aligned}
\rho_w(\mathsf{G}_{[0,\tau]}((x > \pi).\omega), x, t) &= \inf_{t \in [0,\tau]} (\omega \cdot (\pi - x(t))) \\
&= \omega \cdot \inf_{t \in [0,\tau]} (\pi - x(t)) \\
&= \omega \cdot \rho(\mathsf{G}_{[0,\tau]}(x > \pi), x, t),
\end{aligned}$$

recalling that ρ is the STL quantitative semantics.

Example 5.3. We could improve Formula (4.10) by expressing it as the WSTL formula

$$\varphi_{\texttt{sp_rpm_w}} = \varphi.\{\omega_1, \omega_2\} = \mathsf{G}((\texttt{speed} \leq 120).\omega_1) \wedge \mathsf{G}((\texttt{RPM} \leq 4500).\omega_2). \tag{5.6}$$

Further, we could change PSTL Formula (4.9) to PWSTL:

$$\varphi(\pi_{speed}, \pi_{rpm}).\{\omega_1, \omega_2\} = \mathsf{G}((\texttt{speed} \leq \pi_{speed}).\omega_1) \wedge \mathsf{G}((\texttt{RPM} \leq \pi_{rpm}).\omega_2). \tag{5.7}$$

If choosing $f_1 = \pi_{speed} - \texttt{speed}$ and $f_2 = \pi_{rpm} - \texttt{RPM}$, applying rules (4.5) and (4.8) the qualitative semantics of the WSTL formula is defined as:

$$\begin{aligned}
&\rho_w(\varphi(\pi_{speed}, \pi_{rpm}).\{\omega_1, \omega_2\}, \mathbf{x}, t) \\
&= \min(\inf_{t \in \mathbb{T}} (\omega_1 \cdot (\pi_{speed} - \texttt{speed}(t))), \inf_{t \in \mathbb{T}} (\omega_2 \cdot (\pi_{rpm} - \texttt{RPM}(t)))) \\
&= \min(\omega_1 \cdot \inf_{t \in \mathbb{T}} (\pi_{speed} - \texttt{speed}(t)), \omega_2 \cdot \inf_{t \in \mathbb{T}} (\pi_{rpm} - \texttt{RPM}(t)))
\end{aligned}$$

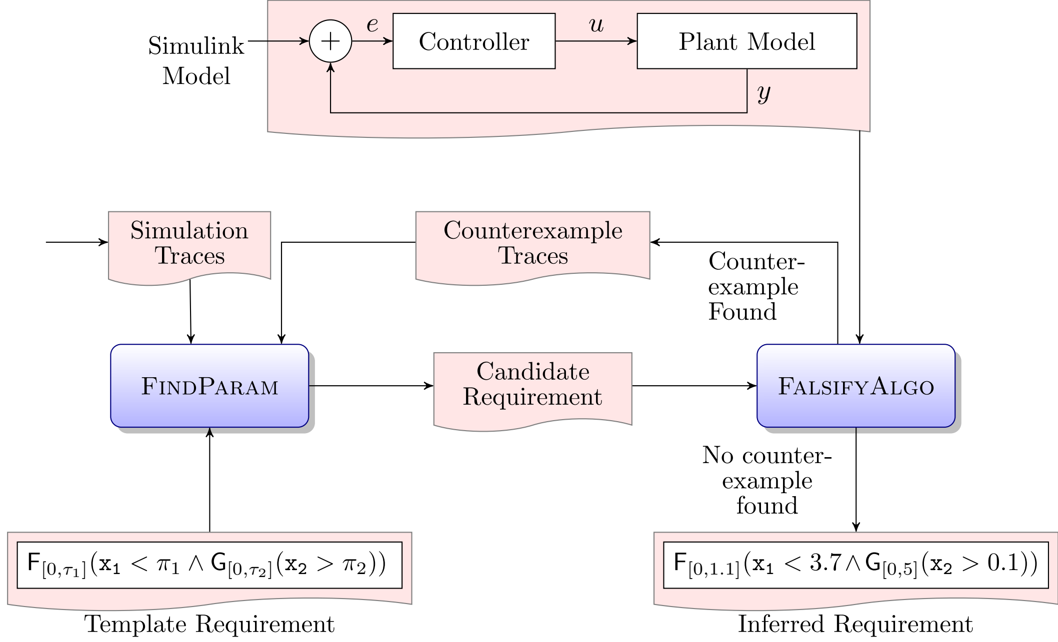

Figure 5.1: Flowchart of the requirement mining framework. This framework is an instance of a counterexample-guided inductive synthesis procedure. After given a PSTL or PWSTL template requirement formula, the synthesis engine will find an appropriate valuation for parameters in the parametric formula to form a STL or WSTL formula as a candidate requirement. The candidate requirement will be used by the falsification engine to search a counterexample to falsify the formula. If found, the counterexample trace, together with previous traces, is passed to the synthesis engine to start the next iteration.

An especially meaningful case would be choosing $\omega_1 = 1/\pi_{speed}$ and $\omega_2 = 1/\pi_{rpm}$ which eliminates the impact of different units of measure. Moreover, if $\omega_1/\omega_2 > \pi_{rpm}/\pi_{speed}$, we assign more weight to predicate $\mathtt{speed} \leq 120$, indicating that scenarios where `speed` is over the threshold are more critical.

5.3 Requirement mining framework

Figure 5.1 shows the proposed framework to mine STL or WSTL requirements from a closed-loop model. Note that the weight parameters $\boldsymbol{\omega}$ are predefined by users and are not targets for mining. This framework is an instance of a counterexample-guided inductive synthesis procedure [189]. Since STL and PSTL are a special case for

WSTL and PWSTL, the following sections give algorithms and explanations in WSTL and PWSTL. The corresponding framework for STL and PSTL simply changes a WSTL formula $\varphi.\boldsymbol{\omega}$ to an STL formula φ and replaces the quantitative semantics ρ_w to ρ defined in Formulas (5.4) and (4.1).

Given a system $\mathcal{S}$ with a set $\mathcal{U}$ of inputs, a PWSTL formula with n symbolic parameters $\varphi(\mathbf{p}).\boldsymbol{\omega}$ where $\mathbf{p} = \{p_1, \ldots, p_n\}$ and p_i could either be a scale parameter π or a time parameter τ, two key components in the framework are:

1. A falsification engine is required to search the system's input space for a counterexample that falsifies the given WSTL formula. Formally, given a formula $\varphi.\boldsymbol{\omega}$, the falsification engine generates an input $\mathbf{u}$ such that $\mathbf{x}(t) = \mathcal{S}(\mathbf{u})(t) \not\models \varphi.\boldsymbol{\omega}$, if there exists such a $\mathbf{u}$, and returns $\perp$ otherwise. We denote this functionality by FalsifyAlgo.

2. A synthesis engine is required to search the state space of the PWSTL parameters for a proper valuation. Moreover, the instantiated WSTL formula from the valuation must be satisfied by a set of recorded counterexample traces. Formally, given a set of traces $\{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ and a parametric formula $\varphi(\mathbf{p}).\boldsymbol{\omega}$, the synthesis engine finds parameters $\mathbf{p}$ such that $\forall i$, $\mathbf{x}_i \models \varphi(\mathbf{p}).\boldsymbol{\omega}$. We denote this functionality by FindParam.

As an example, consider the following natural language specification: "eventually, between time 0 and some unspecified time τ_1, the signal x_1 is less than some value π_1, and from that point, for some τ_2 seconds, the signal x_2 is greater than some value π_2". For simplicity, all predicates have the same weight, so we can express this property using PSTL as:

$$\mathbf{F}_{[0,\tau_1]}(x_1 < \pi_1 \ \wedge \ \mathbf{G}_{[0,\tau_2]}(x_2 > \pi_2)),$$

with two unspecified time parameters τ_1 and τ_2 as well as two scale parameters π_1 and π_2. The proposed mining algorithm iterates the following steps:

- FindParam synthesizes a candidate requirement as a WSTL formula from a given template requirement expressed in PWSTL and a set of simulation traces of the model. Initially, we use a random input $\mathbf{u}$ to generate the first trace.
- FalsifyAlgo tries to falsify the candidate WSTL formula using a falsification engine, such as that in S-Taliro [20] or Breach [83].
- If FalsifyAlgo finds a counterexample, add this trace to the existing set of simulation traces, go to Step 1, and start the next iteration. If no counterexample is found, the algorithm terminates and returns the inferred WSTL requirement.

As shown in Figure 5.1, the inferred STL formula would be $\mathsf{F}_{[0,1.1]}(x_1 < 3.7 \wedge \mathsf{G}_{[0,5]}(x_2 > 0.1))$ which indicates that within 1.1 seconds the signal x_1 of the system is less than 3.7, and from that point the signal x_2 is always greater than 0.1 for at least 0.5 seconds. If the designers are not satisfied with the system performance, the stored counterexamples will give them sufficient knowledge of the temporal behavior and help to debug and redesign. Next, we will detail each component of the proposed framework.

5.4 Revisiting the falsification problem

The mining framework requires us to implement a function

$$\mathbf{x} = \textsc{FalsifyAlgo}(\mathcal{S}, \varphi)$$

such that $\mathbf{x}$ is a valid output signal of a system $\mathcal{S}$ and $\mathbf{x} \not\models \varphi$. This is the falsification problem we solved using quantitative semantics in Section 4.4:

$$\text{Solve } \rho^* = \min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$$

if $\rho^* < 0$, return $\mathbf{u}^* = \arg\min_{\mathbf{u} \in \mathcal{U}} \rho(\varphi, \mathcal{S}(\mathbf{u}), 0)$, otherwise, $\mathcal{S} \models \varphi$.

Unfortunately, this is an undecidable problem for general hybrid systems, except for subclasses such as initialized RHA [108]. For the latter subclasses, the mining technique can be *complete*, i.e., absence of a counterexample means that we have found the strongest requirement. However, as discussed in Section 4.4, in general it is possible that the falsification tool may not be able to find a counterexample even if one exists.

On the bright side, a requirement mined in this fashion is still useful as it is something that FalsifyAlgo is unable to disprove even after extensive simulations, and is thus likely to be close to the actual requirement. An alternative is to use a sound verification tool that employs abstraction [99, 200]. However, these tools have not scaled to the complex control systems that we consider here.

5.5 Parameter synthesis

The general problem of parameter synthesis can be formalized as follows:

Problem 5.4. Given a system $\mathcal{S}$ with a PWSTL formula with n symbolic parameters $\varphi(p_1, \ldots, p_n)$, the objective is to find a tight valuation function v such that

$$\forall \mathbf{u} \in \mathcal{U} : \mathcal{S}(\mathbf{u}) \models \varphi(v(p_1), \ldots, v(p_n)).\boldsymbol{\omega},$$

where $\mathcal{U}$ is the input space. □

Note that $\mathcal{U}$ is an infinite set. Clearly, it is impossible to explore all input signals to find the best function v. Instead, FindParam synthesizes a set of repre-

```
v FindParam(x, φ.ω, p, δ)
 1  if ∃v s.t. x ⊨ φ(v⊤)  • A trace x, a PWSTL Formula φ.ω and parameter set p
 2    v⊤ ← v;
 3  else
 4    return ⊥;                                                • φ is unsatisfied.
 5  if ∃v s.t. x ⊭ φ(v⊤)
 6    v⊥ ← v;
 7  else
 8    return ⊤;                      • φ is trivially satisfied and v may not be tight.
 9  v ← v⊤;
10  for i = 1 to n
11    Find v_i and set v(p_i) = v_i s.t. x ⊨^i_δ φ(v);              • precision δ > 0
12  return v;
```

Figure 5.2: The FindParam algorithm for synthesizing a candidate valuation from a given template requirement in PWSTL.

sentative traces which are the recorded counterexamples found by FalsifyAlgo. For clarity, we restrict our explanation of FindParam algorithm to one trace even though in Figure 5.1 FindParam is applied to a set of traces. The generalization to a set of traces is straightforward. Thus, the problem is reduced to given a trace $\mathbf{x}$, find a valuation v for the parameters $p_1, \ldots, p_n$, of φ such that $\mathbf{x}$ satisfies $\varphi(v(p_1), \ldots, v(p_n))$, which we sometimes abbreviate in $\varphi(v)$ in the following. This problem is the dual of the falsification problem in Formula (4.11) and can be solved in a similar way:

$$\max_{v} \rho(\varphi(v).\boldsymbol{\omega}, \mathbf{x}, 0). \tag{5.8}$$

However, an important difference is that the cost function can be expressed as a closed-form expression of the decision variable v whereas Formula (4.11) is a function of $\mathbf{u}$.

The second issue is how to characterize the notion of "tight" more precisely. By tight, we mean to enforce mining of non-trivial or not overly conservative requirements. One solution is to impose an additional constraint to the parameter synthesis problem. Thus, the WSTL formula mined should be tightly satisfied by the system up to a given precision $\delta \geq 0$. Formally, we have the following definition:

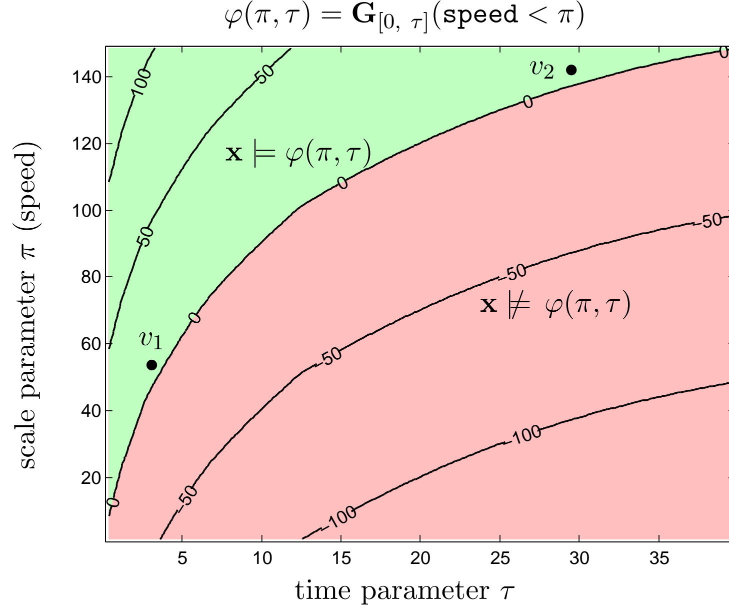

Figure 5.3: Validity domain of a simple formula for a trace $\mathbf{x}$ obtained from the automatic transmission model. The FindParam algorithm will return valuation v_1 (resp. v_2) depending on whether the time (resp. scale) parameter is optimized first. The contour lines are isolines for the satisfaction function ρ.

Definition 5.5. (**(δ-satisfaction)** The signal $\mathbf{x}$ δ-satisfies $\varphi(v).\boldsymbol{\omega}$ for p_i denoted by $\mathbf{x} \models_\delta^i \varphi(v).\boldsymbol{\omega}$ iff $\mathbf{x} \models \varphi(v).\boldsymbol{\omega}$ and there exists a valuation v' such that $|v(p_i) - v'(p_i)| \leq \delta$ and $\mathbf{x} \not\models \varphi(v').\boldsymbol{\omega}$. The signal $\mathbf{x}$ δ-satisfies $\varphi(v).\boldsymbol{\omega}$, denoted by $\mathbf{x} \models_\delta \varphi(v).\boldsymbol{\omega}$ if $\forall i$, $\mathbf{x} \models_\delta^i \varphi(v).\boldsymbol{\omega}$. □

The rationale is that, for a specification to be useful, it should not be too conservative. For instance, the requirement that “the car cannot go faster than 500 mph” is not very interesting. This means that it is not enough to find a satisfying valuation, and each parameter needs to be optimized to get δ-satisfaction. If there is more than one parameter, then the solution is not unique. In fact, all valuations that are within a distance δ from the boundary of the validity domain of φ and $\mathbf{x}$, i.e., the set of valuation v for which $\mathbf{x} \models \varphi(v)$, are valid solutions.

Example 5.6. Consider an STL formula for Figure 2.9, $\varphi(\pi,\tau) = \mathsf{G}_{[0,\ \tau]}(\texttt{speed} < \pi)$

and the scenario where the vehicle constantly accelerates at `throttle` $= 100$. The validity domain of φ is plotted on Figure 5.3. The algorithm will return different values depending on the tightness parameter δ and on whether we order the parameters as (π,τ) or (τ, π). Here, the order represents the preference in optimizing a parameter over the other when mining for a tight specification.

In [22], researchers noted that, if the formula is monotonic, then this boundary has the properties of a Pareto surface for which there are efficient computational methods, basically equivalent to a multi-dimensional binary search. Fortunately, many formulas are monotonic. For example, for the property $\mathbf{F}_{[0,\tau]}(x > \pi)$, the satisfaction value monotonically increases in the parameter τ and decreases in π. Here, we propose an algorithm for monotonic formulas that takes advantage of this property when implementing FindParam in Figure 5.2. It starts by trying to find a valuation $v_\top$ that satisfies the property and a valuation $v_\bot$ that violates it in a parameter range $\mathcal{P}$ provided by the user. If the property is monotonic, it is sufficient to check the corners of $\mathcal{P}$ for the existence of $v_\top$ and $v_\bot$. Then, each parameter i is adjusted by using a binary search initialized with $v_\top(p_i)$ and $v_\bot(p_i)$. The user can also specify the priorities among different input parameters which define the optimization order.

5.5.1 Satisfaction monotonicity

Since if PSTL formulas are monotonic the corresponding PWSTL formulas share the same property, we will use PSTL for explanation in this section. We first show that checking whether an arbitrary PSTL formula is monotonic in a given parameter is undecidable.

Theorem 5.7. The problem of checking if a PSTL formula $\varphi(\mathbf{p})$ is monotonic in a given

parameter p_i is undecidable. $\square$

Proof. First, we observe that STL is a superset of MTL. We know from [11] that the satisfiability problem for MTL is undecidable. Thus, it follows that the satisfiability problem for STL is also undecidable. This, in turn, implies undecidability of the satisfiability problem of PSTL with at most one parameter (denoted as PSTL-1-SAT). We now show that PSTL-1-SAT can be reduced to a special case of the problem of checking monotonicity of a PSTL formula.

Let $\varphi(\mathbf{p})$ be an arbitrary PSTL formula where the set of parameters $\mathbf{p}$ is the singleton set with one time parameter τ (thus, $\tau \geq 0$). Construct the formula $\psi(\mathbf{p}) \doteq (\tau = 0) \vee \varphi(\mathbf{p})$.

Consider the monotonicity query for $\psi(\mathbf{p})$ in parameter τ:

$$\forall v, v', \mathbf{x} : \; [\mathbf{x} \models \psi(v(\tau)) \;\wedge\; v(\tau) \leq v'(\tau)] \;\Rightarrow\; \mathbf{x} \models \psi(v'(\tau)).$$

Consider the specialization of this formula for the case $v(\tau) = 0$. Note that, in this case, $\psi(0) = \top$, and that $v'(\tau) \geq 0$ for all v'. Thus, the query simplifies to

$$\forall v', \mathbf{x} : \; \mathbf{x} \models \psi(v'(\tau)),$$

which is checking the validity of the PSTL formula $\psi(\tau)$.

If one needs to check monotonicity of PSTL formula φ in one parameter τ, one needs to check that the negation of $\psi(\tau)$ is unsatisfiable. Thus, the above specialization of the problem of checking the monotonicity of PSTL formulas is also undecidable, implying undecidability of the general case. $\square$

Monotonicity is closely related to the notion of polarity introduced in [22], in which syntactic deductive rules are given to decide whether a formula is monotonic based on the monotonicity of its subformulas. Thus, one way to tackle undecidability is

to first query whether the given PSTL formula belongs to the syntactic class described in [22]. Unfortunately, the syntactic rules described therein are not complete; there are monotonic PSTL formulas that do not belong to this syntactic class, for instance, formulas with intervals in which both end-points are parameterized, such as

$$\mathbf{G}_{[\tau,\tau+1]}((x \geq 3) \Rightarrow \mathbf{F}_{(0,\infty)}(x < 3)). \tag{5.9}$$

Next, we show how we can use SMT solving to query monotonicity of a formula. If the SMT solver succeeds, it tells us that the formula is monotonic and allows us to use a more efficient search in the parameter space. For instance, we were able to show that the PSTL formula represented in Formula (5.9) is monotonically decreasing in the parameter τ.

Encoding PSTL as constraints. Given a PSTL formula φ, we define the SMT encoding of φ in a fragment of first-order logic with real arithmetic and uninterpreted functions. Let $\mathcal{E}(\varphi)$ denote the encoding of φ, which we define inductively as:

- Consider a constraint $\mu \doteq g(\mathbf{x}) > \tau$, where $\mathbf{x} = (x_1, \ldots, x_n)$. We model each signal x_i as an uninterpreted function χ_i from $\mathbb{R}$ to $\mathbb{R}$. We create a new free variable t of the type `Real` (defined in an SMT solver) and replace each instance of the signal x_i in $g(\mathbf{x})$ by $\chi_i(t)$. We assume that the function g itself has a standard SMT encoding. For example, consider the formula $g(\mathbf{x}) > \tau$, where $\mathbf{x} = \{x_1, x_2\}$, and $g(\mathbf{x}) = 2 \cdot x_1 + 3 \cdot x_2$. Then $\mathcal{E}(\mu)$ is: $2 \cdot \chi_1(t) + 3 \cdot \chi_2(t) \ > \tau$.

- For boolean operations, the SMT encoding is inductively applied to the subformulas, i.e., if $\varphi = \neg\varphi_1$, then $\mathcal{E}(\varphi) = \neg\mathcal{E}(\varphi_1)$. If $\varphi = \varphi_1 \wedge \varphi_2$, then first we ensure that if $\mathcal{E}(\varphi_1)$ and $\mathcal{E}(\varphi_2)$ both have a free time-domain variable, we make it the same

variable, and then, $\mathcal{E}(\varphi) = \mathcal{E}(\varphi_1) \wedge \mathcal{E}(\varphi_2)$. Note that as a consequence, there is at most one free time-domain variable in any subformula.

- Consider $\varphi = \mathrm{H}_{(a,b)}(\varphi_1)$, where a, b are constants or parameters, and H is a unary temporal operator (i.e., $\mathbf{F}, \mathbf{G}$). There are two possibilities:

  (1) The SMT encoding $\mathcal{E}(\varphi_1)$ has one free variable t. In this case, we bound the variable t over the interval (a, b) using a quantifier that depends on the type of the temporal operator H. With $\mathbf{F}$, we use $\exists$ as the quantifier, and with $\mathbf{G}$ we use $\forall$. E.g., let $\varphi = \mathbf{F}_{(2.3,\tau)}(x > \pi)$, then $\mathcal{E}(\varphi)$ is:

$$\exists t : \quad (2.3 < t < \tau) \wedge (\chi(t) > \pi).$$

  (2) The SMT encoding $\mathcal{E}(\varphi_1)$ has no free variable. This can only happen if φ_1 is $\top$ or $\bot$, or if all variables in φ_1 are bound. In the former case, the encoding is done exactly as in case 1. In the latter case, the encoding proceeds as before, but all bound variables in the scope are additionally offset by the top-level free variable. Suppose, $\varphi = \mathbf{G}_{(0,\infty)}\mathbf{F}_{(1,2)}(x > 10)$. Then, the encoding of the inner $\mathbf{F}$-subformula has no free variable. Note how the bound variable of this formula is offset by the top-level free variable in the underlined portion in $\mathcal{E}(\varphi)$ below:

$$\forall t : [\exists u : [\underline{(t + 1 < u < t + 2)} \wedge (\chi(u) > 10)]].$$

- Consider $\varphi = \varphi_1 \mathbf{U}_{(a,b)} \varphi_2$, where a, b are constants or parameters. For simplicity, consider the case where φ_1 and φ_2 have no temporal operators, i.e., $\mathcal{E}(\varphi_1)$ and $\mathcal{E}(\varphi_2)$ both have exactly one free variable each. Let t_1 be the free variable in $\mathcal{E}(\varphi_1)$ and t_2 the free variable in $\mathcal{E}(\varphi_2)$. Then $\mathcal{E}(\varphi)$ is given by the formula:

$$\exists t_2 : [(t_2 \in (a, b)) \wedge \mathcal{E}(\varphi_2) \wedge \forall t_1 : [(t_1 \in (a, t_2)) \Rightarrow \mathcal{E}(\varphi_1))].$$

Formula	Monot.	Time
$\mathbf{G}_{(0,\infty)}(x < \pi)$	+	< 0.09
$\mathbf{G}_{[s,s+1]}(x \geq 3 \Rightarrow \mathbf{F}_{(0,\infty)} x < 3)$	–	0.1
$\mathbf{G}_{(0,100)}((x < \pi) \Rightarrow \mathbf{F}_{(0,5)}(x > \pi))$	–	< 0.09
$\mathtt{gear}_i \mathbf{U}_{(s,s+5)} \mathtt{gear}_{i+1}$	*	0.13

Table 5.1: Proving monotonicity with an SMT solver. Time is measured in seconds.

If φ_1, φ_2 contain no free variables, then t_1, t_2 are respectively used to offset all bound variables in their scope as before.

Using an SMT solver to check monotonicity. To check monotonicity, we check the satisfiability of the negation of each of the following assertions:

$$\mathcal{E}(\varphi(\tau)) \wedge (\tau > \tau') \wedge \neg\mathcal{E}(\varphi(\tau')) \text{ and } \mathcal{E}(\varphi(\tau)) \wedge (\tau < \tau') \wedge \neg\mathcal{E}(\varphi(\tau')).$$

If either of these queries is unsatisfiable, this means that satisfaction of φ is indeed monotonic in τ. If both queries are satisfiable, this means that there is an interpretation for the (uninterpreted) function representing the signal $\mathbf{x}$ and valuations for τ, τ' which demonstrates the non-monotonicity of φ. We conclude by presenting a small sample of formulas for which we could prove or disprove monotonicity using the Z3 SMT solver [80] in Table 5.1. The symbols +, –, and * represent monotonically increasing, decreasing, and non-monotonic formulas, respectively.

5.6 Case studies

First, we compare the performance of STL-based mining frameworks using the falsification engine from S-Taliro and Breach. We also show the performance of the

Template	S-Taliro-based mining				
	Parameter values	Fals.	Synth.	#Sim.	Rob./$\mathbf{x}$
$\varphi_{\texttt{sp_rpm}}(\pi_1, \pi_2)$	(155 mph, 4858 rpm)	55	12	255	0.004
$\varphi_{\texttt{rpm100}}(\pi, \tau)$	(3278.3 rpm, 49.91 sec)	6422	26.5	9519	0.327
$\varphi_{\texttt{rpm100}}(\tau, \pi)$	(4997 rpm, 12.20 sec)	8554	53.8	18284	0.149
$\varphi_{\texttt{stay}}(\pi)$	1.79 sec	18886	0.868	130	147.2

Template	Breach-based mining				
	Parameter values	Fals.	Synth.	#Sim.	Rob./$\mathbf{x}$
$\varphi_{\texttt{sp_rpm}}(\pi_1, \pi_2)$	(155 mph, 4858 rpm)	197.2	23.1	496	0.043
$\varphi_{\texttt{rpm100}}(\pi, \tau)$	(3273 rpm, 49.92 sec)	267.7	10.51	709	0.026
$\varphi_{\texttt{rpm100}}(\tau, \pi)$	(4997 rpm, 12.20 sec)	147.8	5.188	411	0.021
$\varphi_{\texttt{stay}}(\pi)$	0.102 sec	430.9	2.157	1015	0.032

Table 5.2: Results on mining requirements for the automatic transmission control model. We compare runs of requirement mining algorithm using either S-Taliro or Breach as falsifiers. In each case and for each template formula, we give the parameters valuations found, the time spent in falsification, and in parameter synthesis, the number of simulations, and the averaged time spent computing the quantitative satisfaction of the formula by one trace. Time is measured in seconds.

parameter synthesis algorithm implemented in BREACH.

5.6.1 Automatic transmission model

For the model described in Section 2.3.2.1, we tested the following different STL (or transformed equivalent MTL for S-TALIRO) and PSTL requirements:

1. Requirement $\varphi_{\texttt{sp_rpm}}(\pi_1, \pi_2)$ specifying that always the `speed` of the system is

always below π_1 and the RPM is below π_2 :

$$\mathbf{G}\left(\ (\mathtt{speed} < \pi_1) \wedge (\mathtt{RPM} < \pi_2)\ \right).$$

2. Requirement $\varphi_{\mathtt{rpm100}}(\tau, \pi)$ specifying that the vehicle cannot reach the speed of 100 mph in τ seconds with RPM always below π:

$$\neg(\mathbf{F}_{[0,\tau]}(\mathtt{speed} > 100) \wedge \mathbf{G}(\mathtt{RPM} < \pi)).$$

3. Requirement $\varphi_{\mathtt{stay}}(\tau)$ specifying that whenever the system shifts to gear 2, it dwells in gear 2 for at least τ seconds:

$$\mathbf{G}\left(\left(gear \neq 2 \wedge \mathbf{F}_{[0,\varepsilon]}\mathtt{gear} = 2\right) \Rightarrow \mathbf{G}_{[\varepsilon,\tau]}\mathtt{gear} = 2\right).$$

Here, the left-hand-side of the implication captures the event of the transition from gear 2 to another gear. The operator $\mathbf{F}_{[0,\varepsilon]}$ here is an STL substitute for a next-time operator. With dense time semantics, ε should be an infinitesimal quantity but, in practice, we use a value close to the simulation time-step.

The above requirements have strong correlation with the quality of the controller. The first is a safety requirement characterizing the operating region for the engine parameters speed and RPM. The second is a measure of the performance of the closed loop system. By mining values for τ, we can determine how fast the vehicle can reach a certain speed, while by mining π we find the lowest RPM needed to reach this speed. The third requirement encodes undesirable transient shifting of gears. Rapid shifting causes abrupt output torque changes leading to a jerky ride.

Results about the mined specifications are given in Table 5.2. We used the Z3 SMT solver [80] to show that all of the requirements were monotonic. As expected, the FindParam algorithm takes only a fraction of the total time in the entire mining

process. For the second template, we try two possible orderings for the parameters. By prioritizing the time parameter τ, we obtain the δ-tight requirement that the vehicle cannot reach 100 mph in less than 12.2 seconds (we set δ to 0.1). As the requirement mined is δ-tight, it means that we find a trace for which the vehicle reaches 100 mph in 12.3 seconds. Similarly, by prioritizing the scale parameter π, we get the result that the vehicle could reach 100 mph in 50 seconds while keeping `RPM` below 3278 ($\delta = 5$ in that case). For the third requirement, we find that the transmission controller could trigger a transient shift in as short as 0.112 seconds. This corresponds to the up-shifting sequence 1-2-3.

The comparison between S-TALIRO and BREACH falsification engines in Table 5.2 shows that the requirement mining has better performance with the BREACH engine. The result is similar to the one we conclude in Chapter 4. The BREACH engine is able to find stronger requirements using fewer of simulations and less computational time. Also, we have the following observations:

- The space of input signals needs to be parameterized with a sensible number of signal-parameters. If too many parameters are used, the search space is too big and falsification becomes difficult. For instance, the short transient shifting of $\varphi_{\texttt{stay}}$ is found by introducing a signal-parameter controlling the time of initial acceleration and by preventing acceleration and braking at the same time. We remark that the flexibility of BREACH to enforce such constraints over the input signal space is a key reason for its better performance, and a fair comparison would be possible only after repeating these steps for S-TALIRO.
- Requirements involving discrete modes are challenging, because they induce "flat" quantitative satisfaction functions that are challenging to optimizers and thus have

limited value in guiding the falsifier. This is related to the problem of finding a good metric between discrete states in hybrid systems. This was particularly an issue when mining the $\varphi_{\texttt{stay}}$ requirement. We are able to tune our falsifier by turning off its local optimization phase and using uniform random sampling which led us to obtaining a tighter requirement than with S-Taliro.

- While both falsifiers are expected to exhibit run-times linear in the size of the traces and the formula [85, 91], in some cases, Breach runs faster. In particular, S-Taliro is more sensitive to parameter priorities. For the same template $\varphi_{\texttt{rpm100}}$, depending on which parameter τ or π is prioritized, S-Taliro performs differently. This can be explained by the fact that τ affects the horizon of the temporal operator **F**. We conjecture that the difference in run-times and mined parameter values for the $\varphi_{\texttt{stay}}$ template is due to our inability express signal parameterization in S-Taliro.

Next, we compare the WSTL- and PWSTL-based requirement mining results. Unfortunately, S-Taliro does not support these two advanced temporal logic. Breach, on the other hand, uses predicates to express property and is very flexible. Thus, we extend WSTL and PWSTL in Breach, and the following experiments are carried out only on Breach. WSTL and PWSTL embrace users' knowledge to set up the correct weights. However, if variables of weighted predicates are correlated, applying weight does not improve much on the results. Next, we examine two requirements in detail:

1. Requirement $\varphi_{\texttt{sp_rpm_w}}(\pi_1, \pi_2)$ having the same same specification as $\varphi_{\texttt{sp_rpm}}$ with $\omega_1 = 50$ for the `speed` predicate and $\omega_2 = 0.01$ for the `RPM` predicate:

$$\mathbf{G}\left(\ (\texttt{speed} < \pi_1).\omega_1 \wedge (\texttt{RPM} < \pi_2).\omega_2\ \right).$$

Template	STL-based mining Parameter values	Fals.	Synth.	#Sim.	Rob./$\mathbf{x}$
$\varphi_{\texttt{sp_rpm}}(\pi_1, \pi_2)$	(155 mph, 4858 rpm)	197.2	23.1	496	0.043
$\varphi_{\texttt{gs}}(\tau)$	(5.40 sec)	348.1	8.15	682	0.029

Template	WSTL-based mining ω value	Parameter values	Fals.	Synth.	#Sim.	Rob./$\mathbf{x}$
$\varphi_{\texttt{sp_rpm_w}}(\pi_1, \pi_2)$	$(50, 0.01)$	(155 mph, 4858 rpm)	172.6	26.3	468	0.043
$\varphi_{\texttt{gs_w}}(\tau)$	(1000)	(5.40 sec)	215.6	15.31	426	0.030

Table 5.3: Results on STL- and WSTL-based mining requirements for the automatic transmission control model using Breach. In each case and for each template formula, we give the chosen weight vector, the parameters valuations found, the time spent in falsification and in parameter synthesis, the number of simulations, and the averaged time spent computing the quantitative satisfaction of the formula by one trace. Time is measured in seconds.

2. Requirement $\varphi_{\texttt{gs}}$ specifying that within the first τ seconds, the vehicle is unable to shift to gear 4 and, if the speed reaches 70 mph in 2 seconds, it is able to decrease and stay below 30 mph within 10 seconds.

$$(\neg \mathsf{F}_{[0,7]}\ (\texttt{gear} = 4)) \wedge ((\mathsf{F}_{[0,2]}\ \texttt{speed} > 70) \Rightarrow (\mathsf{G}_{[10,\infty)}\ \texttt{speed} < 30).$$

The corresponding WSTL $\varphi_{\texttt{gs_w}}$ is

$$(\neg \mathsf{F}_{[0,7]}\ (\texttt{gear} = 4).\omega) \wedge ((\mathsf{F}_{[0,2]}\ \texttt{speed} > 70) \Rightarrow (\mathsf{G}_{[10,\infty)}\ \texttt{speed} < 30),$$

with $\omega = 1000$. This requirement only applies to the special case where the vehicle speeds up for the first several seconds and then brakes for the rest of the time to simulate an emergency situation. Thus, the inputs of the model are the initial `throttle` position, with the range in [0,100]%, and the starting time to brake

with the range in [0,20] seconds.

Results given in Table 5.3 show the benefits of using WSTL. For $\varphi_{\texttt{sp_rpm_w}}$, the advantage is not obvious, because speed and RPM are correlated. Maximizing speed is indirectly maximizing RPM, although the gear shifting logic limits the maximum RPM based on speed and gear value. However, the use of weights still saves over 20 seconds from the falsification engine.

For weakly-related constraints, especially for non-convex problems, the STL-based requirement mining framework can only leverage the priority to choose better parameters from the validity domain. However, WSTL formulas directly affect the calculation of the robustness value through weights. Consider requirements $\varphi_{\texttt{gs}}$ and $\varphi_{\texttt{gs_w}}$ with $\omega = 1000$. Assuming, in the mining process, that the synthesis engine gives the valuation $v = \{\tau = 7\}$, the robustness satisfaction value of $\varphi_{\texttt{gs}}$ at 100 sample points throughout the input domain is shown in Figure 5.4 and that for $\varphi_{\texttt{gs_w}}$ is shown in Figure 5.5. The first observation is that the robustness surface is much smoother for $\varphi_{\texttt{gs_w}}$ than for $\varphi_{\texttt{gs}}$. The smoother surface helps to speed up the falsification engine to find counterexamples. Also, since this requirement contains two conjunctive predicates, falsifying any would result in a negative robustness value. However, if paying no attention to the predicate related to the discrete gear mode changes, the "flat" quantitative will make the falsification engine concentrate on the other predicate related to the continuous speed value. To falsify the second predicate, speed has to reach 70 mph, which normally indicates that the vehicle is in a high gear position. Thus, giving weight to the discrete predicate will indirectly help the falsification engine look in the right direction. The results confirm this observation. Falsifying $\varphi_{\texttt{gs_w}}$ uses about half as much time as $\varphi_{\texttt{gs}}$ and fewer simulations. This is the main reason for the better performance of the

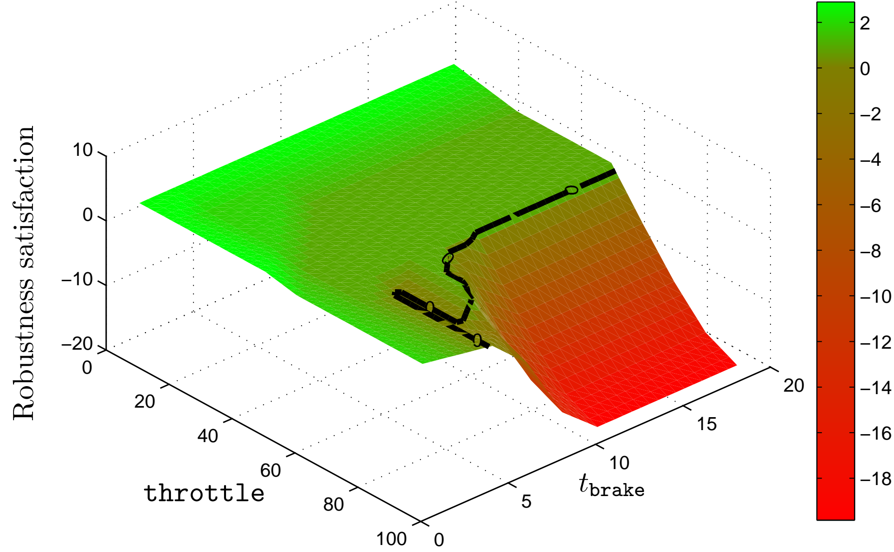

Figure 5.4: The robustness satisfaction surface of PSTL $\varphi_{\texttt{gs}}$ with $v(\tau) = 7$ consists of 100 sample points from the input domain. The inputs are the position of throttle `throttle` and the initial time to start to brake $t_{\texttt{brake}}$ with the range of `throttle` in [0,100]% and [0,20] seconds for $t_{\texttt{brake}}$.

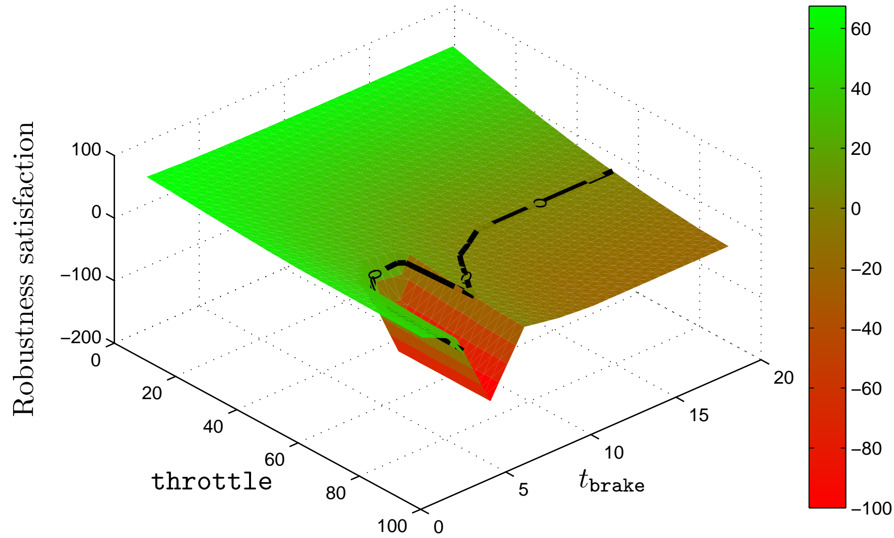

Figure 5.5: The robustness satisfaction surface of PWSTL $\varphi_{\texttt{gs_w}}$ with $v(\tau) = 7$ and $\omega = 1000$ consists of 100 sample points from the input domain. The inputs are the position of throttle `throttle` and the initial time to start to brake t with the range of `throttle` in [0,100]% and [0,20] seconds for t.

WSTL-based framework. However, we notice that the synthesis engine spends more time to find parameter valuations. Since both formulas are monotonic, the overhead is quite low. Also, introducing weight does not affect the computation of robustness values in BREACH.

5.6.2 Industrial diesel engine model

Next, we consider an industrial-scale, closed-loop Simulink model of an experimental airpath controller for a diesel engine. This model has more than 4000 Simulink blocks such as data store memories, integrators, 2D-lookup tables, functional blocks with arbitrary Matlab functions, S-Function blocks, and blocks that induce switching behaviors such as level-crossing detectors and saturation blocks. The models take two signals as inputs: the fuel injection rate and the engine speed. The output signal is the intake manifold pressure denoted by x. For proprietary reasons, we suppress the mined values of the parameters and the time-domain constants from our requirements. We replace the time-domain constants by symbols such as c_1 and c_2.

We find from the control designers that characterizing the overshoot behavior is important for the signal under consideration. The inputs to the closed-loop model are a step function of the fuel injection rate input at time c_1, and a constant value for the engine speed input. The first requirement is:

$$\varphi_{\texttt{overshoot}}(\pi) = \mathbf{G}_{(c_1,\infty)}(\texttt{x} < \pi).$$

This template characterizes the requirement that the signal $\mathbf{x}$ never exceeds π during the time interval (c_1, ∞), i.e., it finds the maximum peak value π of the step response. Our mining algorithm obtained seven intermediate candidate requirements that are falsified by S-TALIRO, until we found a requirement that it could not falsify in its 8th iteration.

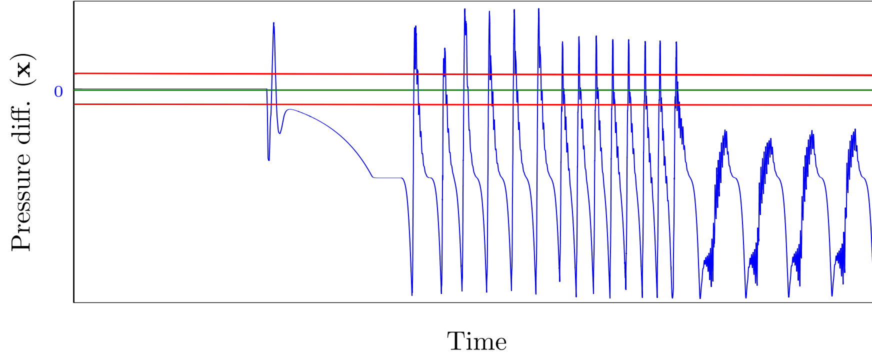

Figure 5.6: The simulation trace (in blue) for the signal $\mathbf{x}$ denoting the difference between the intake manifold pressure and its reference value found when mining $\varphi_{\texttt{settling_time}}(\tau, \pi)$ displays unstable behavior. The maximum error threshold that we expected to mine is depicted in red. The ideal $\mathbf{x}$ signal is in green. The values along the axes have been suppressed for proprietary reasons. We remark that the actual values are irrelevant and the intention is to show an oscillating behavior arising from a real bug in the design.

The total number of simulations is 7000 over a period of 13 hours.

Next, we choose to mine the settling behavior of the signal. The settling time is the time after which the amplitude of a signal is always within a small range from its calculated ideal reference value. We wish to mine both the range and how fast the signal settles. Such a template requirement is given by the following PSTL formula:

$$\varphi_{\texttt{settling_time}}(\tau, \pi) = \mathbf{G}_{[\tau,\infty)}(|\mathrm{x}| < \pi).$$

It specifies that the absolute value of $\mathtt{x}$ is always less than π starting from the time τ to the end of the simulation. The smaller the settling time and the error, the more stable the system is. We find out from the control designer that a smaller settling time needs to be prioritized over the range (as long as the range lies within 10% of the signal amplitude), so we prioritize minimizing τ over minimizing π.

After four iterations, the procedure stops as the inferred value for τ is very close to the end of the simulation trace, but the range is still larger than the tolerance.

The implication here is that the algorithm pushed the falsifier to finding a behavior in the model that exhibits hunting behavior, or oscillations of magnitude exceeding the tolerance. This output signal is shown in Figure 5.6.This behavior is unexpected; discussions with the designers revealed that it was a real bug. Investigating further, we trace the root-cause to an incorrect value in a lookup table; such lookup tables are commonly used to speed up the computation time by storing pre-computed values approximating the control law.

This experiment demonstrates the use of requirement mining as an advanced, guided debugging strategy. Instead of verifying correctness with a concrete formal requirement, the process of trying to infer what requirement a model must satisfy can reveal erroneous behaviors that could be otherwise missed. In the course of our experiments, we encounter other suspicious (for instance Zeno-like) behaviors, which we suspect to be either an error in the model, or an improper tuning of the numerical solver leading to discontinuities in the dynamics.

5.7 Conclusion

This chapter proposed a scalable requirement mining framework that is able to mine requirements from a closed-loop model and express them in a temporal logic. The major components of this framework are a falsification engine and a synthesis engine.

STL and PSTL are particularly well-suited for this framework. However, we observed a drawback in STL and PSTL: they do not distinguish the different contributions associated with each component constraint in a complex requirement nor do they consider the effect of different units of measure used for various signals. Thus, we propose WSTL and PWSTL to improve this framework.

We observed that the satisfaction monotonicity of PSTL or PWSTL plays an important role to find a tight parameter value for the synthesis engine. If monotonicity holds, we can get exponential savings when searching over the parameter space, by using methods like binary search. Though syntactic rules for polarity of a PSTL property identified in previous work [22] ensure satisfaction of monotonicity, these rules are not complete. Hence, we provide a general way of reasoning about monotonicity of arbitrary PSTL properties using an SMT solver.

Finally, we demonstrated the application of this mining framework to two non-trivial models, one of which comes from a real industrial study. The framework is able not only to generate high-level requirements, but also to serve as an important bug finding tool.

Chapter 6

Summary and Future Research

6.1 Summary

In this thesis, we focused on verification and validation methods for improving the design efficiency and safety of hybrid systems. In Chapter 2, we gave a brief review of the dynamics of hybrid systems and presented the major modeling methods. The fundamental characteristic of hybrid systems is the exhibition of both discrete control logic and physical continuous dynamics. One straightforward modeling approach is to discretize continuous dynamics and analyze the resulting full discrete systems, such as via NCFSMs or ECA rules. In Chapter 3, we described important contributions of using symbolic techniques to verify these two types of discrete systems. For an NCFSM model, we further introduced an application that uses equivalence verification to generate a test suite for fault-based testing. For ECA rules, we tackled two critical problems to verify termination and confluence properties. In particular, we proposed a fully symbolic algorithm to verify the confluence property and provided the first practical experimental results in the field.

Discretization gives us a chance to analyze an infinite hybrid system using

well-studied methods, such as model checking techniques. However, it largely ignores the inherent continuous dynamics and might cause critical errors when analyzing time-sensitive systems. HA were then proposed to faithfully model hybrid systems and represent continuous dynamics as flow activities. However, the weak scalability and the lack of accommodation for hierarchical composition designs of HA encourage researchers to model a hybrid system simply as an input-output signal mapping black box based on simulation results. This modeling method is widely adopted by industry due to its great scalability and the trend of toward MBD design process. In Chapter 4, we used a thermostat system to demonstrate both modeling methods. Although HA offer a solid mathematical foundation, most verification problems are undecidable, even for the basic reachability problem under severe limitations. However, modeling a hybrid system as signals, together with advanced temporal logics, such as STL or MTL, provide a scalable solution to validate the system instead of verifying it. The validation techniques use boolean semantics to reason about whether the system satisfies a set of temporal logic formulas that express properties of interest. Moreover, the quantitative semantics of STL or MTL offers a better way to measure the robustness or tolerance of the system with respect to these formulas. One direct application of the quantitative semantics is a falsification engine that uses this semantics to search stimulating input signals for undesirable system behaviors as counterexamples for designers to debug the design system.

Requirement defects are considered as pitfalls for hybrid system designs. Inadequate requirements cause a formidable challenge to the adoption of formal validation approaches in an industrial setting. One of the major contributions of this thesis, the proposed general framework in Chapter 5, is able to bridge the gap for high-level requirement defects. It is based on the counterexample-guided refinement procedure

which was proposed for software verification [72] and mines requirements in a temporal logic formula from a closed-loop model. We further improved the performance of this mining framework by introducing weighted temporal logics: WSTL and PWSTL. Also, we observed that monotonicity improves the performance of the synthesis engine. We proposed to formulate the query for monotonicity in a fragment of first-order logic with quantifiers, real arithmetic, and uninterpreted functions so that we can then use an SMT solver to check the monotonicity of an STL or WSTL formula. We used two non-trivial models to demonstrate the mining results, one of which is from a real industrial model. This framework has the following two applications: mined requirements can be used to validate future modifications of the model and they can be used to enhance understanding of legacy models; the framework can also guide the process of bug-finding through simulations.

6.2 Future research

As shown in this dissertation, modeling, analyzing, validating, and verifying hybrid systems are extraordinarily challenging research areas. Many researchers have devoted themselves to these research areas for the past few decades and have achieved great success. However, current approaches still have many limitations and restrictions. In this section, we suggest some preliminary future research directions to extend the work of this thesis.

- **Model-based testing using symbolic model checkers.** In Section 3.3.7, we have successfully shown that symbolic equivalence verification is able to generate a test suite that guarantees to kill all first-order mutants. This test suite can then be used to test implementations. This technique is called model-based testing [96].

This work can be extended to automating graphical user interface (GUI) testing. Adequate GUI testing enhances the safety, robustness, and usability of the whole system. GUI testing is also resource-intensive. GUI test cases require testers to follow some complex sequences of GUI events in order to test certain functionality. Also, even with the help of advanced automated test scripts, the regression test for GUIs is a major problem since the GUI may change significantly during development. Fortunately, GUIs can be modeled using FSM-based formalisms [34, 55]. Symbolic model checkers can then be applied to traverse the state space and generate test cases with great coverage of the graph or to automatically generate useful counterexamples against temporal logic requirements for critical paths, and they can be used for fault-based testing as well.

- **Applying abstraction to improve the scalability of verification algorithms.** Symbolic techniques help to relieve the stress of the state space explosion problem. However, for many applications, especially when the structure of the system is irregular, symbolic techniques can only help to a certain extent. On the other hand, abstraction is an active research area to improve scalability. Here are some successful examples: the counterexample-guided abstraction refinement framework [73] can analyze larger models; predicate abstraction is another active research area. Its application includes two famous model checkers for C programs, SLAM [28] and Blast [37]. This technology can certainly be used to verify discrete systems. For instance, when analyzing ECA rules in Section 3.5, the condition of an ECA rule normally consists of predicates in the form: as some variable is below or above a certain threshold. If we can divide the domain of the variable into disjunctive regions based on these predicates, mapping a larger domain into a

smaller one will significantly reduce the state space analyzed. However, considerations such as whether the abstraction is sound need further research. Also, if the initial abstraction is unsound, the counterexample guided refinement framework may still be able to be applied to refine the abstraction.

- **Requirement management system.** The requirement mining framework proposed provides a solution to mine formal requirements from a closed-loop model. In order to thoroughly resolve requirement defects, a requirement management system is still in demand. From our experience, the requirement management system should have the following features: first, the system should integrate a version control system; second, the system should be able to automatically mine requirements, to automatically use these requirements to validate future versions of the design, and to report the results; third, the system should be capable of checking the completeness and consistency of requirements, since all requirements are represented in a formal format; last but not least, a good GUI design is definitely of significant importance. Since the mined requirements are in the form of STL or WSTL formulas, design engineers, without knowledge of verification or temporal logic, may have a difficult time understanding these formulas. Sometimes, writing correct PSTL or PWSTL formulas is tricky even for experienced people. Thus, if the management system is able to automatically form the correct temporal logic formulas from lower-level language or even plain English, it will motivate design engineers to use this tool in their daily work.

- **Improving the efficiency of the synthesis and falsification engines.** The synthesis and falsification engines are the two major components of the requirement mining framework. The performance of the framework depends on both of

them. If the synthesis engine cannot find tight bound parameters, the resulting STL or WSTL formulas are either trivial or extremely difficult to falsify. On the other hand, if the falsification engine is not able to falsify reasonable formulas, the whole mining process aborts too early to generate requirements closer to real system behaviors. Also, if monotonicity holds for PSTL or PWSTL formulas, the synthesis engine is able to find relatively tight bounds efficiently. However, if the property does not hold, improving the accuracy of synthesis results is worth further research. For the falsification engine, current approaches rely heavily on convex optimization or stochastic optimization techniques. However, many systems are non-convex and the random nature of stochastic optimization sometimes frustrates design engineers. We propose searching for and developing better optimization algorithms to improve the performance of the falsification engine.

Bibliography

[1] AAQSC. AS9100, Quality Systems - Aerospace - Model for Quality Assurance in Design, Development, Production, Installation and Servicing, 2001.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 2011.

[4] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 59–68. ACM Press, 1992.

[5] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

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

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

[8] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Berlin / Heidelberg, 1993.

[9] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335. Springer-Verlag New York, Inc., 1990.

[10] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, Apr. 1994.

[11] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, Jan. 1996.

[12] R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings on Real-Time Systems Symposium*, pages 2 –11, dec 1993.

[13] R. Alur and T. A. Henzinger. A really temporal logic. In *FOCS*, pages 164–169, 1989.

[14] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, UK, 1992. Springer-Verlag.

[15] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *ACM SIGPLAN Notices*, volume 40, page 98109, 2005.

[16] G. Ammons, R. Bodk, and J. R. Larus. Mining specifications. In *ACM Sigplan Notices*, volume 37, page 416, 2002.

[17] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times ba tool for modelling and implementation of embedded systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464. Springer, 2002.

[18] W. Ang and Y. Park. *Ordinary Differential Equations: Methods and Applications.* Universal Publishers, 2008.

[19] Y. S. R. Annapureddy and G. E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *Proceedings of the 36th Annual Conference of IEEE Industrial Electronics*, pages 91–96, 2010.

[20] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *Proc. of TACAS*, pages 254–257, 2011.

[21] A. Anta and P. Tabuada. Self-triggered stabilization of homogeneous control systems. In *American Control Conference, 2008*, pages 4129–4134. IEEE, 2008.

[22] E. Asarin, A. Donzé, O. Maler, and D. Nickovic. Parametric identification of temporal properties. In *RV*, pages 147–160, 2011.

[23] J. C. Augusto and C. D. Nugent. A new architecture for smart homes based on ADB and temporal reasoning. In *Toward a Human-Friendly Assistive Environment*, volume 14, pages 106–113, 2004.

[24] A. Aziz, S. Taşiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *Proc. DAC*, pages 283–288. ACM Press, 1994.

[25] R. Baheti and H. Gill. Cyber-physical systems. *The Impact of Control Technology*, pages 161–166, 2011.

[26] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[27] J. Baillieul and J. Willems. *Mathematical control theory.* SPRINGER VERLAG GMBH, 1999.

[28] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.

[29] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, volume 985 of *LNCS*, pages 163–181. Springer, 1995.

[30] E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM Transactions on Database Systems*, 25(3):269–332, Sept. 2000.

[31] P. I. Barton. Modeling, simulation and sensitivity analysis of hybrid systems. In *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on*, pages 117–122. IEEE, 2000.

[32] BEEM models. `http://anna.fi.muni.cz/models/cgi/models.cgi`.

[33] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT 2004)*, LNCS 3185, pages 200–236, 2004.

[34] F. Belli. Finite state testing and analysis of graphical user interfaces. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 34–43. IEEE, 2001.

[35] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAALa tool suite for automatic verification of real-time systems.* Springer, 1996.

[36] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, C. Weise, and W. Yi. New generation of UPPAAL. In *Proc. Int. Workshop on Software Tools for Technology Transfer*, pages 43–52, June 1998.

[37] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[38] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.

[39] G. Booch. *The Unified Modeling Language User Guide, 2/E.* Pearson Education, 2005.

[40] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip-synchronisation protocol using uppaal. *Formal Aspects of Computing*, 10(5-6):550–575, 1998.

[41] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Computer Aided Verification*, pages 546–550. Springer, 1998.

[42] M. Bozga, H. Jianmin, O. Maler, and S. Yovine. Verification of asynchronous circuits using timed automata. *Electronic Notes in Theoretical Computer Science*, 65(6):47–59, 2002.

[43] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Computer Aided Verification*, pages 179–190. Springer, 1997.

[44] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[45] M. Branicky. *Handbook of Networked and Embedded Control Systems*. Control Engineering. Birkhuser Boston, 2005.

[46] M. S. Branicky, M. M. Curtiss, J. Levine, and S. Morgan. Sampling-based planning, control and verification of hybrid systems. *IEE Proceedings-Control Theory and Applications*, 153(5):575–590, 2006.

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

[48] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 535–541, 1995.

[49] J. Burch, E. M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. *Computer Science Department*, page 435, 1991.

[50] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. CAD of Integr. Circ. and Syst.*, 13(4):401–424, Apr. 1994.

[51] D. B. Camarillo, T. M. Krummel, J. K. Salisbury Jr, et al. Robotic technology in surgery: past, present, and future. *American Journal of Surgery*, 188(4A Suppl):2S–15S, 2004.

[52] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[53] X. Chen, E. Abrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, 2013.

[54] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193 –200, sep 1994.

[55] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. L. Jones. Model-based testing of gui-driven applications. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 203–214. Springer, 2009.

[56] E.-H. Choi, T. Tsuchiya, and T. Kikuno. Model checking active database rules under various rule processing strategies. *IPSJ Digital Courier*, 2:826–839, 2006.

[57] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. In *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 78–97. Springer, 2003.

[58] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.

[59] G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In *Proc. ICATPN*, LNCS 4546, pages 83–103. Springer, 2007.

[60] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4–25, 2006.

[61] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[62] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE Int. Computer Performance and Dependability Symp. (IPDS'96)*, page 60. IEEE Comp. Soc. Press, 1996.

[63] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD*, LNCS 2517, pages 256–273. Springer, 2002.

[64] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. 4th FMCAD*, pages 256–273, Nov. 2002.

[65] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. CAV*, LNCS 1633, pages 495–499. Springer, 1999.

[66] E. Clarke, A. Donzé, and A. Legay. On simulation-based probabilistic model checking of mixed-analog circuits. *Formal Methods in System Design*, 36(2):97–113, 2010.

[67] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency Reflections and Perspectives*, pages 124–175. Springer, 1994.

[68] E. M. Clarke and I. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Springer, 1989.

[69] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer, 1981.

[70] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, pages 147–158. Springer, 1998.

[71] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[72] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[73] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, Sept. 2003.

[74] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[75] S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE Transactions on Knowledge and Data Engineering*, 15:257–270, 2003.

[76] D. J. Cook, M. Youngblood, E. O. Heierman III, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. Mavhome: An agent-based smart home. In *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 521–524. IEEE, 2003.

[77] CORDIS. European commission CORDIS, seventh framework programme FP7. http://cordis.europa.eu/fp7/home_en.html.

[78] J. Cortadella. Combining structural and symbolic methods for the verification of concurrent systems. In *Proc. of the International Conference on Application of Concurrency to System Design*, pages 2–7, Mar. 1998.

[79] L. De Alfaro, M. Faella, and M. Stoelinga. Linear and branching metrics for quantitative transition systems. In *Automata, Languages and Programming*, pages 97–109. Springer, 2004.

[80] L. De Moura and N. Bjrner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, page 337340, 2008.

[81] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. erics: a tool for verifying timed automata and estelle specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 278–283. Springer, 2003.

[82] P. Derler, E. Lee, and A.-S. Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[83] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, pages 167–170, 2010.

[84] A. Donzé, T. Ferrère, and O. Maler. Efficient robust monitoring for stl. In *Computer Aided Verification*, 2013.

[85] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *FORMATS*, pages 92–106, 2010.

[86] R. G. Dromey. Making real progress with the requirements defects problem. *IDEA GROUP PUBLISHING*, page 90, 2006.

[87] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized büchi automata. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 76–83. IEEE, 2004.

[88] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):3545, 2007.

[89] J. Esparza, S. Melzer, and J. Sifakis. Verification of safety properties using integer programming: Beyond the state equation, 1997.

[90] Event-B examples. `http://wiki.event-b.org/index.php/Event-B_Examples`.

[91] G. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using s-taliro. In *Proceedings of the American Control Conference*, 2012.

[92] G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

[93] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using s-taliro. In *American Control Conference (ACC), 2012*, pages 3567–3572. IEEE, 2012.

[94] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 152–163. ACM, 2001.

[95] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[96] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[97] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *Journal of Guidance, Control, and Dynamics*, 25(1):116–129, 2002.

[98] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, Jan. 1993.

[99] F. G., L. C., and D. A. Spaceex: Scalable verification of hybrid control systems. In *Proc. of Computer-Aided Verification*, 2011.

[100] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, page 5160, 2008.

[101] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[102] T. V. Group". VIS: A system for verification and synthesis. In *Proc. CAV*, LNCS 1102, pages 428–432. Springer, July 1996.

[103] Guam. Official guam crash site center - korean air flight 801. http://ns.gov.gu/guam/indexmain.html.

[104] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Computing unique input/output sequences using genetic algorithms. In *Proc. FATES*, pages 169–184, 2004.

[105] A. Gupta. Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pages 5–92. Springer, 1993.

[106] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[107] T. Henzinger. The theory of hybrid automata. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 278 –292, jul 1996.

[108] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[109] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 394 –406, jun 1992.

[110] T. A. Henzinger. *The theory of hybrid automata*. Springer, 2000.

[111] T. A. Henzinger, P.-H. Ho, and H. Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[112] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *Automatic Control, IEEE Transactions on*, 43(4):540–554, 1998.

[113] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[114] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. CAV*, volume 1855 of *LNCS*, pages 20–35. Springer, 2000.

[115] R. Hierons. Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proc., Software Engineering*, 144(5):291–295, 1997.

[116] R. M. Hierons. Checking states and transitions of a set of communicating finite state machines. *Microprocessors and Microsystems*, 24:443–452, 2000.

[117] G. Holzmann and D. Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211. Chapman & Hall, Oct. 1994.

[118] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[119] J. E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation, 3/E*. Pearson Education India, 2008.

[120] R. T. Howard and T. C. Bryan. Dart avgs flight results. In *Defense and Security Symposium*, pages 65550L–65550L. International Society for Optics and Photonics, 2007.

[121] IEC. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[122] ISO. ISO 26262-10, Road vehicles Functional safety Part 10 Guideline on ISO 26262, 2012.

[123] D. Jackson, M. Thomas, L. I. Millett, et al. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[124] T. Javed, M. e. Maqsood, and Q. S. Durrani. A study to investigate the impact of requirements instability on software defects. *SIGSOFT Softw. Eng. Notes*, 29(3):1–7, May 2004.

[125] D. Jianmin, Q. Huan, and L. Yunfeng. Study of hybrid system modeling language using xml. In *IEEE International Conference on Information Reuse and Integration*, pages 393 – 397, oct. 2003.

[126] X. Jin, G. Ciardo, T.-H. Kim, and Y. Zhao. Symbolic verification and test generation for a network of communicating FSMs. In T. Bultan and P.-A. Hsiung, editors, *Proc. ATVA*, LNCS 6996, pages 432–442, Taipei, Taiwan, Oct. 2011. Springer.

[127] X. Jin, A. Donzé, and G. Ciardo. Mining weighted requirements from closed-loop control models. In *Proc. Sixth International Workshop on Numerical Software Verification (NSV)*, April 2013.

[128] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 43–52. ACM, 2013.

[129] X. Jin, Y. Lembachar, and G. Ciardo. Symbolic verification of eca rules. In *International workshop on Petri Nets and Software Engineering (PNSE)*, volume 989, pages 35–53, 2013.

[130] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[131] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[132] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[133] K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in SQL3. In *Active Rules in Database Systems*, pages 197–219. Springer, New York, 1999.

[134] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. In *Proc. TACAS*, LNCS 2619, pages 52–66. Springer, Apr. 2003.

[135] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th Conference on Design Automation*, pages 608–613. IEEE Computer Society Press, June 1992.

[136] K. G. Larsen, P. Petterson, and W. Yi. UPPAAL: status & developments. In *Proc. CAV*, pages 456–459, 1997.

[137] S. Lasota and I. Walukiewicz. Alternating timed automata. In *Foundations of Software Science and Computational Structures*, pages 250–265. Springer, 2005.

[138] S. Lauesen and O. Vinter. Preventing requirement defects: An experiment in process improvement. *Requirements Engineering*, 6(1):37–50, 2001.

[139] C. Lee, F. Chen, and G. Rosu. Mining parametric specifications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, page 591600, 2011.

[140] E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, University of California, Berkeley, May 2007.

[141] E. A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008. Invited Paper.

[142] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[143] J. J. Li and W. E. Wong. Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In *Proc. IEEE ISORC*, pages 181–185, 2002.

[144] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, page 755760, 2010.

[145] X. Li, J. M. Marín, and S. V. Chapa. A structural model of ECA rules in active database. In *Proc. of the Second Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*, MICAI '02, pages 486–493. Springer-Verlag, 2002.

[146] D. P. Lindorff. *Theory of sampled data control systems*. Wiley, 1965.

[147] J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18(1):39–76, 2011.

[148] I. Lopes Margarido, J. Faria, R. Vidal, and M. Vieira. Classification of defect types in requirements specifications: Literature review, proposal and assessment. In *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, pages 1–6, 2011.

[149] D. Luenberger. *Introduction to dynamic systems: theory, models, and applications*. Wiley, 1979.

[150] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Softw. Eng.*, 20:149–162, Feb. 1994.

[151] J. Lygeros, G. Pappas, and S. Sastry. An introduction to hybrid system modeling, analysis, and control. *Preprints of the First Nonlinear Control Network Pedagogical School*, pages 307–329, 1999.

[152] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.

[153] P. Marwedel. *Embedded system design: Embedded systems foundations of cyber–physical systems.* Springer Science+ Business Media, 2011.

[154] A. P. Mathur. *Foundations of Software Testing.* Pearson Education, 2008.

[155] A. P. Mathur. *Foundations of software testing.* Pearson Education, 2008.

[156] D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM Sigmod Record*, 18(2):215–224, 1989.

[157] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[158] K. L. McMillan. The smv language. *Cadence Berkeley Labs*, pages 1–49, 1999.

[159] D. L. Mills. Exterior gateway protocol formal specification, 1984.

[160] Modelica. *version 2.0.* Modelica Association, `www.modelica.org`, 2012.

[161] T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.

[162] D. Nazareth. Investigating the applicability of Petri nets for rule-based system verification. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):402–415, 1993.

[163] T. Nghiem, S. Sankaranarayanan, G. E. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proc. of Hybrid Systems: Computation and Control*, pages 211–220, 2010.

[164] G. Nicolescu. *Model-Based Design for Embedded Systems.* CRC Press, 2009.

[165] NSF. National science foundation, cyber-physical systems (CPS) program solicitation NSF 08-611. http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.htm.

[166] K. Ogata. *Discrete-time control systems*, volume 1. Prentice-Hall Englewood Cliffs, NJ, 1987.

[167] H. Ogawa, M. Matsuki, and T. Eguchi. Development of a power train for the hybrid automobile: The civic hybrid. *SAE transactions*, 112(3):373–384, 2003.

[168] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Proc. ICATPN*, LNCS 815, pages 416–435. Springer, June 1994.

[169] A. Platzer. *Logical Analysis of Hybrid Systems.* Springer, 2010.

[170] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57. IEEE Comp. Soc. Press, Nov. 1977.

[171] A. Pnueli. Development of hybrid systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 77–85. Springer, 1994.

[172] A. A. Porter and L. G. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 103–112, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[173] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *Proc. ICSTW*, pages 132 –141, 2010.

[174] R. Pratap. *Getting started with MATLAB*. Saunders College Publishing, 2002.

[175] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010.

[176] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[177] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. ICCAD*, pages 154–158. IEEE Comp. Soc. Press, 1995.

[178] I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. In *Proc. of the 5th East European Conference on Advances in Databases and Information Systems*, pages 266–279. Springer-Verlag, 2001.

[179] G. W. Recktenwald. *Numerical methods with MATLAB: implementations and applications*. Prentice Hall Upper Saddle River, NJ, 2000.

[180] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4), 2006. RFC 4271.

[181] K. Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.

[182] S. Sankaranarayanan and G. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2012.

[183] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Mining library specifications using inductive logic programming. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, page 131140, 2008.

[184] Sha, Lui and Abdelzaher, Tarek and Årzén, Karl-Erik and Cervin, Anton and Baker, Theodore and Burns, Alan and Buttazzo, Giorgio and Caccamo, Marco and Lehoczky, John and Mok, Aloysius K. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[185] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *Software Engineering, IEEE Transactions on*, 34(5):651666, 2008.

[186] Simulink. *version 8.0 (R2012b)*. The MathWorks Inc., `www.mathworks.com/products/simulink/`, 2012.

[187] Simulink. *Thermal Model of a House.* The MathWorks Inc., `www.mathworks.com/products/simulink/examples.html?file=/products/demos/shipping/simulink/sldemo_househeat.htmls`, 2013.

[188] S. Skogestad and I. Postlethwaite. *Multivariable feedback control: Analysis and Design.* Wiley, 2007.

[189] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[190] M. Solé and E. Pastor. Traversal techniques for concurrent systems. In *Proc. FMCAD*, LNCS 2517, pages 220–237. Springer, Nov. 2002.

[191] S. Sridhar, A. Hahn, and M. Govindarasu. Cyber–physical system security for the electric power grid. *Proceedings of the IEEE*, 100(1):210–224, 2012.

[192] I. Standard. Iso 11898, 1993. *Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication*, 1993.

[193] U. Stern and D. L. Dill. Parallelizing the murϕ verifier. In *Proc. CAV*, LNCS 1254, pages 256–278. Springer, June 1997.

[194] A. Tanenbaum. *Computer networks.* Prentice Hall, 2003.

[195] TargetLink. *version 3.3.* dSPACE, `www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm`, 2012.

[196] J. Taylor. Toward a modeling language standard for hybrid dynamical systems. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, volume 3, pages 2317 –2322, dec 1993.

[197] J. Taylor. A modeling language for hybrid systems. In *Proceedings., IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pages 339 –344, mar 1994.

[198] J. Taylor and D. Kebede. Modeling and simulation of hybrid systems. In *Proceedings of the 34th IEEE Conference on Decision and Control*, volume 3, pages 2685 –2687, dec 1995.

[199] P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[200] A. Tiwari. Hybridsal relational abstracter. In *CAV*, pages 725–731, 2012.

[201] R. Valk. Generalizations of Petri nets. In *Mathematical foundations of computer science*, LNCS 118, pages 140–155. Springer, 1981.

[202] A. Valmari. A stubborn attack on the state explosion problem. In *Proc. CAV*, pages 156–165. Springer, June 1991.

[203] D. Varró. A formal semantics of uml statecharts by model transition systems. In *Processings of ICGT 2002: International Conference on Graph Transformation*, pages 378–392. Springer, 2002.

[204] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. SOFSEM*, pages 582–594, 2009.

[205] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. SOFSEM*, LNCS 5404, pages 582–594. Springer, 2009.

[206] F.-Y. Wang. Parallel control and management for intelligent transportation systems: Concepts, architectures, and applications. *Intelligent Transportation Systems, IEEE Transactions on*, 11(3):630–638, 2010.

[207] W. Weimer and G. Necula. Mining temporal specifications for error detection. *Tools and Algorithms for the Construction and Analysis of Systems*, page 461476, 2005.

[208] D. Work and A. Bayen. Impacts of the mobile internet on transportation cyber-physical systems: traffic monitoring using smartphones. In *National Workshop for Research on High-Confidence Transportation Cyber-Physical Systems: Automotive, Aviation, & Rail*, pages 18–20, 2008.

[209] D. Work, A. Bayen, and Q. Jacobson. Automotive cyber-physical systems in the context of human mobility. In *National Workshop on high-confidence automotive cyber-physical systems, Troy, MI*, 2008.

[210] H. Yang, B. Hoxha, and G. Fainekos. Querying parametric temporal logic properties on embedded systems. In *Testing Software and Systems*, pages 136–151. Springer, 2012.

[211] K. Yano, M. Koga, and E. Yarnarnura. Control system design method based on multiple-precision arithmetic with guaranteed accuracy. In *SICE Annual Conference, 2008*, pages 1772–1777. IEEE, 2008.

[212] Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proc. ATVA*, LNCS 5799, pages 368–381. Springer, 2009.

[213] Y. Zhao and G. Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.

[214] Y. Zhao, J. Xiaoqing, and G. Ciardo. A symbolic algorithm for shortest EG witness generation. In *Proc. TASE*, pages 68–75. IEEE Comp. Soc. Press, 2011.