Ammar Mohammed Ammar

Hybrid Multi-agent Systems: Modeling, Specification and Verification

Dissertation

Department of Computer Science
University of Koblenz-Landau

Ammar Mohammed Ammar

Hybrid Multi-agent Systems: Modeling, Specification and Verification

Vom Promotionsausschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.) genehmigte Dissertation.

Vorsitzender des Promotionsausschusses:	Prof. Dr. Dieter Zöbel
Vorsitzender der Promotionskommission:	Prof. Dr. Dietrich Paulus
Berichterstatter:	Prof. Dr. Ulrich Furbach
	Prof. Dr. Frieder Stolzenburg

Die wissenschaftliche Aussprache fand am 27. Oktober 2010 statt.

Acknowledgements

No man is an island. This thesis would not have been possible without the support and encouragement of many people to whom I would like to express my deep gratitude here.

First of all, I would like to express my deep and sincere gratitude to my supervisor, Prof. Dr. Ulrich Furbach the head of Artificial Intelligence Research Group (AGKI), for giving me the opportunity to work in his group and for his valuable support and guidance throughout doing this thesis. I have learned a lot from his discussion during our regular group-meeting seminars.

I'm thankful to Prof. Dr. Frieder Stolzenburg not only for reviewing thesis, but also for his precious ideas that helped me to do work together.

I am also grateful to my current and previous colleagues at the AGKI. In particular, i would like to thank Prof. Dr. Bernhard Beckert for supporting me with questions and suggestions at many meeting. Special thanks goes to Christian Schwarz for his feedback on drafts of the thesis and for good collaboration on related topics. Further, I would like to thank Claudia schon and Björn Pelzer for their help in proof-reading some of my papers. Special thanks goes to Beate Köner who helped me and my family to tackle lots of obstacles. I would like also to extend my thanks to the following members and ex-members of the group who helped me directly or indirectly during working in AGKI (in no particular order) Markus Maron, Christoph Gladisch, Ekaterina Pek, Thorsten Bormer, Gerd Beuster, Margret Gross-Hardt and Jan Murray.

Outside of AGKI, I owe thanks to all my friends in Germany. Among of them I am indebted to Safiye Ilhan, Gökhan Er, Fatih Gülen and Jessica Gülen.

Finally, the greatest encouragement was given to me by my Family: my mother, wife and children. They provided me with emotional support and patience. Without them, this work wouldn't be complete.

I gratefully acknowledge the support of my work by Egyptian Ministry of higher education as well as university of Koblenz-landau.

Koblenz, November 2010 *Ammar Mohammed Ammar*

Abstract

Specifying behaviors of multi-agent systems (MASs) is a demanding task, especially when applied in safety-critical systems. In the latter systems, the specification of behaviors has to be carried out carefully in order to avoid side effects that might cause unwanted or even disastrous behaviors. Thus, formal methods based on mathematical models of the system under design are helpful. They not only allow us to formally specify the system at different levels of abstraction, but also to verify the consistency of the specified systems before implementing them. The formal specification aims a precise and unambiguous description of the behavior of MASs, whereas the verification aims at proving the satisfaction of specified requirements.

A behavior of an agent can be described as discrete changes of its states with respect to external or internal actions. Whenever an action occurs, the agent moves from one state to another one. Therefore, an efficient way to model this type of discrete behaviors is to use a kind of state transition diagrams such as finite automata. One remarkable advantage of such transition diagrams is that they lend themselves formal analysis techniques using *model checking*. The latter is an automatic verification technique which determines whether given properties are satisfied within a model underlying a particular system.

In realistic physical environments, however, it is necessary to consider continuous behaviors in addition to discrete behaviors of MASs. Examples of those type of behaviors include the movement of a soccer agent to kick off or to go to the ball, the process of putting out the fire by a fire brigade agent in a rescue scenario, or any other behaviors that depend on any timed physical law. The traditional state transition diagrams are not sufficient to combine these types of behaviors.*Hybrid automata* offer an elegant method to capture such types of behaviors. Hybrid automata extend regular state transition diagrams with methods that deal with those continuous actions such that the state transition diagrams are used to model the discrete changes of behaviors, while differential equations are used to model the continuous changes. The semantics of hybrid automata make them accessible to formal verification by means of model checking.

The main goal of this thesis is to approach hybrid automata for specifying and verifying behaviors of MASs. However, specifying and and verifying behaviors of MASs by means of hybrid automata raises several issues that should be considered. These issues include the complexity, modularity, and the expressiveness of MASs' models. This thesis addresses these issues and provides possible solutions to tackle them.

Zusammenfassung

Die Beschreibung des Verhaltens eines Multi-Agenten-Systems (MAS) ist eine fordernde Aufgabe, besonders dann, wenn es in sicherheitskritischen Umgebungen eingesetzt werden soll. Denn in solchen Umgebungen muss die Beschreibung besonders sorgfältig ausgeführt werden um Seiteneffekte zu vermeiden, die ungewünschte oder sogar zerstörische Folgen haben könnten. Deshalb sind formale Methoden nützlich, die auf mathematischen Modellen des zu entwerfenden Systems basieren. Sie erlauben es nicht nur das System formal auf verschiedenen Abstraktionsebenen zu spezifizieren, sondern auch seine Konsistenz noch vor der Implementation zu verifizieren. Das Ziel der formalen Spezifikation ist eine präzise und eindeutige Beschreibung des Verhaltens des Multi-Agenten-Systems, während die Verifikation darauf abzielt, geforderte Eigenschaften dieses Systems zu beweisen.

Üblicherweise wird das Verhalten eines Agenten als diskrete Änderung seines Zustands im Bezug auf externe oder interne Aktionen aufgefasst. Jedes mal, wenn eine Aktion auftritt, ändert sich der Zustand des Agenten. Deshalb sind Zustandsübergangsdiagramme bzw. endliche Automaten ein naheliegender Ansatz das Verhalten zu modellieren. Ein weiterer Vorteil einer solchen Beschreibung ist, dass sie sich für das sogenannte *Model Checking* eignet. Dabei handelt es sich um eine automatische Analysetechnik, die bestimmt, ob das Modell des Systems spezifizierten Eigenschaften genügt.

Allerdings muss in realistischen, physikalischen Umgebungen neben dem diskreten auch das kontinuierliche Verhalten des Multi-Agenten-Systems betrachtet werden. Dabei könnte es sich beispielsweise um die Schussbewegung eines Fussballspieler-Agenten, den Prozess des Löschen durch einen Feuerwehr-Agenten oder jedes andere Verhalten handeln, das auf zeitlichen physikalischen Gesetzen basiert. Die üblichen Zustandsübergangsdiagramme sind nicht ausreichend, um diese beiden Verhaltensarten zu kombinieren. *Hybride Automaten* stellen jedoch eine elegante Lösung dar. Im Wesentlichen erweitern sie die übliche Zustandsübergangsdiagramme durch Methoden, die sich mit kontinuierlichen Aktionen befassen. Die Zustandsübergänge modellieren weiterhin die diskreten Verhaltenswechsel, während Differentialgleichungen verwendet werden um das kontiniuierliche Verhalten zu beschreiben.

Besonders geeignet erscheinen Hybride Automaten, weil ihre formale Semantik die Verfikation durch Model Checking erlaubt.

Deshalb ist das Hauptziel dieser Arbeit, Hybride Automaten für die Modellierung und die Verifikation des Verhaltens von Multi-Agenten-Systemen einzusetzen. Jedoch bringt ihr Einsatz mehrere Probleme mit sich, die betrachtet werden sollten. Zu diesen Problemfeldern zählen Komplexität, Modularität und die Aussagestärke der Modelle. Diese Arbeit befasst sich mit diesen Problemen und liefert mögliche Lösungen.

Contents

List of Figures

1 Introduction

In this Chapter, we motivate our work, outline the thesis and summarize its contributions.

1.1 Overview and Motivation

Multi-Agent Systems (MASs) is the subfield of Artificial Intelligence that aims at providing principles for building complex systems involving several interacting agents. An agent is an autonomous decision maker on behalf of some real world entity. It is generally agreed that there is no universally accepted definition of the term agent, but the one presented in this thesis is taken from [Wooldridge and Jennings, 1995]:

> *An agent is as an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.*

Generally, the agent acts in its environment according to a reasoning process that relies on its internal behaviors/states and the stimulus received thereof. An abstract relation between the agent and its environments is described in [Russell et al., 2003]. In this abstract, the agent is seen as a reactive component which monitors its environment through sensors and acts upon it through effectors.

When several agents operate and interact in an environment, they form what is called a Multi-Agent System (MAS). According to [Wooldridge., 2002], an MAS is defined as a distributed system containing a collection of agents that work together in order to solve problems. Agents in an MAS should be able to interact through communication and cooperate in order to fulfill certain tasks.

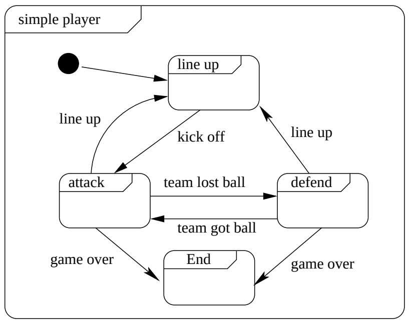


Fig. 1.1. A description of a simple agent as a transition diagram.

The development of MAS applications to be applied in safety-critical systems—a critical system is a system that must satisfy critical properties, such as safety, real-time and security properties—asks for specifying their behaviors cautiously in order to avoid side effects that might bring about unwanted or even disastrous behaviors. To tackle this challenge, the use of rigorous techniques in specification and analysis of the MASs is required. For this purpose, formal techniques based on mathematical models of the system under design are helpful. They allow not only us to formally specify the system at different levels of abstraction, but also to analyze the consistency of the specified systems before implementing them. The formal specification aims at presenting a precise and unambiguous behavior description of an MAS, whereas the formal verification looks at proving the compliance with specified requirements.

An agent behaves with respect to the occurrence of external or internal actions. Whenever an action occurs, the agent moves from one state to another. Therefore, an efficient way to model agents' behaviors is to use state transition diagrams. Fig. 1.1 shows the behavior of a simple abstract agent playing soccer modeled as a state transition diagram. Formally, a state transition diagram is defined by a set of states and a set of possible transitions. Each transition is labeled by the name of an action or event whose occurrence triggers the change of state.

One remarkable advantage of state transition diagrams is that they allow for formal analysis using *model checking* techniques [Clarke et al., 1999]. Model checking is an automatic verification technique, which determines whether given properties of a system are satisfied by a model described as a transition system. A model checker takes both a model and a property specified by using temporal logics and automatically checks either whether the property is correct or a counter-example falsifying that property.

Although state transition diagrams can describe the discrete behaviors of agents in terms of how the agents act in certain scenarios, it is necessary to consider continuous behaviors too. Examples of such types of behaviors include the movement of a robot to kick off or to go to the ball, the process of putting out the fire by a fire brigade agent in a rescue scenario, the approaching of a train to a gate controlling a road intersection, or any other behaviors that depend on any continuous physical law. This asks for a method that can capture both types of behaviors. *Hybrid automata* [Henzinger, 1996] offer an elegant method to model such types of behaviors. They integrate differential equations within regular state transition diagrams. The state transition diagrams are used to model the discrete changes of the agents' behaviors, while differential equations are used to model the continuous changes. The semantics of hybrid automata make them accessible to formal verification by means of model checking. Thus, it is possible to prove desirable features and the absence of unwanted properties for those systems, which are modeled using hybrid automata. Hybrid automata cannot only be used to specify behaviors of MASs, but also to prove their properties.

Specifying and verifying behaviors of MASs by means of hybrid automata, however, reveal several issues that should be taken into consideration. The first issue deals with the main challenge of applying model checking to MASs. Within hybrid automata, the team of agents is described as concurrent automata. It is known that the major problem in applying model checking is the potential combinatorial explosion of the state space arising from analyzing concurrent systems. The problem becomes more complex when permitting continuous dynamics within systems. This is why a model checker keep tracks not only of the part of the explored state space, but also of the timing and continuous evolution associated with each state, which is time and spac-consuming. This requires techniques that help to cope with this problem.

Another important issue deals with the modularity of hybrid automata models. Hybrid automata lack support for modularity being an important aspect when we model complex MASs containing similar sub-systems. Therefore, the description of the internal behavior of each agent as well as the external interactions among agents are equally visible and are considered to be at the same level of abstraction. Models of MASs can be cluttered and illegible as a result. This asks for structured and systematic methods to support modularity and to analyze the behaviors of complex systems.

A further issue deals with the expressiveness of hybrid automata to specify behavior of MASs. When the behaviors of agents are defined using hybrid automata, their decision making relies on the evolution of the continuous dynamics. However, there are still favorable situations for agents to make decisions depending on some utility/payoff functions, e.g. shortest distance, max. or min. values that might appear during the continuous evolution of agents. Neither hybrid automata nor their support tools can model such type of behaviors.

An additional issue deals with the expressiveness of the standard tools of hybrid automata to specify and verify those properties of MASs that depend on the occurrences of events. The importance of events stems from their ability to not only construct the overall model of an MAS through the composition of agents, but also to reason about behaviors of the MAS through communication among agents. The standard tools of hybrid automata, e.g. Hytech [Henzinger et al., 1997] and PHAVer [Frehse, 2005], provide little support to verify properties of events directly. In order to do so, these tools have to indirectly re-specify those properties into an acceptable form to the verification engine in a way that may add further complexity to the original model. Let us assume that one wants to specify and prove that whenever an agent sends a request, it will be acknowledged within t time units in a model M of an MAS. A typical solution to verify this with standard hybrid automata tools is to translate the previous specification into a model A. Then, the original goal to verify the specification is to check whether the parallel model of A and M can reach a designated state of A. It is an advantage if one can verify such types of properties directly from the original model without the process of composition.

1.2 Contributions

The expected main contribution of this thesis is to approach hybrid automata for specifying and verifying behaviors of MASs and to provide ways for addressing the challenging issues, which have been previously mentioned. More precisely, this thesis provides a novel framework to specify and to verify MASs based on hybrid automata. The framework presents an approach that addresses the complexity raised by the composition of agents by constructing the composition of agents' behaviors dynamically during the verification process such that the only necessary parts of state space are considered.

Additionally, the framework presents a novel variant of temporal logics, called RCTL (Region Computation Tree Logic) which extends CTL in order to specify both qualitative and quantitative properties of systems under consideration.

The thesis provides various aspects to extend hybrid automata. Firstly, the thesis presents a slight extension to hybrid automata allowing agents to have control over their behaviors in a way that they can react to the change of the environment based on their preferences. Secondly, in order to cope with complex multi-agent structures, the thesis shows how to integrate the hierarchical notations of UML statecharts together with the formal semantics of hybrid automata. This integration is advantageous. On the one hand, hierarchical notations allow specifying MASs with different levels of abstraction. On the other hand the formal semantics of hybrid automata allow for analyzing the behaviors of those MASs.

Graphical modeling languages are used extensively to specify behaviors of systems, particularly MASs. Although they do not require experts and are favored by a lot of users, they provide little support for formal analysis of those systems. For bridging this gap, the thesis proposes to use graphical notations for specifying behaviors of MASs and formal verification to support analysis of those MASs.

1.3 Publications

Almost results presented in this thesis have already been published in the proceedings of various international conferences, workshops and in a book. The following is a full list of these publications.

- Mohammed, A. and Furbach, U. (2010a). Extending CTL to Specify Quantitative Temporal Requirements. In Sopena, J. G. and l. Capel-Tunon, M., editors, In Proceedings of the 8th International Workshop on Modeling, Simulation, Verication and Validation of Enterprise Information Systems, MSVVEIS 2010, pages 70–79, Funchal, Madeira, Portugal. INSTICC PRESS. Held in conjunction with 11th International Conference on Enterprise Information Systems (ICEIS 2010).

- Mohammed, A. and Furbach, U. (2010b). Multi-agent systems: modeling and verification using hybrid automata. In Lars Braubach, J.-P. B. and Thangarajah, J., editors, Post-Proceedings of 7th International Workshop on Programming Multi-Agent Systems at 8th International Joint Conference on Autonomous Agents and Multi-Agent Systems, LNAI 5919, pages 49–66. Springer, Berlin, Heidelberg.

- Mohammed, A., Furbach, U., and Stolzenburg, F. (2010). Multi-robot systems: Modeling, specification, and model checking. In Papic, V., editor, Robot Soccer, chapter 11, pages 241–265. IN-TECH.

- Schwarz, C., Mohammed, A., and Stolzenburg, F. (2010). A tool environment for specifying and verifying multi-agent systems. In Filipe, J., Fred, A., and Sharp, B., editors, Proceedings of the 2nd International Conference on Agents and Articial Intelligence, volume 2, pages 323–326. INSTICC Press.

- Mohammed, A. and Schwarz, C. (2009). Hieromate: A graphical tool for specification and verification of hierarchical hybrid automata. In B. Mertsching, M. H. and Aziz, Z., editors, KI 2009: Advances in Articial Intelligence, Proceedings of the 32nd German Conference on Articial Intelligence, LNAI 5803, pages 695–702.Springer.

- Mohammed, A. and Furbach, U. (2009). From reactive to deliberative multi-agent planning. In Ultes-Nitsche, U., Moldt, D., and Augusto, J. C., editors, In Proceedings of the 7th International Workshop on Modelling, Simulation, Verication and Validation of Enterprise Information Systems, MSVVEIS 2009, pages 67–75, Milan, Italy. INSTICC PRESS. Held in conjunction with 11th International Conference on Enterprise Information Systems (ICEIS 2009).

- Mohammed, A. and Stolzenburg, F. (2008). Implementing hierarchical hybrid automata using constraint logic programming. In Schwarz, S., editor, Proceedings of 22nd Workshop on (Constraint) Logic Programming, pages 60–71, Dresden. University Halle Wittenberg, Institute of Computer Science. Technical Report 2008/08.

- Mohammed, A. and Furbach, U. (2008a). Modeling multi-agent logistic process system using hybrid automata. In Ultes-Nitsche, U., Moldt, D., and Augusto, J. C., editors, In Proceedings of the 7th International Workshop on Modelling, Simulation, Verication and Validation of Enterprise Information Systems, MSVVEIS 2008, pages 141–149, Barcelona, Spain. INSTICC PRESS. Held in conjunction with 10th International Conference on Enterprise Information Systems (ICEIS 2008).

- Mohammed, A. and Furbach, U. (2008b). Using CLP to model hybrid systems. In Proceedings of Annual ERCIM Workshop on Constraint Solving

Programming (CSCLP2008), Rome, Italy. Published online http://pst.istc.cnr.it/CSCLP08/program/index.h

1.4 Structure of the Thesis

The rest of the thesis is organized as follows:

Part I

Chapter 2 provides introductory material on hybrid automata.

Chapter 3 shows how to use hybrid automata to model behaviors of MASs. It demonstrates that by modeling an MAS scenario that follows a standard interaction protocol. With one of the standard model checkers of hybrid automata, the Chapter shows how several properties about this scenario can be investigated. The contribution of this Chapter has been published in [Mohammed and Furbach, 2008a].

Part II

Chapter 4 discusses the syntax and semantics of the new proposed approach, which aims at constructing behaviors of MASs on-the-fly during the verification phase. The Chapter also shows how to implement the proposed approach using constraint logic programming. The main core of this Chapter has been published in [Mohammed and Furbach, 2010b; Mohammed et al., 2010]. An early implementation of the model has been published in [Mohammed and Furbach, 2008b; Mohammed and Stolzenburg, 2008].

Chapter 5 introduces the syntax and semantics of the quantitative temporal logic RCTL. It demonstrates how several RCTL requirements can be verified using the model presented in Chapter 4. In addition, the Chapter surveys the other quantitative temporal logics related to RCTL. The contribution of this Chapter has been published in [Mohammed and Furbach, 2010a].

Chapter 6 evaluates the proposed approach with several standard examples taken from the context of hybrid automata. The Chapter refers to those works that are related to our proposed approach as well. The main results of this Chapter are presented in [Mohammed and Furbach, 2009a].

Part III

Chapter 7 shows that hybrid automata can be used as a conceptual model for planning the behavior of MASs. The Chapter focuses on the key relation between planning problems and model checking. Furthermore, it looks at extending the decision making of the hybrid automata models to contain preferences of agents. The main contribution of this Chapter has been published in [Mohammed and Furbach, 2009a].

Chapter 8 presents an approach to extend hybrid automata with hierarchical notations. It discusses the formal syntax and semantics of this extension. It also implements a prototype of this approach using constraint logic programming. Furthermore, the Chapter supports the evaluation of this approach with several examples and discusses other related work. The contribution of this Chapter is presented in [Mohammed and Stolzenburg, 2008; Mohammed et al., 2010].

Chapter 9 presents a tool environment that integrates the implementations of those models which are presented in Chapter 4 and 8. This tool aims at simplifying the specification process by incorporating graphical notations within the models. The Chapter demonstrates the tool on an MAS scenario taken from the Robocup Rescue. Additionally, the Chapter shows other work related to this tool. The contribution of this Chapter has been published in [Mohammed and Schwarz, 2009; Schwarz et al., 2010].

Part IV

Chapter 10 summarizes the thesis and shows some futures work.

Part I

Background

2 Background literature

This chapter displays background material on hybrid automata.

2.1 Introduction

Reactive systems are coined in [Harel and Pnueli, 1985] to describe those systems that react to inputs from an environment by generating corresponding responses. Typical examples of such systems include on-line interactive systems, such as automatic teller machines (ATMs) and flight reservation systems; computer-embedded systems, such as automotive and telecommunication systems; and control systems, such as chemical and manufacturing systems.

A special class of systems which belongs to reactive systems is the class of *real-time systems*. In such systems, the reaction to a certain stimulus should be done within given time bounds. For example, a gate controlling a rod crossing tracks of trains should be forced to close the rod within reasonable time during the approaching of any train.

Another important class of systems which belongs to reactive systems is the class of those systems which react to their environment according to the evolution of their own physical rules. Such types of systems are known as *Hybrid systems*. A hybrid system is defined as a reactive system consisting of continuous and discrete components [Olderog and Dierks, 2008]. The continuous components are time-dependent physical variables ranging over a continuous value set, like speed, temperature, pressure or position. The discrete components are controllers that alter the physical variables in a desired way. An example for such type of hybrid systems is a heating system whose objective is to keep the room temperature within certain limits. Real-time systems can be considered as hybrid systems with at least one continuous variable

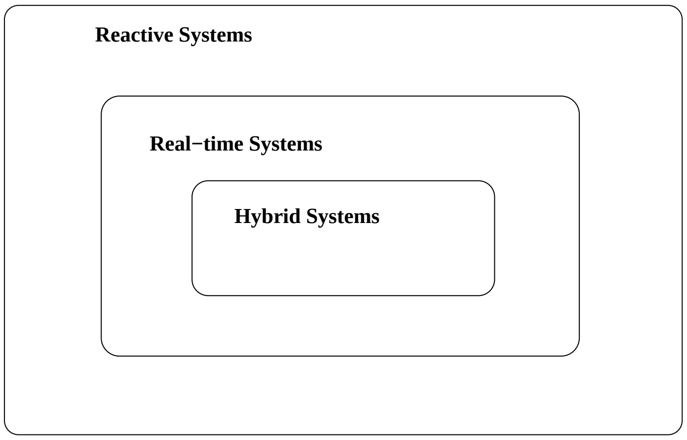


Fig. 2.1. Classes of systems

representing time. Real-time systems are often obtained as abstractions of the more detailed hybrid systems. The main relation between reactive systems and their special classes are summarized in Fig. 2.1 [Olderog and Dierks, 2008].

Reactive systems often appear in safety-critical applications where failure is unacceptable. Therefore, they must be carefully designed with a high degree of precision. For this purpose, the use of rigorous formal methods in specification and verification of such systems are helpful.

When formal methods are taken in consideration to specify and verify reactive systems, the classes of Fig. 2.1 are reversed, as shown in Fig. 2.2. One can conclude that formal methods of hybrid systems can be used as general methods to analyze real-time systems as well as reactive systems. We will concentrate on the formal methods of hybrid systems.

Formal methods provide ways to formally specify and verify systems. Specification is the process of describing a particular system and its desired requirements/properties. Formal specification is a technique using a language with a mathematically defined syntax and semantics. A formal specification of a system can help to obtain a better description and understanding of systems' abstraction. Formal verification provides an analysis method to verify the behavior of systems regarding their compliance with requirements.

Hybrid automata are mathematical formalisms that can formally capture the behavior of hybrid systems. Their formal semantics allows us to prove desirable features and the absence of unwanted properties in the specified systems. In the following, we will concentrate on hybrid automata as a formal model of specifying systems.

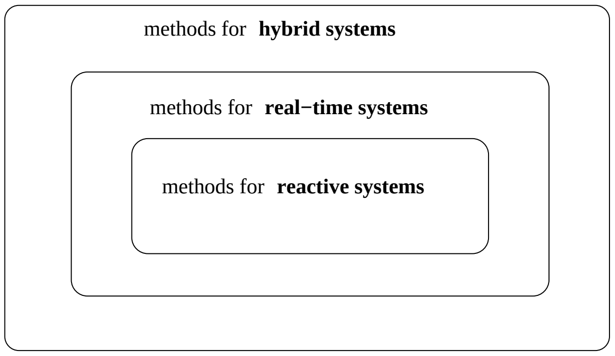

methods for **hybrid systems**
methods for **real−time systems**
methods for **reactive systems**

Fig. 2.2. Formal methods for systems

The rest of this chapter is organized as follows: Sec.2.2 provides a background on hybrid automata and their classes. Sec.2.3 briefly introduces reachability analysis as an automatic approach for verifications of hybrid automata.

2.2 Hybrid Automata

It is generally agreed that finite automata are a natural medium to describe dynamic behaviors of reactive systems. They are not sufficient to model real-time or continuous dynamical systems. Therefore, finite automata have been extended in ways to integrate the real-time or continuous dynamics. The most successful model of real-time systems is the timed automata [Alur and Dill, 1994]. Timed automata are finite automata equipped with a finite number of variables/clock representing time. The most successful model or hybrid systems are hybrid automata—they are also a natural generalization of timed automata—in which the finite automata are equipped with variables that represent the dynamical parts of systems.

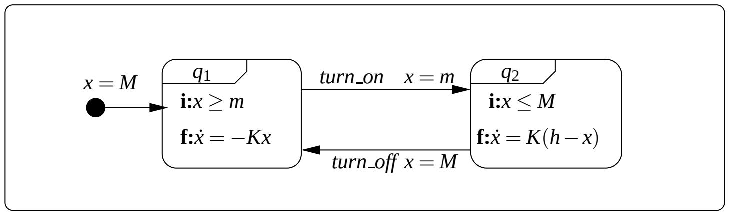


Fig. 2.3. A simple hybrid automaton.

2.2.1 What is Hybrid Automaton ?

A hybrid automaton [Henzinger, 1996] is a formal model to describe reactive systems with discrete and continuous components. A hybrid automaton $H = (\mathbb{X}, Q, Flow, Inv, Init, E, Jump, \sum, sync)$ consists of the following components:

- A finite set $\mathbb{X} = \{x_1, x_2, ..., x_n\}$ of real-valued variables that represent the continuous dynamics.
- A finite set Q of control locations or modes. It should be noted that in the classical automaton, these control locations are called states; however, that term is defined differently for hybrid automata.
- $Flow$ (continuous activity) is a labeling function that assigns to each control location $q \in Q$ a flow conditions $Flow(q)$ whose free variables are from $\mathbb{X} \cup \dot{\mathbb{X}}$, where the dotted variables $\dot{\mathbb{X}} = \{\dot{x_1}, \dot{x_2}, ..., \dot{x_n}\}$ denote the first derivative of the variables $\mathbb{X}$. When the control of hybrid automaton is in a location q, the variables evolve according to differentiable functions which satisfy the flow condition $Flow(q)$.
- An invariant Inv is a labeling function that assigns to each control location $q \in Q$ an invariant condition $Inv(q)$ whose free variables are in X.
- A labeling function $Init$ that assigns to each control location $q \in Q$ an initial condition $Init(q)$ whose free variables are from X.
- $E \subseteq Q \times Q$ is a finite set of discrete transitions– also called control switches– among control locations. Each transition $(q_1, q_2) \in E$ has a source location q_1 and target location q_2.
- An edge labeling function $Jump$ that assigns a jump condition– also called guard action– $jump(e)$ to each transition $e \in E$. The jump condition $jump(e)$ is a predicate whose free variables are from $\mathbb{X} \cup \mathbb{X}'$, where the primed variables $\mathbb{X}' = \{x'_1, x'_2, ..., x'_n\}$ are used to represent values at the conclusion of discrete change.Consequently, any jump condition relates the values of the variables before a discrete transition to the possible values after the discrete transition.
- A finite set $\sum$ of events which are used to synchronize concurrent automata.
- A labeling function $sync$:$E \rightarrow \sum$ that assigns to each transition $e \in E$ an event.

A hybrid automaton can be represented graphically, as a state diagrams of a finite state automaton augmented with flows, invariants, and jumps. Each location q is drawn as a circle or rectangle shape labeled with a name. Throughout this thesis, locations are drawn conventionally as rectangles with rounded corners. Furthermore, inside each location q, both the invariant $Inv(q)$ and the flow $Flow(q)$ are labeled with the symbols **i:** and **f:** respectively. Setting an invariant in a location q to be $true$—i.e. **i:**$true$ in the graphical representation—means that the invariant is always achievable at that location. On the other hand, setting flow to be $true$—i.e. **f:**$true$ means that nothing changes continuously. An edge $e = (q_1, q_2) \in E$ is represented graphically as an arrow from location q_1 to location q_2 labeled with the jump condition and the action event. We use guarded assignments to represent jump conditions; for example, assuming we have only a variable x. If the jump condition $x = 10, x := 0$ is declared on a transition, it stands for the jump condition $x = 10 \wedge x' = 0$. On the other hand, the jump condition of the form $x = 10, x := x$ stands for the jump condition $x = 10 \wedge x' = x$, which means that the value of x does not change before and after the discrete transition. Hence, we omit this type of assignments in the graphical representation.

Let us give an example of a hybrid automaton. Consider the hybrid automaton of Fig. 2.3, which models a thermostate. This hybrid automaton consists of two locations q_1 and q_2, and the variable x, which evolves under the differential equations $\dot{x} = -K \cdot x$ in location q_1, whereas evolves under differential equations $\dot{x} = K(h - x)$ in location q_1 for some constants K and h. The invariant associated with the locations q_1 and q_2 are $x \geq m$ and $x \leq M$ respectively for some predefined constant m and M. The initial location of the automaton starts in q_1 with $x = M$. There are two edges from q_1 to q_2 and vice versa with guards $x = m$ and $x = M$ respectively. In addition, the two edges are annotated with the events $turn_$on and $turn_$off.

The behavior of the thermostate automaton starts in location q_1, at which the heater is off. The temperature x decreases linearly proportionally to K. The heater stays off as long as the temperature exceeds the minimum m. When the temperature drops to m, the invariant for staying in off $(x \geq m)$ is violated, and the condition $x = m$ on the state transition labeled with $turn_$on is met and the control of the automaton jumps to the location q_2. In the later location, the heater stays on as long as the temperature does not exceed the maximum M. As soon as the invariant condition is violated, the thermostat switches the heater off again and returns to location q_1.

A run of a hybrid automaton starts from an initial state, and consists of infinite sequences of states, where the transition from one state to another state follows one of the following transitions:

- Discrete transitions corresponding to instantaneous transitions between control locations.

- Flow transitions corresponding to the continuous evolution of the system at a particular control location q according to the dynamics specified by the $Flow(q)$.

2.2.2 Automata Composition

Hybrid systems typically consist of several components that operate concurrently and communicate with each other. Each component can be described as a hybrid automaton. The component automata coordinate their behaviors through shared variables and synchronization labels. The automaton that models the entire system is obtained from the component automata using a product construction.

Formally, let $H_1 = (\mathbb{X}_1, Q_1, Flow_1, Inv_1, Init_1, E_1, Jump_1, \sum_1, sync_1)$ and $H_2 = (\ \mathbb{X}_2,\ Q_2,\ Flow_2,\ Inv_2,\ Init_2,\ E_2,\ Jump_2,\ \sum_2,\ sync_2\ )$ be two hybrid automata. The product automaton $H_1 \times H_2$ is a hybrid automaton $H = (\ \mathbb{X}_1 \cup \mathbb{X}_2, Q_1 \times Q_2, Flow, Inv, Init, E, Jump, \sum_1 \cup \sum_2, sync)$ with the following restrictions:

- The flow $Flow(q)$ of each product location $q = (q_1, q_2) \in Q_1 \times Q_2$ is $Flow_1(q_1) \wedge Flow_2(q_2)$.
- The invariant $Inv(q)$ of each product location $q = (q_1, q_2)$ is $Inv_1(q_1) \wedge Inv_2(q_2)$.
- Initial condition $Init(q)$ is $Init_1(q_1) \cup Init_2(q_2)$.
- Each transition $e = ((q_1, q_2), (q_1', q_2')) \in E$ if
  1. $e_1 = (q_1, q_1') \in E_1$, $q_2 = q_2'$, and $sync_1(e_1) \notin \sum_2$; or
  2. $e_2 = (q_2, q_2') \in E_2$, $q_2 = q_2'$, and $sync_2(e_2) \notin \sum_1$; or
  3. $e_1 = (q_1, q_1') \in E_1$,$e_2 = (q_2, q_2') \in E_2$, and $sync_1(e_1) = sync_2(e_2)$.

2.2.3 Classes of Hybrid Automata

In the literature of hybrid systems there are different classes of hybrid automata, depending on the type of continuous dynamics of the systems. For each class of dynamical laws, we obtain a class of hybrid automata. In the following we some of these important classes.

Linear Vs. Non-linear Hybrid Automaton

A *linear expression* over a set $\mathbb{X}$ of real valued variables is a linear combination of variables from $\mathbb{X}$ with rational coefficients. A *linear inequality* over $\mathbb{X}$ is an inequality between a rational constant and a linear expression. A *convex* *linear predicate* over $\mathbb{X}$ is a finite conjunction of linear inequalities over $\mathbb{X}$. A *linear predicate* is a finite disjunction of convex linear predicates.

A hybrid automaton H is called linear hybrid automaton [Alur et al., 1994] if it satisfies the following two requirements:

1. For each control location and each discrete transition, the flow, the invariant, the initial, and jump conditions are convex linear predicates.
2. For each control location $q \in Q$, the flow condition $Flow(q)$ is a predicate over the variables in $\dot{\mathbb{X}}$ only—i.e. does not contain any variables from $\mathbb{X}$.

A linear hybrid automata is called *simple* if the invariants and jump conditions are of the form $x \leq k$ or $x \geq k$, and all assignments are of the form $x := k$ or $x := x$, for a variable $x \in \mathbb{X}$ and an integer constant k.

When the flow condition $Flow(q)$ includes a predicate over both variables in $\dot{\mathbb{X}}$ and $\mathbb{X}$, a hybrid automaton is called *non-linear hybrid automata* [Henzinger et al., 1998b]. For example, the automaton of Fig. 2.3 is a non-linear hybrid automaton as it contains a flow of the form $\dot{x} = -K \cdot x$.

Discrete automaton

A variable $x \in \mathbb{X}$ is called a *discrete variable*, if its flow is of the form $\dot{x} = 0$ in each control location $q \in Q$. Thus, a discrete variable changes only when the control location changes. A *Discrete automaton* is a linear hybrid automaton of whose variables are discrete.

Timed and Multirate Automaton

For a linear hybrid automaton H, A variable $x \in \mathbb{X}$ is called a *skewed clock* if at every control location, the flow of x is determined by differential equation of the form $\dot{x} = k$ for a nonzero integer k, and on each transition $e \in E$ implies $x' = 0$ or $x' = x$; that is, the value of the variable x always increases uniformly with time at some fixed rate, and each transition either resets x to 0, or leaves it unchanged. A variable $x \in \mathbb{X}$ is called a *clock* if it is skewed clock with flow of the form $\dot{x} = 1$. A linear hybrid automaton H is called *timed automaton* [Alur and Dill, 1994] when the following hold:

1. Each variable $x \in \mathbb{X}$ is a clock.
2. All invariants and jump conditions are combinations of simple inequalities $x_1 \bowtie c$ or $x_1 - x_2 \bowtie c$, where $x_1, x_2 \in \mathbb{X}$, c is a nonnegative integer and the operator $\bowtie \in \{<, \leq, =, \geq, >\}$.

A linear hybrid automaton H is called *multirate automaton* [Alur et al., 1994] if each variable $x \in \mathbb{X}$ is a skewed clock.

Rectangular Hybrid Automaton

A rectangular inequality over real valued variables $\mathbb{X}$ is a formula $x \bowtie c$, where c is an integer constant, and $\bowtie$ is one of $\{<, \leq, \geq, >\}$. A *rectangular predicate* over $\mathbb{X}$ is a conjunction of rectangular inequalities.

A *Rectangular Automaton* is a hybrid automaton, in which all the initial conditions, invariants, flows, and jump conditions are rectangular predicates whose flow conditions refer only to variables in $\dot{\mathbb{X}}$. Thus, each continuous variable $x \in \mathbb{X}$ satisfies nondeterministic differential equation $a \leq \dot{x} \leq b$—also written as $\dot{x} = [a, b]$—where a and b are integer constants.

It is worth mentioning that adding several restrictions to a rectangle automaton can lead to further subclasses. A simple form of a rectangular automaton can be obtained from adding rectangular flow to a simple linear hybrid automaton. An *initialized rectangular automaton* [Henzinger et al., 1998a] can be obtained from rectangular automaton provided the following constraints are met: if each edge $e = (q_1, q_2)$ and for all $x \in \mathbb{X}$ flowing in both q_1 and q_2, then the value of x is nondeterministically reinitialized.

2.3 Reachability of Hybrid Automata

Automatic verification through model checking [Clarke et al., 1999] has been proven as a powerful technique for verifying finite-state systems. *Reachability analysis* is a variant of model checking, which amounts to compute iteratively all the reachable states of the systems from an initial state until reaching a fixed point. This can be done either enumeratively or symbolically. Reachability analysis has been motived to prove safety property; that is verifying that something *bad* never happens in a model underlying some systems. This property is encoded as: can a bad state be reached from an initial state by executing a model ? Technically, reachability analysis of a certain model can be performed by either forward or backward reachability. Forward reachability starts with an initial state I and checks if a run exists which can reach a target T. Backward reachability starts in a target T, and checks if a run exists which can reach to the the initial state I.

Recently reachability analysis of model checking has been extended to deal with hybrid systems. The *decidability problem* of such systems is one of the central issues. Given a class of hybrid systems, the decidability problem is to determine whether a certain property can be verified by an algorithm that terminates in a finite number of steps. Decidability is not an issue in the verification of purely finite state systems, since in the worst case the verification can be performed by exhaustively searching the whole state space.

In the case of hybrid systems, the decidability is a critical issue in algorithmic analysis because of the unaccountability of the state space. Although the reachability problem of hybrid automata is undecidable, there are several classes for which the reachability is decidable. In [Alur and Dill, 1994], the first decidability result for hybrid automata has been obtained for timed automata. In [Alur et al., 1994], it has been proven that the reachability problem over *multirate automata* is not decidable in general. By imposing a restriction on dynamics by what so called *simplicity condition*— i.e. the invariants and jump conditions are of the form $x \leq k$ or $x \geq k$, and all assignments are of the form $x := k$ or $x := x$, for a variable $x \in \mathbb{X}$ and an integer constant k— decidability for reachability problem can be achievable. In [Kopke, 1996], it has been also proven that the reachability of rectangular hybrid automata is in general undecidable, but it has been shown that the reachability of initialized rectangular automata is decidable.

Although the reachability problem for linear hybrid automata is undecidable, there are some algorithms for the analysis of time automata that have been extended to obtain semi-decision procedures for solving the verification problem of linear hybrid automata [Alur et al., 1994]. In order to analyze the behavior of nonlinear hybrid automata, there are techniques that approximate the non-linear linear hybrid automata with linear ones [Henzinger et al., 1998b]. Hytech [Henzinger et al., 1997] and PHAVer[Frehse, 2005] are examples of model checking tools supporting the previous procedures.

3

Multi-agent Scenario as Hybrid Automata

This chapter illustrates the use of hybrid automata to specify behaviors of Multi-agent systems (MASs). It describes a simple MAS scenario taken from the transportation logistic domain. The communication among the agents follows a well-known standard agent interaction protocol. With the help of Hytech, a standard model checker for hybrid automata, several properties of the MAS can be investigated. The contribution of this chapter has been presented in [Mohammed and Furbach, 2008a].

3.1 Introduction

The increasing interest in Multi-Agent Systems (MASs) has led to the development of new modeling languages and methodologies—a survey of those efforts are presented in [Wood and DeLoach, 2001]. The main purpose of these modeling languages is to offer notations to developers that are used to analyze, design, and implement MASs. In fact, most of theses methodologies have emerged from Unified Modeling Language [UML, 2009]. Among those methodologies, Agent UML [Bauer et al., 2001] has gained wide acceptance to model MASs. Agent UML basically extends UML with specific features including the sequence diagram, which has been chosen by the Foundation for Intelligent Physical Agents association (FIPA) [FIPA, 2002] as an acceptable standard language to model interactions among agents or what is the so-called Agent Interaction Protocol (AIP). Currently, one of the key features of any agent-based product has to be FIPA-compliant. For this aim, people working on agent development tools and libraries increasingly interested in offering the possibility to realize FIPA-compliant agent-based products.

Although methodologies of MASs are clear to understand and easy to develop, they are unable to verify the properties of MASs because of their lack of formal semantics or their ambiguous and vague semantics. To cope with this limitation, formal modeling approaches are helpful. Ideally, formal modeling approaches based on state transition diagrams can specify behaviors of MASs. This is because behaviors of an agent can be described as the discrete changes of its internal states with respect to an internal or external stimulus. In realistic physical environments, it is necessary to not only consider the discrete changes of the behaviors of the agent, but also their continuous changes. Therefore, hybrid automata are a suitable framework to capture both types of changes in a way that the discrete changes are modeled using a dialect of state transition diagrams, e.g. finite state machine, or finite automata, while the continuous changes are modeled using differential equations. Hybrid automata are equipped with formal semantics that make them accessible to formal validation of modeled behaviors. Thus, it is possibly to prove desirable features as well as the absence of unwanted properties for the modeled behavior automatically with the help of model checking methods.

To this end, this chapter aims at showing that an MAS, compliant to a standard agent interaction protocol, can be modeled using hybrid automata. In particular, the Chapter shows a model of an MAS scenario in a logistic process. Each agent involved in the scenario is described as a hybrid automaton and the communication between agents is represented using shared variables and synchronization labels. By using the formal verification of hybrid automata, several properties can be proven within the model. To do so, we use Hytech [Henzinger et al., 1997], a standard model checker of hybrid automata.

The rest of this chapter is organized as follows: Sec.3.2 describes the logistic scenario. Sec.3.3 describes the model of MAS in terms of hybrid automata. Finally, Sec.3.4 shows the formal verification of the model by means of model checking. Sec.3.5 shows related works.

3.2 Autonomous Logistic Processes

Getting the right products to the right place in time are the requirements in logistics. Nevertheless, with highly dynamic markets and increasingly complex logistic networks, it is becoming more and more difficult to meet these standards with conventional methods of planning and control. In future, aspects such as flexibility, adaptability and reactivity will be of primary importance. The paradigm of autonomous logistic processes [Scholz-Reiter et al., 2004] addresses these aspects by decentralizing logistic control to single logistic entities, e.g. freight items, transport containers, means of transport, or storage facilities. Therefore, autonomous logistic processes aim at managing logistics in a highly distributed way by transferring decision-making competencies to the logistic entities. MASs-engineering is an adequate and promising technique to implement the autonomous logistic process [Gehrke et al., 2006]. Logistic entities as well as secondary logistic services, e.g. traffic information, route planning and service brokerage, are represented by software agents interacting with each other to coordinate the logistic process. Agent communication and coordination follows standards defined by FIPA using Agent Communication Language (ACL) and interaction protocols for specific agent conversations. In what follows, we will describe an MAS in a logistic scenario and show how this can be modeled with hybrid automata.

3.2.1 Scenario Description

The MAS scenario constitutes four agents, namely *cargo*, *environment*, and two *trucks*. The *cargo* has the objective to be transported to a certain destination. The *trucks* may offer transportation service. Additionally, the environment agent represents an external disturbance to the transportation process. In the following, we will discuss the scenario in more details.

Initially, the *cargo* tries to contact the two trucks requesting for the transportation service. The two trucks are located in two different cities. When the *cargo* calls for a proposal, it supplies the trucks with information including destination point of the shipment and its due time. Onces each truck receives the call for proposal, it evaluates and estimates this request according to decision criteria—e.g. its speed limit, distance to destination and the deadline of delivery. The reason behind the estimation process is to know, whether the *truck* can provide the transportation under certain restrictions. If any truck can offer transportation, it accepts the proposal and initiates its intended price. In case the *cargo* received multiple proposals from trucks, it makes a contract with the *truck*, which provides the lowest price.

Once a contract is made, the *truck* begins the process of transportation. In the later case, the *truck* may be exposed to some environment conditions; that is, un-anticipated environmental interactions such as traffic or bad weather occurs. For simplicity, we will use two different environment conditions namely bad and good conditions. These conditions simply simulate the change of the environment in a way that influences the speed of the *truck*. The *truck* slows down to its minimum limit, whenever it is subjected to a bad condition environment, whereas it accelerates to its maximum limit, whenever environment conditions are good. The effect of the environment is seriously limited in this way. In reality, these conditions are more complex than

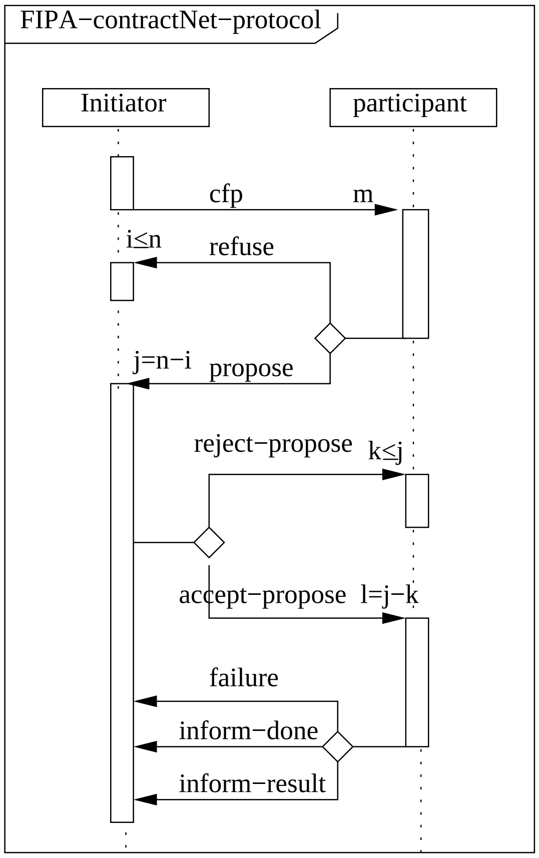


Fig. 3.1. FIPA contract net protocol.

our scenario. In a more realistic model of the environment, a stochastic characterization of disturbances would be used. Stochastic models, however, go beyond the expressiveness of current framework of hybrid automata.

At the end of the transportation process, the *truck* reports its delivery time with comparison to the due time. Therefore, if the *truck* delivered the shipment after the deadline, it informs the *cargo* with failure in the transportation; otherwise, it informs the *cargo* that the transportation was successful.

The previous scenario can be modeled using FIPA contract net protocol [FIPA, 2002], as it is shown in Fig. 3.1. In this protocol, the initiator and participant represent the *cargo* and *truck* respectively. The vertical lines represents the time threads from up to down. The arrows reflect the communication between the initiator and the participant. Each arrow is annotated with a communication message. Additionally, the number attached to any arrow indicates the number of participants in the message. As we previously mentioned that FIPA specification gains widely acceptance in modeling MASs especially for representing the interactions among the agents. It lacks, however, from proving certain properties of its model. In addition, FIPA specifications are unable to specify the internal behavior of the agent, and consequently suffer from the absence of decision making, which is crucial in MASs. Therefore, we intend in the next section to model the previous scenario using hybrid automata, and with the help of model checking we check certain features.

3.3 Model Specification

In this section, we show how to model the MAS scenario as concurrent hybrid automata. Each automaton represents an agent in the scenario. *truck*, *cargo*, and environment disturbance automata will be described in the following subsections in more details.

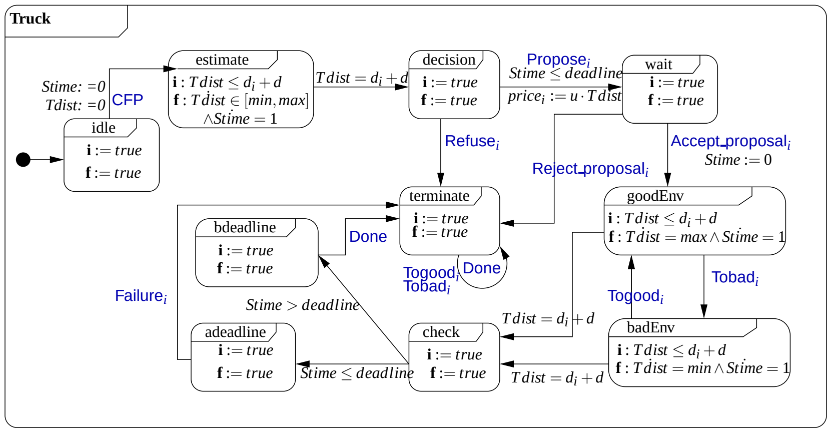


Fig. 3.2. Truck automaton.

Truck Automata

Fig. 3.2 depicts the model a truck as a hybrid automaton. In the scenario, there are two trucks having the same behaviors, but with different capabilities including the speed, total distance to travel, and the price needed to perform the transportation. These capabilities are marked in the model of Fig. 3.2 as, $\dot{Tdist}$, d_i and $price_i$, for $i = 1,2$. Initially, the behavior of the *truck* starts with location *Idle*, at which it waits for receiving any incoming proposal from the *cargo*. The initiation of the proposal is represented by the synchronization label *CFP*. Once the *truck* receives the *CFP* message, its control goes to the location *estimate*. In this location, the *truck* estimates and evaluates that proposal, in order to take the right decision; that is, whether it accepts or rejects the proposal. There are constraints that are involved in the estimation process including the speed limit of the *truck* and the expected delivery time. Once the estimation process has been done, the control goes to the location *decision*, from which the control goes to either the location *terminate* or *Wait*. The former location will be chosen, whenever the expected estimation time exceeds the deadline of delivering the shipment. However, if it goes to the *Wait*, the *truck* proposes to perform the transportation, and in this case bids its intended price. In the location *wait*, the *truck* waits for the type of incoming messages received from the *cargo*. If the *cargo* replies with rejection of the proposal, which is represented by synchronization label *Reject_proposal*, then the control goes to the location *terminate*. On the other hand, upon receiving a confirmation from the *cargo* with the acceptance of the proposal, the *truck* starts the transportation and goes to the location *goodEnv*. During performing the transportation, the *truck* mutually alters its behavior between the locations *goodEnv* and *badEnv* location according to the disturbance *Togood* and *Tobad* received from the environment. In both locations, the *truck* either accelerates to its maximum or slows down to its minimum speed. After certain time passes, the control goes to the location *check*, at which the *truck* checks its destination point against the deadline; that is whether after of before the deadline. In both cases, the *truck* has to inform the *cargo* with either failure or done.

Environment Automaton

Fig. 3.3 models an environment that generates disturbance during transportation process. This disturbance might occur as a reason of traffics, or a change in weather. The *Environment* automaton is augmented with the variable *envtime*, which calculates the elapsed time at both location *gconditon* and *bcondition*. The behavior of the environment automaton mutually oscillates between these two locations. The control waits for *gtime* units at the location *gcondition*, while it waits for *btime* time units at *bcondition* location, for given constants *gtime* and *btime*. The effect of the disturbance is terminated upon receiving the message *Done*.

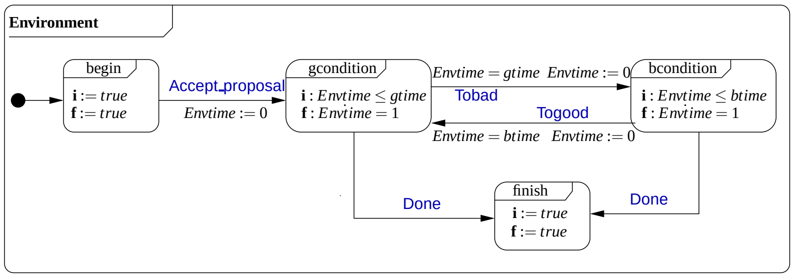


Fig. 3.3. Environment Automaton.

Cargo Automaton

The automaton *cargo* is shown in Fig. 3.4. The control of the *cargo* begins at the location *Start*. In this location, it initiates a call-for-proposal to all the possible trucks in the scenario by sending the message *CFP*. Then, it goes to the location *wait-proposal*, in which it reports the incoming messages received from the trucks. The messages are represented either by $Refuse_i$ or $Propose_i$ synchronization labels for $i = \{1,2\}$. Such messages indicate that a truck refuses or accepts the call-for-proposal. As soon as all trucks have sent their intended messages, the control goes to the location *evaluate*. From this location, the control may go to one of the locations *terminate*, *select*, or *bid*. The choice among these locations depends on the number of received proposals, such that if no truck offered a proposal, the control goes to the location *Terminate*, which means there is no *truck* agreed to perform the transportation. If there is only one *truck* offered a proposal, the control goes to location *Bid*. However, if more than a truck offered proposals, the control goes to the location *select*. At this latter location, the *cargo* selects the *truck* which provides the minimum price, and then the control goes to the location *Bid*. At this location, the *cargo* informs the selected *truck* with acceptance of the proposal. In addition, the *cargo* will exclude the remaining truck by sending *Reject-proposal*. After that, the control goes to the location *Wait-arrive* at which the *cargo* waits for an incoming report from the selected *truck*, which is responsible for the transportation process. If the incoming message was *Done*, the mission of the *cargo* is terminated, but if the message was *Failure*, the control goes to the location *Unsafe*, before its terminated in the location *Terminate*.

Overall Model

The previous MAS scenario consists of several agents that operate concurrently and communicate with each other. A model of hybrid automaton is

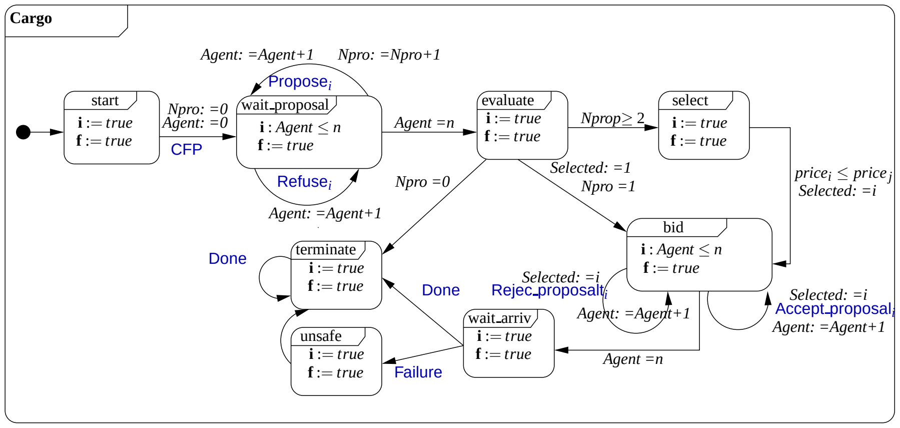


Fig. 3.4. Cargo Automaton.

given to each agent in the MAS, and the communication between those agents occurs by means of shared variables and synchronization labels. Generally, analyzing the behavior of each agent individually, is not sufficient to analyze the entire behavior of the MAS. This is because, in the usual case, each agent coordinates its behavior based on what it receives from other agents. Therefore, we need a way to show how well the entire MAS behaves while performing some tasks. One way to do so is to construct a model which captures all the possible interleaving behaviors of all agents in the MAS. However, constructing that model manually is not an easy work, and it will be difficult to understand the entire behavior of the MAS, especially when the number of agents increases. Fortunately, model-checking tools are helpful in this situation. This is because such tools can automatically construct a model of the entire behavior by means of the parallel composition. In turn, the constructed model can be automatically analyze by asking whether a certain behavior can be reached in it.

3.4 Model Checking Using Hytech

Formal verification provides an effective way to check the correctness of models of systems against certain behaviors. It can determine a design problem of a system, or improve existing one. Currently, one of the most successful techniques used in formal verification is model checking [Clarke et al., 1999]. Generally, model checking allows to verify automatically whether properties can be satisfied in the all possible evolutions of a certain model. In the framework of model checking, both a model together with its specifications should be represented in a suitable textual format to a model checker that checks automatically the satisfiability of requirements within the model.

To verify properties of those systems which can be modeled using hybrid automata, several model checkers are existing. Among of them, we use Hytech [Henzinger et al., 1997] to verify our MAS model. Within Hytech, model checking starts with computing the reachability of the entire state space by getting the set all possible states, which can be reached from an initial state. The resulting set of reachable states will form a base of the model checking; that is, to verify certain property, a traditional way is to check whether the intersection of that property with the reachable states is empty. If so, the property can be reached within the model; otherwise it can not.

Hytech provides a way that aids in design and debugging a system. For example, if a system description contains design parameters, whose values are not specified, then Hytech computes the necessary and sufficient constraints on the parameter values that guarantee correctness of the system. In addition, if a system fails to satisfy a correctness requirement, then Hytech generates an error trajectory, which contains a time stamped sequence of events that leads to a violation of the requirement.

Hytech Code description

To start Hytech the input file representing a model and its properties have be given. Typically, the input file is partitioned into two parts. The first part describes the model, whereas the second part contains a list of iterative analysis commands— For more details about Hytech syntax, see [Henzinger et al., 1995]. The model description is a straightforward textual representation of hybrid automata. Fig. 3.5 shows the Hytech description of the *environment* automaton of Fig. 3.3. The description of the automaton component starts with the declaration of the automaton name, as it is shown in line 1 of Fig. 3.5. Line 2 declares the synchronization labels which will be used to communicate with other automata. Line 3 provides the initial location and the initial conditions on the variables of the automaton. After this, the array of the locations of the automaton has to be defined. Lines 4-7 show the definition of the location *begin*. The definition of location starts with naming the location as it is shown in line 4. The rate conditions, as well as the invariant may also be provided as it is shown in line 5. Each location is associated with a list of transitions originating from it— e.g., line 6,7. Each transition lists a guard condition, synchronization label and a successor location to jump upon the

```
1.automaton Environment
2.synclabs: accept_propos1,accept_propos2,
tobad1,togood1,tobad2,togood2,done;
3.initially begin & envTime=0;
4.loc begin:
5.while True wait {}
6.when True sync accept_propos1 do envTime'=0 goto gcondition;
7.when True sync accept_propos2 do envTime'=0 goto gcondition;

8.loc gcondition:
9.while envTime<=2 wait {}
10.when envTime>=2 sync tobad1 do envTime'=0 goto bcondition;
11.when envTime>=2 sync tobad2 do envTime'=0 goto bcondition;
12.when True sync done goto finish;

13.loc bcondition:
14.while envTime<=5 wait {}
15.when envTime>=5 sync togood1 do envTime'=0 goto gcondition;
16.when  envTime>=5 sync togood2 do envTime'=0 goto gcondition;
17.when True sync done goto finish;

18.loc finish:
19.while True wait {}
20.when True goto finish;
21.end
```

Fig. 3.5. Hytech input code of the environment automaton

satisfaction of the guard condition. Additionally, the transition might update some variables. Line 21, shows the end of the automaton declaration.

Having defined the description of the model of the first part of the input file, the analysis commands of the second part must be given to analyze the behavior of the model. Fig. 3.6 shows the analysis commands of the model. The first line declares three regions variables, namely *ireg*, *freg*, and *reached*. In line 2, the region *ireg* represents the initial state of the whole model; that is the conjunction of initial locations and initial values of the variables of each participating automaton. In line 3, the region *freg* characterizes the states of interest to be (un)reached. In our example, it specifies that either one of the trucks will reach before the deadline. Line 4 assigns to *reached* the set of states reachable from the initial state. The model satisfies the property specified by *freg*, if the intersection between the set of reachable states and the region *freg* is not empty. Lines 5-8 depict this process.

```
var ireg,freg,reached:region;

ireg:=loc[Cargo]=start & loc[Truck1]=idle &loc[Truck2]=idle
& d=1000 & deadline=17 & sagent=0 & npro=0 & envTime=0 &
price1=0 & d1=0 & stime1=0 & tdist1=0 & agent1=0 &
price2=0 & d2=0 & stime2=0 & tdist2=0 &  agent2=0;

freg:= loc[Truck2]=adeadline | loc[Truck1]=adeadline;
reached:=reach forward from ireg endreach;
if empty(reached & freg)
then prints"the truck meets the deadline";
else prints"deadline violation"
endif;
```

Fig. 3.6. Analysis commands in Hytech

Checking Properties

Now after describing the model and the analysis commands, Hytech can be invoked to check the properties of interest within that model. In the following, we present some model checking experiments on our scenario. We have proved various properties, depending on different values of the involved variables in our model. Here, we will focus on some of them.

Reachability of states: One of the properties, which is the general interest of the presented model, is to check the reachability of a certain state. For example, one can check the reachability of the final locations of the automata in the model; that is, reaching the locations *Terminate* in both cargo and trucks. Using Hytech, we can show this by asking if the following region can be reached: $location[truck1] = terminate$ & $location[truck2] = terminate$ & $location[cargo] = terminate$.

Is possible for any truck to perform the transportation ? In our scenario, two trucks are involved in transportation process. We can check that only one truck will be responsible for performing the transportation. Moreover, this truck always provides the minimum price. This can be accomplished by checking if the following can be reached: $location[cargo] = select$

What about Deadlines: One of the most important concernes in the logistics domain is the question whether a deadline can be met or not. Clearly, the question if a truck will arrive before or after a given deadline depends on a number of factors like the condition of the environment during the transportation, the distance to travel, and of course the deadline itself. Using Hytech we did some experiments to answer this question for various values of the deadline as well as the times *gtime* and *btime* of the environment. The speed of the trucks in our experiments lays between 60 and 90, and the total distance that the trucks had to travel, was 1100 and 1150. Given *btime=0* and *gtime=5*, several values for the deadline have been investigated. It turned out that the truck could always reach its destination on time if the deadline was 17 time units, while a deadline of 12 time units was impossible to meet. In order to determine the closest deadline for which the truck was guaranteed to be on the due time, we have used the parametric analysis provided by Hytech, which has yielded 15.55 time units as the closest deadline that could always be met.

Similarly, some experiments have been done to investigate the influence of the environment during the transport. For a given *deadline=17* and *btime=5*. The analysis of Hytech has shown that *gtime* $\geq$*0.888* time units is enough to ensure that the truck will always arrive on time. On the other hand, If *gtime=2*, then *deadline=17* can only be met if *btime* $\leq$*14.33*. It is easy to see that the knowledge of boundaries and dependencies between certain values as we presented above will help both the transport agent and the customer to negotiate a contract that suits both parties.

3.5 Related work

As we said earlier that AUML, as a modeling language, lacks precise semantics. Consequently it does not allow to verify required properties of MASs based on interaction protocols. To overcome this limitation, several works have been proposed. In principle, these works have been devoted to translate AUML models into formal models that can be verified using existing verification tools. For example, Wen and Mizoguchi [1999] have translated a model of protocol based MASs into concurrent finite state machines, which in turn can be verified using SMV model checker [McMillan, 1993]. Similar approach have been presented by Mokhati et al. [2007], who translate AUML models to models that can be verified using Maud model checkers [Eker et al., 2002]. Another approach of translations has been presented in [Jemni Ben Ayed and Siala, 2008]. In this approach, an MAS interaction protocol is initially modeled using the AUML protocol diagram. Then, the model is translated into a model of Event-B formal specification language [Abrial, 2009]. The resulting model is enriched with required properties to be verified using a B-tool B4free [Cansell et al., 2004].

In contrast to hybrid automata, AUML abstract the behavior of agents in a very restrictive way, such that it can not specify the internal behavior of the agents. Additionally, AUML is far away from modeling, and verifying real time properties of MASs as it focuses only to describe the discrete behavior of the interaction among agents.

Petri nets [James, 1977] are well known forms of states based transition systems which have been originally devoted to model and analyze discrete concurrent systems. In addition to their formal and precise semantics to handle concurrency and synchronization, petri nets provide, like variants of finite state machines, a graphical representation of the underlying physical systems. To support modeling and formal analysis of systems using petri nets, a variety of tools are existing. Accordingly, several works, for example [Celaya et al., 2009; Chainbi, 2004], have approached petri nets to formal model MASs. Among of these works, there are proposed works, which give AUML formal semantics by translating the AUML models into petri nets—see for example [Cabac and Moldt, 2004]. However, the traditional petri nets can not specify the continuous behaviors of their underlying systems. To overcome this limitation, various extensions have been proposed to integrate the continuous behaviors of systems within petri nets. For example, David [1997] presents a framework, called hybrid petri nets, which are used to formal specify the behaviors of hybrid systems. Timed petri nets [Wang, 1998] is another example of these extensions in which the formal semantics of the classical petri nets are augmented with real time constraints. Broadly speaking, these continuous or real time extensions are useful and powerful formalisms to model and verify concurrent continuous systems, and hence can be used to model MASs like our adopted approach. However, hybrid/timed petri nets lack of support rigorous analysis for real time requirements[1] of their underlying systems. Therefore, several works have proposed to translate hybrid/timed petri nets into hybrid or timed automata [Cassez and Roux, 2006; Ghomri and Alla, 2007]. Hence, one can use the powerful of the existing tools of hybrid/timed automata to analyze the behavior of the underlying systems. Obviously, this shows that the direct use of hybrid automata to model MASs has advantages over the use of hybrid/petri nets.

[1] more details about real time requirements will be discussed in Chapter 5

Part II

A Noval Framework

4 The Model

We have shown that hybrid automata offer a method to model and to verify the behavior of Multi-agent Systems (MASs). The main challenge to specify and verify MASs with hybrid automata is the state space problem, which occurs due to the construction of parallel composition as well as the infinite state space representation of the behaviors of agents. This asks for a method that can simplify this type of problems. This Chapter provides a convenient way to cope with the state space problem by constructing the composition dynamically, i.e. during the verification phase, and by representing the infinite states space symbolically. The main core of this Chapter has been published in [Mohammed and Furbach, 2010b; Mohammed et al., 2010].

4.1 Introduction

In Chapter 3, we have demonstrated how to use hybrid automata as a framework to formally specify and automatically verify the behavior of MASs by means of model checking. A team of agents is described as concurrent automata combined via parallel composition into a global automaton responsible for coordinating the behaviors of the team to reach a common goal. The automatic verification of MASs' behaviors, however, suffers from the potential combinatorial explosion of the state space caused by parallel composition. This state space problem is one of the primary challenges in applying model checking to analyze concurrent systems. The state space of the parallel composition of an agent with K_1 states and an agent with K_2 states leads to a state space of $K_1 \times K_2$ states. Accordingly, the parallel composition of N agents, each with a state space of K states, leads to a state space with K^N states. This asks for a method that deals with the parallel composition conveniently. In the framework of hybrid automata, the continuous dynamics add another dimension to the state space problem. In particular, model checkers not only keep track of the part of the state space, but also of the timing and continuous evolution associated with each state. This continuous evolution leads to infinitely reached states that should be managed by model checkers. It turns out that a symbolic technique is needed to finitely handle these infinitely states space. In such symbolic techniques, the states are properly encoded using clever data structures that provide compact representation of large state spaces and allow their efficient manipulation.

When constructing the parallel composition of automata, the possible interleaved locations and transitions are enumerated first, and then the composed automata are given to a model checker which checks the properties of interest by exploring the state space of the composed automaton. During this verification process, the model checker symbolically enumerates the possibly reached state spaces and leaves out the unreached states that have been formed as a result of the composition process. In the latter case, the model checker checks for any incoherent constraints appearing during the exploration of the composed automata. Once we have better knowledge which allows us not to represent those unreachable constraints in advance, then the representation can be reduced significantly. This is the key idea of what is the so-called *on-the-fly* construction of the state space representation [Bouajjani et al., 1997]. This helps relieve the state space in a sense that the only possibly reached states will be activated during the run of systems, instead of checking whether the states are reached or not. In this way, the unreached parts of the state space are removed before the a system under consideration is subjected to the verification process. Applying the on-the-fly construction and symbolic methods of state space in model checking have been proven useful in practice to tackle the state explosion problem [McMillan, 1993].

This chapter provides an approach based on hybrid automata which conveniently allows us to specify and verify MASs. The construction of the parallel composition in this approach is built *on-the-fly*. We use Constraint Logic programming (CLP) to implement this approach. The key advantage to use constraints is that they are effective data structures that can implicitly represent the infinite sets as mathematical relations. In this sense, our approach is a symbolic representation, which helps to relieve the state space problem.

The rest of this chapter is organized as follows: Sec.4.2 shows the influence of on-the-fly composition regarding the state space by demonstrating a simple example,. Sec.4.3 shows the basic syntax and the semantics of hybrid

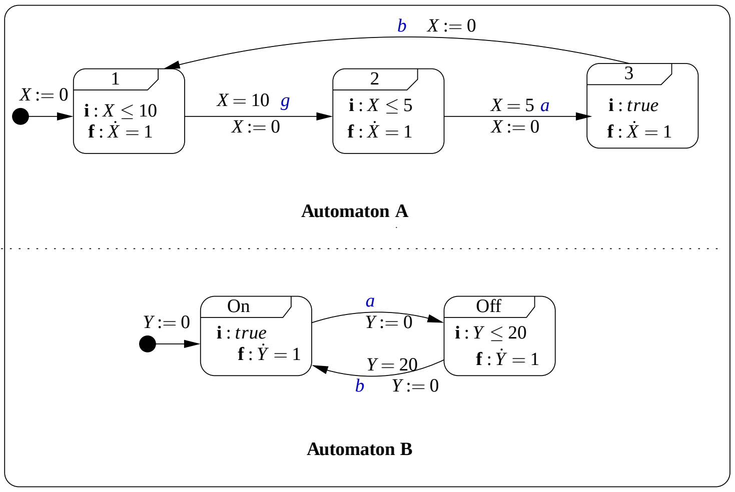


Fig. 4.1. Example of concurrent automata.

state machines that constitute our proposed approach. Sec.4.4 shows how to build an executable model of the presented approach by means of CLP.

4.2 Illustrative Example

Fig. 4.1 shows an example that demonstrates the benefit of the *on-the-fly* approach. Assume that *A* and *B* are two simple hybrid automata interacting my means of shared events, namely $\{a,b\}$. The automaton *A* contains an extra event *g* which might communicate with any other automaton or indicate that the transition between location *1* and location *2* is fired. The behavior of *A* starts at location *1*. After 10 seconds, the control has to jump to location *2*, and then the event *g* must be fired. At location *2*, *A* has to wait up to 5 seconds, then a transition to location *3* must be fired, causing the event *a*. At location *3*, the automaton has to wait until the concurrence of event *b* has occurred. In this case, the control jumps back to location *1*. For automaton *B*, the following behavior is specified. Initially, *B* has to wait at location *On* until the event *a* occurs. If it occurs, the control of *B* goes to location *Off*. At the latter location, *B* has to wait up to 20 seconds before the control goes back to location *On* with firing the event *b*.

In the parallel composition of *A* and *B*, the two automata are synchronized by shared events, that is any shared event can only be executed if the two automata can execute it simultaneously. Private/unshared events of each automaton, e.g. the event *g* in the automaton *A*, are not subject to such previous constraint, but those events can be executed whenever possible. The constructed parallel composition is shown in Fig. 4.2 where the invariance and time constraints in each composed location are defined by the conjunction of the invariance and time constraints of each simple location. Global state-space representations are constructed without regard to whether the states are reachable or not. A model checker usually performs the reachability based on the constraints which it faces during the states' exploration.

The composed automaton of Fig. 4.2 has $3 \times 2 = 6$ locations, but this leaves the location *(3,On)* isolated. If one takes into consideration those constraints which occur inside the locations and transitions, e.g. constraints on events, then one can show that further reduction can be achieved. Since *A* and *B* must be synchronized with their joint event *a*, then no legal transition between the starting location *(1,On)* and the location *(1,Off)* can occur. The location *(1,Off)* will not be explored during the reachability analysis. Consequently, the location *(2,Off)*, reached from *(1,Off)*, will not be reached. In this way, one can show that the exact reached locations are *(1,On)*,*(2,On)* and *(3,Off)*, as shown in Fig. 4.3.

One can see in the previous example that the global state-space representations are constructed without regard to whether the states are reachable or not. In the following, we present an approach aiming at constructing the state space during the execution of a concurrent MAS. In this approach we precisely explore the possibly reached states and avoid any unreached states which may appear due to the traditional composition process. We use mathematical intervals to represent the infinite states raised by the continuous evolution of the real variables.

4.3 Hybrid State Machines

In this section, we show the basic syntax and the semantics of hybrid state machines[1] that constitute our approach. But first we will introduce an illustrative running example throughout this chapter.

4.3.1 Running Example

A train gate controller [Henzinger et al., 2000] is a reactive multi-agent system consisting of three agent components: the *train*, the *gate*, and the *controller*. A road crosses tracks of trains and is guarded by a gate that must be closed or opened upon approaching or leaving of a train respectively. The

[1] In this chapter, the term hybrid state machines and hybrid automata are used synonymously

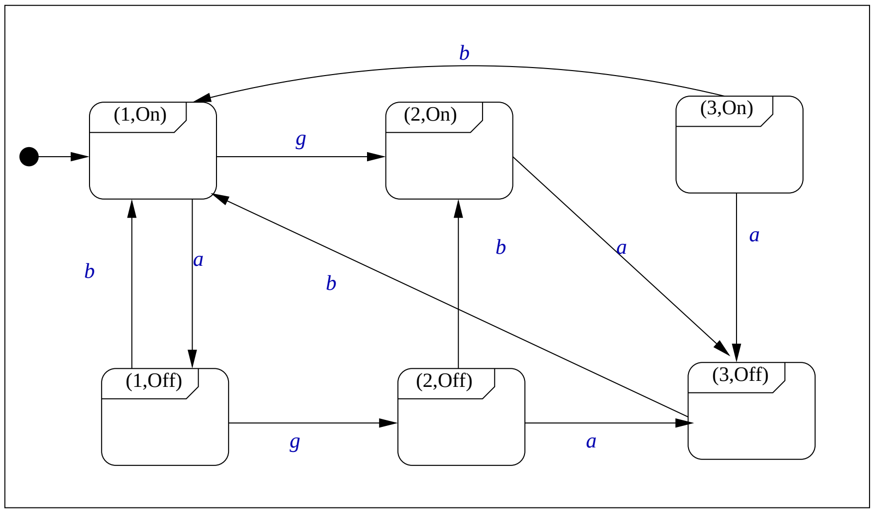


Fig. 4.2. The parallel composition of Fig. 4.1 as a black-box.

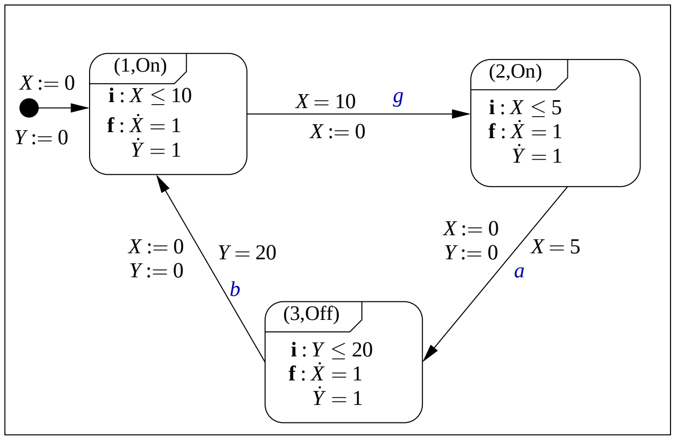


Fig. 4.3. Exact composed automaton of Fig. 4.1

gate is supervised by a controller that has the task to receive signals from the train and to issue lower or raise signals to the gate. A train is initially at a distance of 1000 meters away from the gate and moves at a speed 50 meter per second. A sensor located at 500 meters on the tracks detects the train sending a signal *app* to the controller. The train slows down, following the differential equation $\dot{x} = -\frac{x}{25} - 30$. After a delay of five seconds modeled by the variable t, the controller sends the signal *lower* to the gate, which in turn begins to descend from 90 degrees to 0 degrees at a rate of -20 degrees per second. After

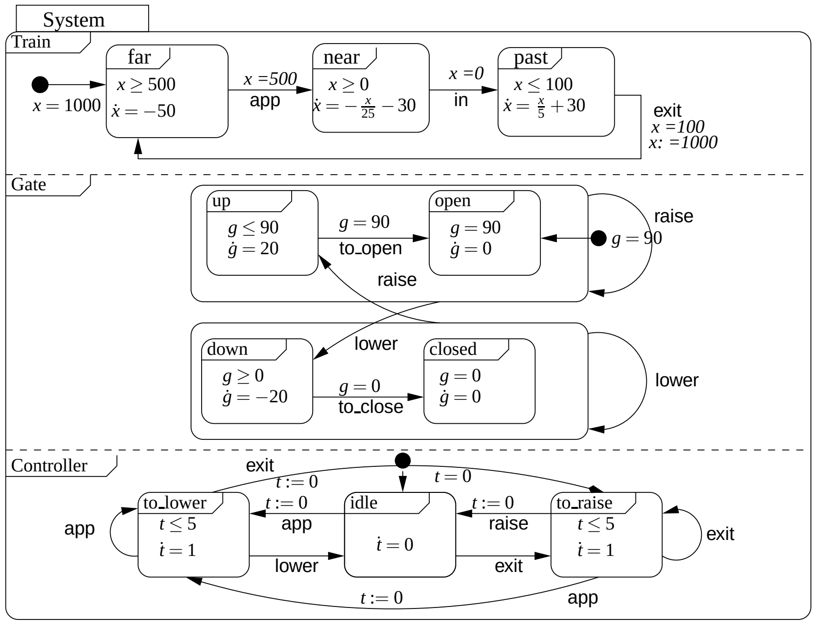


Fig. 4.4. Specification of the train example as hybrid state machines.

crossing the gate, the train accelerates according to the differential equation $\dot{x} = \frac{x}{5} + 30$. A second sensor placed 100 meters past the crossing detects the leaving train, sending a signal *exit* to the controller. After five seconds, the controller raises the gate.

The specification of the previous multi-agent system is graphically illustrated as concurrent hybrid automata in Fig. 4.4. The variable x represents the distance of the train from the gate. The variable t represents the delay time of the controller, while the position of the gate in radius degrees is represented by the variable g.

4.3.2 Syntax

Before we proceed in defining the syntax and semantics of our hybrid automata, we first need to define those constraints which may appear as guards on transitions and invariants inside locations of hybrid automata. We additionally need to define the constraints which define the possible dynamics in our model.

Definition 4.3.1 (Linear Constraints) *Let $\mathbb{X}$ be set of n real variables and $\omega = \sum_{i=1}^{n} a_i \cdot x_i$, with $x_i \in \mathbb{X}$, be a linear combination of variables from $\mathbb{X}$. A* *set $\Phi(\mathbb{X})$ of linear constraints over $\mathbb{X}$, with a typical elements φ, is defined by the following syntax:*

$$\varphi ::= \ \omega \sim b \mid \varphi_1 \wedge \varphi_2 \mid true$$

where $1 \leq i \leq n, a_i,$ and $b \in \mathbb{R}$, $\sim \in \{<, \leq, =, >, \geq\}$, and φ_1, $\varphi_2 \in \Phi(\mathbb{X})$.

The continuous behaviors of hybrid automata show how physical quantities, e.g. position, temperature and humidity, evolve with respect to time. Those behaviors are usually described by differential equations whose solutions can be described as continuous functions in time. In the following, we define the basic constraints that constitute the continuous dynamics of the variables.

Definition 4.3.2 (Dynamical Constraints) *Let $\mathbb{X}$ be a set of n real variables, with a typical element $x \in \mathbb{X}$, and $\dot{\mathbb{X}}$ be set of first derivatives of the variables of $\mathbb{X}$ with a typical element $\dot{x} \in \dot{\mathbb{X}}$. A set $\mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ of dynamical constraints over $\mathbb{X} \cup \dot{\mathbb{X}}$ with typical element d, is defined inductively by the following syntax:*

$$d ::= \ \dot{x} \sim c \mid \dot{x} + a \cdot x = c \mid d_1 \wedge d_2 \mid true$$

where $a \neq 0, c \in \mathbb{R}$, $\sim \in \{=, \leq, \geq\}$, d_1, $d_2 \in \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$.

Having defined the linear and dynamical constraints, we are ready to introduce the syntax of a hybrid state machine, i.e. their structural components.

Definition 4.3.3 (basic components) *A hybrid state machine is a tuple $H = (Q, \mathbb{X}, Inv, Flow, E, Jump, Reset, Event, Event_H, q_0, v_0)$ where:*

- *Q is a finite set of control locations, which defines the possible locations of the state machine.*

- *$\mathbb{X}$ is an ordered set of n real variables.*

- *$Inv : Q \rightarrow \Phi(\mathbb{X})$ is a function that assigns a linear constraint $Inv(q)$ to each location $q \in Q$.*

- *$Flow : Q \rightarrow \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ is a function that assigns a dynamical constraints $Flow(q)$ to each control location $q \in Q$.*

- *$E \subseteq Q \times Q$ is a finite set of transitions among the control locations.*

- *$Jump : E \to \Phi(\mathbb{X})$ is a function that assigns to each transition $e \in E$ a constraints $jump(e)$, which must hold to fire e.*

- *$Reset : E \times \mathbb{X} \to \mathbb{R}$ is a mapping, which assigns a real value to each variable on each transition $e \in E$. A reset of a variable $x \in \mathbb{X}$ on a transition $e \in E$ is denoted as $x := Reset(e,x)$. Conveniently, we write $= Reset(e,\mathbb{X})$ to denote the reset all variables.*

- *$Event_H$ is a finite set of events.*

- *$Event : E \to Event_H$ is a function that assigns an event to each transition $e \in E$ from a set of events $Event_H$.*

- *$q_0 \in Q$ defines the initial location of the automaton.*

- *v_0 defines the initial values of the variables $\mathbb{X}$.*

The previous structure permits the existence of events on each transition $e \in E$. An event is meant to serve as communication between different automata or to denote a change in the internal behavior of an automaton. Thus, we can consider that this set of events is classified into two disjoint sets. The first set contains the events that are used as communication messages among automata, whereas the other set contains the set of events that are used to describe the internal observation of the automaton. The latter set of events can be used to reason about the observational behavior of automata, which is required in the case of MASs.

As shown in Chapter 2, a hybrid automaton with linear constraints on guards and invariants is classified according to the continuous flow into timed, linear hybrid, rectangular hybrid or non-linear hybrid automata. In our approach, this type of classification depends on the choice of the dynamical constraints.

4.3.3 Semantics

Having described the internal structure of a hybrid automaton, we will discuss the semantics of its intended behaviors. A hybrid automaton can exactly be in one of its control locations at each stage of its computation. But knowing the present control location is not enough to determine which of the outgoing transitions can be taken next, at all. A snapshot of the current state of the computation should also keep in mind the present valuation of the continuous variables. To begin formalizing the semantics of a hybrid automaton, we need to define the concept of a *state* and to show how control evolves from one state to another. But first we need to define how continuous variables evolve.

Definition 4.3.4 (Evaluation of Linear Constraints) *Let $\varphi \in \phi(\mathbb{X})$ be a constraints and $v \in \mathbb{R}^n$ be the valuation of the variables $\mathbb{X}$, then we write*

$$v \models \varphi,$$

if v satisfies the constraint φ, which is defined inductively as

$$\begin{array}{lll} \varphi = true. & & \\ \varphi = \sum_{i=1}^{n} a_i \cdot x_i \sim c & iff & \sum_{i=1}^{n} a_i \cdot v_i \sim c \;\; holds. \\ \varphi_1 \wedge \varphi_2 & iff & v \models \varphi_1 \text{ and } v \models \varphi_2. \end{array}$$

where v_i is the valuation of the ith components of v

Definition 4.3.5 (Evaluation of Dynamical Constraints) *Let $d \in \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$ be a dynamical constriants and $f : \mathbb{R}^{\geq 0} \to \mathbb{R}^n$ be a differentiable function, then we write*

$$f \vDash_* d$$

if f satisfies the dynamical constraint d, which is defined inductively as

$$\begin{array}{lll} d = true. & & \\ d = \dot{x} \sim c & iff & f'(t) \sim c \;\; holds. \\ d = \dot{x} + a \cdot x = c & iff & f'(t) + a \cdot f(t) \sim c \;\; holds. \\ d = d_1 \wedge d_2 & iff & f \vDash_* d_1 \text{ and } f \vDash_* d_2. \end{array}$$

where $f'(t)$ is the differentiation of the function f for $t \in \mathbb{R}^{\geq 0}$.

Definition 4.3.6 (State) *At any instant of time $t \in \mathbb{R}^{\geq 0}$, a state of a hybrid automaton is given by $\sigma_i = \langle q_i, v, t \rangle$, where $q_i \in Q$ is a control location, v is the valuation of the real variables. A state $\sigma_i = \langle q_i, v, t \rangle$ is admissible iff $v \models Inv(q_i)$.*

The state transition system of a hybrid automaton H starts with the *initial state* $\sigma_0 = \langle q_0, v_0, 0 \rangle$, where the q_0 and v_0 are the initial location and valuations of the variables respectively. For example, the initial state of the *Train* automaton of Fig. 4.4 can be specified as $\langle far, x = 1000, 0 \rangle$.

The semantics of a hybrid automaton is defined in terms of a labeled transition system between states. Transitions between states are generally categorized into two kinds of transitions: continuous transitions, capturing the continuous evolution of states, and discrete transitions, capturing the changes of location. We will define the semantics of hybrid automaton more formally.

Definition 4.3.7 (Operational Semantics) *A transition rule between two admissible states $\sigma_1 = \langle q_1, v_1, t_1 \rangle$ and $\sigma_2 = \langle q_2, v_2, t_2 \rangle$ is*

Discrete transition *iff $e = (q_1, q_2) \in E$, $t_1 = t_2$ and $v_1 \models Jump(e)$, and $v_2 \models Inv(q_2)$, such that v_2 is the valuations coming from $Reset(e, \mathbb{X})$. In this case an event $a \in Event_H$ occurs. Conventionally, we write this as $\sigma_1 \xrightarrow[t_1]{a} \sigma_2$.*

Continuous(Delay) transition *iff $q_1 = q_2$, $(t_2 - t_1) > 0$ is the duration of time passed at location q_1, there exists a differentiable function f with $f \vDash_* Flow(q_1)$ and $f(t_1) = v_1$ and $f(t_2) = v_2$, and for all $t \in [t_1, t_2]$, $f(t) \models Inv(q1)$.*

In the previous definition, v_2 results from resetting variables on a transition in case of the discrete transition rule, while it results from the continuous evolution of the variables in case of the continuous transition rule. An execution of a hybrid automaton corresponds to a sequence of transitions from onestate to another. For this purpose, we define the valid run as follows:

Definition 4.3.8 (Run: micro level) *A path $\rho = \sigma_1 \sigma_2 \sigma_3, \ldots,$ of a hybrid automaton H is a finite or infinite sequence of admissible states, where the transition from a state σ_i to a state σ_{i+1}, for all $i \geq 1$, is related either by a discrete or continuous transition. A set of all possible paths of A is denoted as $\Pi(H)$. A run of H is a path ρ starting with the initial state σ_0.*

It should be noted that the continuous change of states in a path ρ generates an infinite number of reachable states. Therefore, state-space exploration techniques require a symbolic representation way for representing these infinite states appropriately. One way to do so is to use mathematical intervals. We call this symbolic mathematical interval *region*, which is defined as follows:

Definition 4.3.9 (Region) *Given a path $\rho \in \Pi(H)$, a sub-sequence of admissible states $\Gamma = (\sigma_{i+1} \cdots \sigma_{i+m}) \subseteq \rho$ is called a region, if for all states σ_{i+j} with $1 \leq j \leq m$, it holds $q_{i+j} = q$ and for the states σ_i and σ_{i+m+1} with respective locations q_i and q_{i+m+1}, then it must hold $q_i \neq q$ and $q_{i+m+1} \neq q$. Conventionally, a region Γ is written as $\Gamma = \langle q, V, T \rangle$, where $t_{i+1} \leq T \leq t_{i+m}$ is the interval of continuous time, and V is the tuple of intervals valuations of the variables during the time interval T.*

In the previous definition it should be noted that a region Γ is always admissible since it is a sub-sequence of a run ρ. Γ captures the possible states that can be reached using continuous transitions in each location $q \in Q$. T represents the continuously reached time. A region captures the continuous values for each variable $x_i \in \mathbb{X}$. These continuous values can be represented as an interval V of real values. Let us consider the automaton *train* of Fig. 4.4. The region obtained from the location *far* can be described as $\Gamma = \langle far, X, T \rangle$, where $500 \leq X \leq 1000$, and $0 \leq T \leq 10$.

A run of a hybrid automaton can be re-phrased in terms of reached regions, where the change from one region to another is fired by using a discrete step.

Definition 4.3.10 (Run: macro level) *A run of hybrid automaton H is $\rho_H = \Gamma_0, a_1, \Gamma_1, a_2, \ldots$, a sequence of (possibly infinite) regions, where a transition from a region Γ_i to a region Γ_{i+1}—written as $\Gamma_i \xrightarrow[t_{i+1}]{a_{i+1}} \Gamma_{i+1}$—is enabled, if there is $\sigma_i \xrightarrow[t_{i+1}]{a_{i+1}} \sigma_{i+1}$, where $\sigma_i \in \Gamma_i$, $\sigma_{i+1} \in \Gamma_{i+1}$ and $a_{i+1} \in Event$ is the generated event before the control goes to the region Γ_{i+1}. Γ_0 is the initial region obtained from a start state σ_0 by means of continuous transitions.*

The operational semantics are the basis for verification of a hybrid automaton. In particular, model checking of a hybrid automaton is defined as the reachability analysis of its underlying transition system. The most useful question to ask about hybrid automata is the reachability of a given state. We define the reachability of a region and state as follows.

Definition 4.3.11 (Reachability) *A region Γ_i is called reachable in a run ρ_H, if $\Gamma_i \in \rho_H$. Consequently, a state σ_j is called reachable, if there is a reached region Γ_i such that $\sigma_j \in \Gamma_i$*

Reachability analysis computes all the states that are connected to the initial states by a run. The classical method to compute the reachable states consists of performing a state-space exploration of a system, starting from the initial region and spreading the reachability information along control locations and transitions until fixed regions can be reached. Fig. 4.5 is a simple semi-decision algorithm, which computes the reached regions of a given hybrid automaton. In this algorithm, let *post(R)* be the set of all reached regions connected to the region R with a discrete step, given an initial region Γ_0.

4.3.4 Hybrid State Machines Composition

For the specification of complex systems, we extend hybrid automata by parallel composition. The parallel composition of hybrid automata can be used

```
Wait :=post($\Gamma_0$)
Reached := $\Gamma_0$
while Wait $\neq \emptyset$ do
take R from Wait
if R $\notin$Reached then Reached := Reached $\cup$ R
end if
Wait := Wait $\cup$ (post(R)$\setminus$ Wait)
end while
```

Fig. 4.5. A simple procedure for reachability computation.

to specify larger systems (multi-agent systems), where a hybrid automaton is given for each part of the system and communication between the different parts may occur via shared variables and synchronization labels. It has been previously said that the parallel composition of hybrid automata is technically obtained from the different parts using a product construction of the participating automata. The transitions from the different automata are interleaved, unless they share the same synchronization label. In this case, they are synchronized on transitions. As a result of the parallel composition, a new automaton called composed automaton is created which captures the behavior of the entire system. The composed automaton is, in turn, given to a model checker that checks the reachability of a certain state. The composition of hybrid automata H_1 and H_2 can be defined in terms of synchronized or interleaved regions of the regions produced from run of both H_1 and H_2. As a result of the composition procedure, compound regions are constructed, which consist of a conjunction of a region $\Gamma_1 = \langle q_1, V_1, T\rangle$ from H_1 and another region $\Gamma_2 = \langle q_2, V_2, T\rangle$ from H_2. Therefore, each compound region takes the form $\Lambda = \langle (q_1, V_1), (q_2, V_2), T\rangle$ (shortly written as $\Lambda = \langle \Gamma_1, \Gamma_2, T\rangle$), which represents the reached region at both control locations q_1 and q_2 the during a time interval T.

Definition 4.3.12 (Run Composition) *A run of composed automata is the sequence $\sum_{H_1 \circ H_2} = \Lambda_0, a_1, \Lambda_1, a_2, \ldots$ of compound regions, where a transition between compound regions $\Lambda_1 = \langle \Gamma_1, \gamma_1, T_1\rangle$ and $\Lambda_2 = \langle \Gamma_2, \gamma_2, T_2\rangle$ (written as $\Lambda_1 \xrightarrow[t]{a} \Lambda_2$) is enabled, if one of the following holds:*

- *$a \in Event_{H_1} \cap Event_{H_2}$ is a joint event, $\Gamma_1 \xrightarrow[t]{a} \Gamma_2$, and $\gamma_1 \xrightarrow[t]{a} \gamma_2$. In this case , we say that the region Γ_1 is synchronized with the region γ_1.*
- *$a \in Event_{H_1} \setminus Event_{H_2}$ (respectively $a \in Event_{H_2} \setminus Event_{H_1}$), $\Gamma_1 \xrightarrow[t]{a} \Gamma_2$ and $\gamma_1 \rightarrow \gamma_2$, such that both γ_1 and γ_2 have the same control location—i.e. they relate to each other using a continuous transition.*

To illustrate the previous procedure, consider the train gate controller example. There is a synchronized region between the region obtained from location *far* and from location *idle* in the automata *train* and *controller* respectively. Both regions are synchronized using the joint event *app*. Therefore, the synchronized region can be described as $\langle (far, X), (idle, T_1), T \rangle \xrightarrow[t]{app} \langle (near, X), (to\_lower, T_1), TT \rangle$, where X and T_1 are the continuous valuations of the variable of the automata *train* and *controller* respectively. On the other hand, the region obtained from location *far* and from location *open*, in the automata *train* and *gate* respectively, relates to each other using disjoint event *app*. Therefore the obtained region can be described as $\langle (far, X), (open, G), T \rangle \xrightarrow[t]{app} \langle (near, X), (open, G), TT \rangle$.

The previous procedures give the possibility to construct the composition dynamically during the run/verification phase. As it has been said, computing the composition in such a way is obviously advantageous. This is why only the active parts of the state space will be taken into consideration during the run instead of producing the composition procedure prior to the verification phase. This can relieve the state space problem raised by modeling MASs.

4.4 Constraint-Based Modeling

In [Mohammed and Furbach, 2009b] we showed how to encode the syntax and semantics of hybrid automata, previously described as a constraint logic program (CLP) [Jaffar and Lassez, 1987]. A primary version of this model has been presented in [Mohammed and Furbach, 2008b], as well as in [Mohammed and Stolzenburg, 2008]. There are diverse motives beyond choosing *CLP* as a modeling prototype to implement the framework. Firstly, hybrid automata can be described as a constraint system where the constraints represent the possible flows, invariants, and transitions. Secondly, constraints can be used to characterize certain parts of the state space, e.g. the initial state or a set of unsafe state. Further, there are close similarities in operational semantics between *CLP* and hybrid automata. State transition systems can be ideally represented as a logic program in which the set of reachable states can be computed. Moreover, constraints enable us to represent infinite states symbolically as a finite interval. The infinite states, for instance, can be handled efficiently as an interval constraint that bounds the set of infinite reachable state as a finite interval (i.e., $0 \leq X \leq 250$). A constraint solver can be used to reason about the reachability of a particular state inside this interval. A further motivation to choose *CLP* is its enrichment with many efficient constraint solvers of various domains. *CLP* contains a constraint solver over real interval constraints, which can be used to represent the continuous flows as constraint relations to the time, as well as to reason about a particular valuation. *CLP* also contains a constraint solver over symbolic domains, which are appropriate to represent the synchronization events (communication messages) among agents. Last but not least, by employing *CLP* the composition of automata can be constructed on the fly (during models checking). This can be done by investigating the constraints appeared during running models. The previous can relieve the state space problem raised from specifying MAS.

Let us first look at a preliminary introduction to CLP, before we show how to encode the syntax and semantics of our hybrid state machines in terms of CLP

4.4.1 Overview of Constraint Logic Programming

Constraint Logic Programming (CLP) [Jaffar and Lassez, 1987] has been introduced as an extension to logic programming where unification, the basic operation of logic programming, is replaced by constraint handling in a constraint system. The resulting languages combine the advantages of logic programming with the efficiency of constraint solving algorithms.

Constraints are relations which should hold among variables of a problem and thei domains of values. A general purpose constraint solver is used hereafter to solve such constraints. A constraint solver implements an algorithm for solving allowed constraints in accordance with the constraint theory. The solver collects the constraints that arrive incrementally from a running model. It puts them into a data structure for the constraints which is called constraint store. During the previous process, the solver tests the satisfiability of the constraints, or simplifies them.

A program in CLP typically consists of three sections. Firstly, the declaration of the domains of program variables. Secondly, constraints are stated that are used to build a constraint network at run time. These constraints possibly involv multiple constraint solvers. The last program section defines in which order program variables are assigned values that are consistent with constraints, and in which order those values are tried

Currently there are many CLP languages. The domain of constraints is one of the key point of creating such languages. Prolog II [Colmeraue, 1984], is generally considered as the first CLP language. The constraints of Prolog II are equations and dis-equalities over terms. The next generation of CLP languages, Prolog III [Colmerauer, 1990], CHIP [Dincbas et al., 1988] and CLP(R)[Jaffar et al., 1992], went a step further by introducing constraints over new computation domains including rational and real numbers, integers, Boolean and lists. A good survey about the development of CLP languages can be found in [Rossi, 2000].

ECL^iPS^e Prolog [Apt and Wallace, 2007] and SWI Prolog [Wielemaker, 2008] are among the successful platforms which support the integration of various constraint solvers, including constraints over finite domains and constraints over continuous intervals.

4.4.2 Hybrid Automata in CLP

We use ECL^iPS^e Prolog [Apt and Wallace, 2007] to implement our prototype. ECL^iPS^e includes the *ic* library for interval constraints, as well as finite domain constraint solving. The prototype follows the definitions of both the formal syntax and semantics of hybrid automata, which are defined in the previous section. To start implementing a hybrid state machine, we primarily begin by modeling the locations and their constraints (e.g. flows, invariants), which are modeled as the predicate *automaton* as follows:

```
%%% automaton(+Location,?Vars,+Vars0,+T0,?Time)
%%% models invariant and flow inside location
automaton(Location,Vars,Vars0,T0,Time):-
        Flow(Vars),
        Inv(Vars),Time $>=T0.
```

automaton in the previous predicate indicates the name of the automaton, and *Location* represents the actual name of the current locations of the automaton. *Vars* is a list of real variables belonging to in the automaton, whereas *Vars*0 is a list of the corresponding initial values. $Inv(Vars)$ is the list of invariant constraint on *Vars* inside the location. The constraint predicate $Flow(vars)$ models the continuous flows of the variables *Vars* with respect to time $T0$ and *Time*, given initial values *Vars*0 of the variables *Vars* at the start of the flow. $T0$ is the initial time at the start of the continuous flow. As presented in Sec.4.3.2, a hybrid automaton is classified according to the constraints on the continuous flow. $Flow(Vars)$ is represented in terms of constraints as $Vars = Var0 + c \cdot (Time - T0)$ in case of a linear hybrid automaton, as $Var0 + c \cdot (Time - T0) \leq Vars \leq Var0 + c \cdot (Time - T0)$ in case of a rectangular hybrid automaton, and as $Vars = -c2/c1 + (Var0 + c2/c1) \cdot \exp(c1 \cdot (Time - T0))$ in case of a non-linear hybrid automaton. $(Time - T0)$ models the delay inside the location. It should be noted that after executing the predicate *automaton*, *Vars* and $Time$ holds the reached valuations of the variables together with the reached time respectively. The following is an example showing the concrete implementation of the location *far* in the automaton *train* Fig. 4.4 [2]. The $ symbol in front of the (in)equalities is the constraint relation for interval arithmetic constraints (library *ic* in ECLiPSe Prolog).

```
train(far,[X],[X0],T0,Time):-
     X $= X0-50*(Time-T0),
     X $>=500, Time $>=T0.
```

According to operational semantics defined in Def. 4.3.7, a hybrid automaton has two kinds of transitions: *continuous* transitions capturing the continuous evolution of variables, and *discrete* transitions capturing the changes of location. We encode transition systems into the predicate *evolve*, which alternates the automaton between a discrete and a continuous transition. The automaton evolves with either discrete or continuous transitions according to the constraints appearing during the run.

```
%%% evolve(+Automaton,+State,-Nextstate,+T0,-Time,?Event)
evolve(Automaton,(L1,Var1),(L2,Var2),T0,Time,Event) :-
     continuous(Automaton,(L1,Var1),(L1,Var2),T0,Time,Event);
     discrete(Automaton,(L1,Var1),(L2,Var2),T0,Time,Event).
```

When a *discrete* transition occurs, it gives rise to updating the initial variables from $Var1$ into $Var2$, where $Var1$ and $Var2$ are the initial variables of locations $L1$ and $L2$ respectively. Otherwise, a delay transition is taken using the predicate *continuous*. It is worth noting that there are infinite states due to the continuous progress. However, this can be handled efficiently as an interval constraint that bounds the set of infinite reachable state as a finite interval (i.e., $0 \leq X \leq 250$).

In addition to the variables, each automaton is supplied with a set of events called $Event_{Automaton}$. An example of this set of events of the automaton *train* is denoted as $\{app, in, exist\}$. Each transition is augmented with the variable *Event*, which is used to define the parallel composition from the automata individuals sharing the same event. The variable *Event* ranges over symbolic domains and guarantees that whenever an automaton generates an event, the corresponding synchronized automata have to be taken into consideration simultaneously. It should be mentioned that the declaration of automata events must be provided in the modeling example. The declaration of the possible events domain of Fig. 4.4. is coded as follows :

[2] The full implementation is in the appendix A

```
:- local domain(events(app,in,exit,raise,lower, to_open)).
```

This means that the domains of events are declared symbolically to capture the set of all possible events applicable to the underlying modeled system.

Once the domain of events has been defined, an appropriate solver of a symbolic domain deals with any defined constraints in terms of the declared domains. After defining the domains of events, a variable of type *events* can be declared as follow:

```
  Event &:: events, Event &= domain_value.
```

The variable *Event* is declared with domain values defined by *events*, and is initialized with a specific value from its domain. The & symbol is a constraint relation for symbolic domains (library *sd* in ECLiPSe Prolog).

In the following we present the general implementation of the predicate *discrete*, which defines transitions between locations.

```
%%% discrete(+Automaton,+State1,-State2,+IntTime,-Time,-Event)
discrete(Automaton,(Loc1,Var1),(Loc2,Var2),T0,Time,Event):-
         automaton,(Loc1,Var1,Var,T0,Time),
         jump(Var), reset(Var2),
         Event &::events,Event &=domain_value.
```

In the previous predicate, *domain_value* must be a member in $Event_{Automaton}$. When the *discrete* predicate is fired, the automaton generates an event by constraining the variable *Event* to the suitable value from its domain.

In the following we show an instance of the concrete implementation of the *discrete* predicate between *far* and *near* in the *train* automaton.

```
discrete(train,(far,[X0]),(near,[XX0]),T0,Time,Event):-
         train(far,[X0],[X],T0,Time),
         X $=500, XX0 $=X,
         Event &::events, Event &=app.
```

Once the locations and transition rules have been modeled, a state machine needs to be implemented in order to execute the model. Thereof, a driver program is implemented as shown in Fig. 4.6.

The *driver* is a state machine that is responsible to generate and control the behaviors of the concurrent hybrid automata as well as to provide the reachable regions symbolically. The *driver* takes the starting state for each participating automaton, i.e. a control location as input argument as well as

```
%%% driver(+State1,+State2,...,+Staten,+T0,-Regions,
                                          +PastRegion).
%%% perform composition and reachability
driver((L1,Var01),(L2,Var02),...,(Ln,Var0n),T0,[Reg|NxtReg],
                                                PastReg) :-

automaton1(L1,Var1,Var01,T0,Time),
automaton2(L2,Var2,Var02,T0,Time),
... ,
automatonn(Ln,Varn,Var0n,T0,Time),

evolve(automaton1,(L1,Var01),(NxtL1,Nvar01),T0,Time,T,Event),
evolve(automaton2,(L2,Var02),(NxtL2,Nvar02),T0,Time,T,Event),
... ,
evolve(automatonn,(Ln,Var0n),(NxtLn,Nvar0n),T0,Time,T,Event),

\+ member((L1,L2,..,Ln,Var1,Var2,..,Varn,_,Event), PastReg),
Reg = (L1,L2,..,Ln,Var1,Var2,..,Varn,Time,Event),
NpastReg =[Reg|PastReg],

driver((NxtL1,Nvar01),(NxtL2,Nvar02),...,(NxtLn,Nvar0n),T,
                                          NxtReg,NpastReg).
```

Fig. 4.6. A state machine to drive the run of automata.

the list of initial valuations of the variables. In addition, it takes the starting time $T0$ followed by a list of reached regions, which is needed for the purpose of the verification. It should be noted that during the course of the dirver's execution, there is a symbolic domain variable *Event* shared among automata. That variable is used by the appropriate solver to ensure that only one event is generated at a time. In another words, when an automaton generates an event due to a discrete transition of one of the predicates *evolve* of the concurrent automata, the symbolic domain solver will exclude all the domains of values of the other automata that are not coincident with the generated event. This means that only one event is generated at a time. If more than one automaton generate different events at a time, then the symbolic domain solver will handle only one of them at a time, but the other events will be handled using backtracking.

Since each automaton generates an event by a discrete step at the end of its continuous evolution, then the precedence of events that appear during the run is important to both composition and the verification process. An obvious way to deal with this precedence is to use constraints on the time of the generated events. To accomplish this, we constraint the execution of each automaton with a shared variable *Time*. The constraint solver, in turn, binds this variable with the minimum execution time among the automata. It follows that this variable *Time* eventually holds the minimum time needed to generated an event. The previous computation partitions the state space into regions, where the transition from one region to another depends on the minimum time needed to generate an event. This shows how the automata composition can be implicitly constructed efficiently on the fly, i.e. during the computation.

It has been said that we are not only concerned with running and composing the automata, but also with the their verification. For this purpose, the *driver* is supplemented with the list of reached compound regions. At each step of the execution of the *driver*, a compound region, in the form $\langle locations, Variables, Time, Event \rangle$ is added to the list of reached regions. This region symbolically represents the set of reached states and times to each control location as mathematical constrains. Additionally, each region contains the generated event before the control goes to another region using a discrete step. The *driver* technically computes the set of reached regions until fixed regions are obtained. This is computed by investigating— in each iteration of *driver*—if the reached region is not contained in the list of the previously reached regions. For this purpose, the last argument of the *driver* holds for the list of these regions.

Reachable regions should contain only those variables which are important for the verification of a given property. Therefore, the last argument list of the predicate *driver* can be expanded or shrunk as needed to contain the significant variables.

As soon as the *driver* has been built, the complete model should be invoked for the purpose of execution and verification. the following predicate *reachable* is implemented to invoke the *driver*.

```
reachable(Ψ0,Reached) :-
     driver(Ψ0,0,Reached,[]).
```

The first argument of predicate *reachable* is the states predicate Ψ_0 that represents the initial states of the hybrid automata. An example showing how to run the model on the running example Fig. 4.4, takes the form:

```
reachable((far,[1000]),(open,[90]),(idle,[0]),Reached).
```

4.4.3 Model Analysis

Now we have an executable constraint based specification, which can be used to test properties of systems modeled as hybrid automata. Several properties can now be investigated. In particular, one can check properties on states using reachability analysis. The reachability analysis consists of two basic steps. Firstly, computing the state space of the automaton under consideration. In our case, this is done using the predicate *driver*. Secondly, searching for states that satisfy or contradict given properties. During the searching stage, constraint solvers can be ideally can be used to reason about the reachability of those states within regions.

As far as *CLP* is concerned, a state is reached iff the constraint solver succeeds in finding a satisfiable solution for the constraints representing the intended state. For example, an interesting property is to find the shortest distance of the train to the gate before the gate is entirely closed. This can be checked by posing the following query:

```
?- reachable((far,[1000]),(open,[90]),(idle,[0]),Reached),
   member((near,_,_,Time,to_close,_),Reached),
   get_max(Time,Tm),
   member((near,_,_,Tm,_,X),Reached),
   get_min(X,Min).
```

The previous query returns $Min = 104.8$ meters, which is the minimum distance of the train that the model guarantees before the gate is completely closed.

Since the events and time are recorded particularly at reached regions, verifying timing properties or computing the delay between events are further tasks that can be done within the reachability framework. For instance, we can find the maximal time delay between *in* and *exit* events, by stating the following query:

```
?- reachable((far,[1000]),(open,[90]),(idle,[0]),Reached),
      append(A,[(past,_,_,Time1,exit,_)|_],Reached),
      append(B,[(near,_,_,Time2,in,_)|_],A),
      get_max(Time1,Tmax1),get_max(Time2,Tmax2),
      Delay $= Tmax1-Tmax2.
```

The constraint solver answers *yes* and yields $Delay = 2.554$. This value means that the train needs at most 2.554 seconds to be in the critical crossing section before leaving it. Similarly, other timing properties can be verified.

Last but not least, the expressiveness of CLP makes it possible to reason about the reachability of interesting properties not only within some region, but also on the boundary of the region during firing a transition. This type of properties is important in the sense that the values of continuous variables at the cutoff point can tell how well systems perform relative to the given timing constraints, and shows how critical variables should behave. To do so, we get first the occurrence of the event of interest. Then we constrain time of that event by projecting it on the intended critical continuous variables. For instance, suppose we want to find the shortest distance of the train to the gate before the gate is entirely closed. This can be checked by getting the occurrence time of the event *to_close*, then the constraint solver will bind the train distance X to the value of this time.

5

Region Computation Tree Logic:Specification

In Chapter 4, we provided the syntax and semantics of our proposed approach to hybrid automata. We implemented this approach by means of constraint logic programming. We showed how to analyze simple queries by means of reachability analysis. This chapter shows the general structure to specify those properties, which can be verified within our presented approach. In order to specify properties one needs a suitable specification language. Temporal logics are prominent examples of such specification languages. They have been devoted to specify those properties which depend on the order at which states appeared in a particular model, or what is called qualitative properties of the model. When the time constraints are explicitly required in the specification, the ordinary temporal logics have to be refined. Properties that depend on these time constraints are known as quantitative properties. This chapter presents a variance of temporal logics, RCTL (Region Computation Tree Logic) that extends the computation tree logic by incorporating time on states, events, and constraints of formulas. The RCTL formulas are interpreted over the set of possible regions resulted from the run of hybrid automata. The specification language of RCTL allows us to express many properties in a concise and intuitive manner. To bring model checking into the scope of RCTL, we concentrate on the specification of those properties that can be verified using reachability analysis. The main contribution of this chapter has been published in [Mohammed and Furbach, 2010a].

5.1 Introduction

Model checking asks if possible runs of a model satisfy a given property specified in a formal specification language. In fact, the choice of this language is known to be one of the keys issues in the design of model checkers as this language is one of the primary interfaces to access them. One of the most widely used specification languages for many important systems is *temporal logic*, which was introduced in [Pnueli, 1977] as a subclass of model logic [Van Benthem and ter Meulen, 1997] with possible world semantics and model operators: $\Box$ (for all possible worlds) and $\Diamond$ (there exists a possible world). In temporal logic, $\Box$ is interpreted as from now on, at all states (or henceforth, always), while $\Diamond$ is interpreted as from now on, there exists a state (or eventually). Basically, temporal logic comes in two different views: linear temporal logic LTL [Manna and Pnueli, 1992; Pnueli, 1977] and computation tree Logic CTL [Ben-Ari et al., 1983]. These logics allow us to express real-time requirements of reactive systems. The view of a computation is the key difference between LTL and CTL. The computation can be viewed either as a linear sequence with only one future or as a tree with many possible futures. The former belongs to LTL, while the latter belongs to CTL. Thus, CTL provides the branching operators $\exists$ and $\forall$ to specify the relation among futures. Temporal logics basically allow to express the qualitative properties of reactive systems, that is the properties which focus on the temporal order of the occurrence of events. An example of these properties is to specify that a certain property of interest may eventually occur, or in other words the formula is reached in the model. Another example of properties is to specify that a critical property is never reached in the model or what is so-called safety property. These types of properties have been considered in different model checkers SPIN [Holzmann, 1997] and NuSMV [Cimatti et al., 2002].

The classical temporal logics, however, are insufficient to specify quantitative temporal requirements or what is so-called hard real time constraints, which put timing deadlines on the behavior of reactive systems. Let for example a proposition p correspond to the occurrence of the event $event_1$, and q correspond to the occurrence of the event $event_2$, then in linear temporal logic, the formula $\Box(p \rightarrow \Diamond q)$ states that $event_1$ is always followed by $event_2$, but it does not state anything about the time period between the occurrences $event_1$ and $event_2$. Temporal logics should be refined in order to permit such types of quantitative specifications. The quantitative requirements are favored or are mandatory safety in various scenarios. In the logistic transport example, it is not desirable to specify that the goods will eventually reach their final destination, but reaching within reasonable period of time is favorable. In the train gate example, it is mandatory safety to to guarantee that the gate will be closed in certain time limit while appearing of a train.

For their inexpressiveness to specify quantitative properties, the classical temporal logics have to incorporate the notation of time. For this aim, there have been proposed several extension to temporal logics binding the notation of time to formulas (see [Alur and Henzinger, 1992; Bellini et al., 2000] for a survey). The underlying models of these logics are represented as state transition graphs annotated with time constraints, using either *event* or *state* based approach. In the former approach, events record the changes of states at particular points of time. In the latter approach, the changes of states are recorded at each point of time. Therefore, both approaches use a different time domain. In particular, choosing the domain of time to be the set of natural numbers leads to the so-called *Discrete time* model. In this model a transition between states, represented by events, which happen only at the integer time values. The behavior of a discrete time model is described by the timed trace over a set of events that occur during the evolution of the model. [Koymans, 1990] showed a classification of temporal constraints with respect to event occurrences. The main advantage of event based logics together with their underlying discrete time model are their simplicity to express the quantitative properties. The quantitative requirements of systems often occurs at the discrete change of behaviors. Hence, the use of events is a quite natural to ideally specify such requirements. In the train gate example, in order to specify that whenever a train approaches the intersection, the gate must be closed within a particular time period, it suffices to specify that every occurrence of event *approach* is followed by the events *closed* within such a time period.

Besides their ability to specify quantitative requirements, events provide general observation view of systems' behaviors by abstracting lots of details of those systems. Indeed, the behavior of systems can be characterized by the set of all possible sequences of event instances that happen over time. This type of view of the behaviors can be used to specify the properties of complex timed systems, or particularly multi-agent systems. For example, the behavior of a transport agent in a logistic scenario is described by the sequence of events that the agent should take in order to accomplish a certain task, like *receiving order*, *transport*, *change route plan*, and *reach to destination*.

Specifying quantitative requirements, however, with time at which events occur, i.e. event based approach, is not the general choice to specify such types of requirements. There are quantitative requirements that might not be expressive by means of events. For example, it might be desirable to state that within some interval of time, say $10 \leq t \leq 20$, a certain property holds. This can not be expressed with events unless the boundaries of the interval coincides with occurrence of those events.

Choosing the time domain to be the set of non-negative real numbers, i.e. $\mathbb{R}^{\geq 0}$, leads to what is so called *continuous/dense* model, in which states have to be recorded at each point in time. Therefore, change of states is represented by letting the time to pass between one state to another. The quantitative temporal logics, based on this approach, are powerful and expressive to specify quantitative properties, as they record the state of the model at each point of time. They can cope with the limitations of event based logics mentioned previously. They, however, lack to express properties of events in models directly. They convert the event based into state based representation. Additionally, underlying model checkers—for example UPPAAL [Bengtsson et al., 1996] and Hytech [Henzinger et al., 1997]—convert the formulas depending on the occurrence of events, into formulas with state based representation. For example, to specify and verify that it is always the case that $event_1$ is followed by $event_2$ within t time unit, a traditional solution to verify this within a model M, is to translate this specification to a testing transition model A, and then check whether the parallel model of A and M can reach a designated state of A.

This chapter shows Region Computation Tree Logic (RCTL) that encompass, in the same framework, the expressive power of event and state based approaches. This is done by incorporating time notation on states and events. We use hybrid automata as an interpretation model of RCTL. In particular, the formulas of RCTL are interpreted on the set of all possible runs generated from the transition system of hybrid automata. Time, events, and constraints are the primary components constituting RCTL formulas. To plug the specification of properties, which can be verified within our presented approach in the previous Chapter, we use a fragment of this logic that specifies the properties which can be verified using reachability analysis.

The rest of this chapter is organized as follows: Sec.5.2 shows the syntax and semantic of RCTL. Sec.5.3 shows the specification of those important properties that can be verified by means of reachability analysis. Sec. 5.4, discusses related quantitative temporal logics.

5.2 Region Computation Tree Logic (RCTL)

This section primarily focuses on the definition of the region computation tree logic (RCTL), which extends the qualitative temporal logic of CTL with time on states, events, and constraints of variables. RCTL combines, in the same level of specifications, qualitative together with quantitative requirements. The formulas of RCTL are interpreted over the possible regions obtained from the run of hybrid automata. As described in Chapter 4, a region can be seen as a sequence of states separated by transition points. Each transition point marks the instantaneous exit from region Γ_{i-1} and the entrance into region Γ_i, and corresponds to the occurrence of a particular event. Therefore, we see regions constituting the essence of RCTL, such that RCTL can be viewed as a state based quantitative temporal logics in a sense that regions capture the changes of states, and as event based quantitative temporal logics in a sense that events mark the instantaneous exist from region to another. Thus, RCTL brings together, in the same framework, the advantages of both approaches. In the following we show the syntax and semantics of RCTL.

Definition 5.2.1 (Timed-variables) *Let $\mathbb{T}$ be a set of non-negative real variables called* timed-variables, *and $\Phi(\mathbb{T})$ [1] be a set of linear constraints over $\mathbb{T}$. The valuation ξ of the timed-variables $\mathbb{T}$ is a function $\xi : \mathbb{T} \rightarrow \mathbb{R}^{\geq 0}$. Given $\pi \in \Phi(\mathbb{T})$, we write $\xi \models \pi$, if ξ satisfies the constraint π.*

5.2.1 Syntax of RCTL

Let $\mathbb{X}$ be a set of real variables, $\mathbb{T}$ be a set of non-negative real variables disjoint from $\mathbb{X}$, $\Phi(\mathbb{X})$ and $\Phi(\mathbb{T})$ be two sets of linear constraints with free variables from $\mathbb{X}$ and $\mathbb{T}$ respectively, L be a set of atomic propositions denoting the locations, and $Event$ be a set of atomic propositions denoting events disjoint from L.

Definition 5.2.2 (Formulas of RCTL) *The formula Ψ [2] of RCTL are inductively defined as*

$$\Psi ::= p \mid a \mid \phi \mid y.\Psi \mid \pi \mid \neg\Psi \mid \Psi_1 \wedge \Psi_2 \mid \exists(\Psi_1 U \Psi_2) \mid \forall(\Psi_1 U \Psi_2)$$

for $y \in \mathbb{T}$, $p \in L$, $a \in Event$, $\phi \in \Phi(\mathbb{X})$, $\pi \in \Phi(\mathbb{T})$, and Ψ_1, Ψ_2 are RCTL formulas.

In addition to the definition of formulas, the following are standard abbreviations in RCTL similar to CTL:

$$\exists\Diamond\Psi \equiv \exists(true\, U \Psi) \qquad \forall\Diamond\Psi \equiv \forall(true\, U \Psi)$$

$$\exists\Box\Psi \equiv \neg\forall\Diamond\neg\Psi \qquad \forall\Box\Psi \equiv \neg\exists\Diamond\neg\Psi$$

[1] see Chapter 4 of syntax and semantics

[2] *true* is defined implicitly in $\Phi(\mathbb{X})$, see Chapter4

Given the previous abbreviations, the formula $\exists\Diamond\Psi$ indicates that there exists a path where Ψ is eventually true, whereas $\exists\Box\Psi$ indicates that there exists a path where Ψ is always true. The quantifiers $\forall$ and $\exists$ in front of the model operators $\Diamond$ and $\Box$ indicate a universal and existential quantifier on paths respectively. It should be notated that the abbreviation of the logical operators $\vee$ and $\rightarrow$ are defined as usual.

5.2.2 Semantics of RCTL

We will interpret the formulas of RCTL over the set of all possible regions generated from possible runs of hybrid automata. Recall again, let a region Γ take the form $\Gamma = (q,V,T)$, with $\delta(\Gamma) = q$ is its location, and V and T are the interval of valuations and time respectively, in which the region is admissible. If there is a transition from a region Γ_1 to a region Γ_2, then an event a occurs at some timing point t, written as $\Gamma_a \xrightarrow[t]{a} \Gamma_2$. A sub-region $\beta \subseteq \Gamma$, with $\beta \neq \emptyset$ means that $\beta = (q,V^{'},T^{'})$ with $T^{'} \subseteq T$ and $V^{'} \subseteq V$. A state $\sigma \in \Gamma$ means that $\sigma = (q,v,t)$, with $v \in V$ and $t \in T$. σ satisfies a constraint $\phi \in \Phi(\mathbb{X})$, written as $\sigma \models \varphi$, iff $v \models \varphi$. In the following, we show the semantics of RCTL formulas on the set of all possible runs Π_H.

Definition 5.2.3 (Semantics) *Let Ψ is a RCTL formula, H be a hybrid automaton, Π_H be the possible runs of H with a region $\Gamma = (q,V,T) \in \Pi_H$, and ξ is a valuation function of timed-variables. The satisfaction relation $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi$, which means that Ψ is satisfied in the region Γ within the time interval (duration) T for some valuation function ξ, is defined inductively as follows:*

- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} p$ iff $\quad p = \delta(\Gamma)$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} a$ iff $\;$ *there is* $t' \in T$ *with* $\;\Gamma \xrightarrow[t']{a} \Gamma'$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \phi$ iff $\;$ *there is* $\beta \subseteq \Gamma$, *for each* $\sigma_k \in \beta, \sigma_k \models \phi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} y.\Psi$ iff there is $t \in T$ such that $\xi(y) = t$ *and* $\langle \Pi_H, \Gamma \rangle \overset{T:=t}{\underset{\xi}{\models}} \Psi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \pi$ iff $\;\xi \models \pi$.

- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \neg\Psi$ iff $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\not\models}} \Psi$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi_1 \wedge \Psi_2$ iff $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \Psi_1$ and $(\Pi_H, \Gamma) \overset{T}{\underset{\xi}{\models}} \Psi_2$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \exists(\Psi_1 U \Psi_2)$ iff there is a run $\Pi \in \Pi_H, \Pi = \Gamma_0, \Gamma_1, \cdots$, with Γ=Γ_0,

  for some $j \geq 0$, $\langle \Pi_H, \Gamma_j \rangle \overset{T_j}{\underset{\xi}{\models}} \Psi_2$, and $\langle \Pi_H, \Gamma_k \rangle \overset{T_k}{\underset{\xi}{\models}} \Psi_1$ for $0 \leq k < j$.
- $\langle \Pi_H, \Gamma \rangle \overset{T}{\underset{\xi}{\models}} \forall(\Psi_1 U \Psi_2)$ iff for every run $\Pi \in \Pi_H, \Pi = \Gamma_0, \Gamma_1, \cdots$, with Γ=Γ_0,

  for some $j \geq 0$, $\langle \Pi_H, \Gamma_j \rangle \overset{T_j}{\underset{\xi}{\models}} \Psi_2$, and $\langle \Pi_H, \Gamma_k \rangle \overset{T_k}{\underset{\xi}{\models}} \Psi_1$

  for $0 \leq k < j$.

The quantifiers $\forall$, and $\exists$, in the previous semantics, are called paths quantifiers. The variable y in the formula $y.\Psi$ holds the time at which Ψ is satisfied. $y := t$ means the variable y is set to the value t. $\langle \Pi_H, \Gamma \rangle \overset{T:=t}{\underset{\xi}{\models}} \Psi$ means that the formula Ψ is satisfied in the region Γ when the time T is restricted to the time point t. In case Ψ represents an atomic proposition from the set $Events$, then $y.\Psi$ binds the time at which the event has occurred. This can be used to specify various quantitative properties, such as time bound response properties as we will see in what follows. However, if Ψ represents a constraint formula, then $y.\Psi$ evaluates the time interval at which the constraint Ψ is satisfied. This allows to specify quantitative properties, which could not be specified using events.

Definition 5.2.4 (Satisfiability) *Let H be hybrid automaton with Π_H as its possible runs. We say that H satisfies the RCTL formula Ψ, written as $H \models \Psi$, iff $(\Pi_H, \Gamma_0) \models \Psi$, where Γ_0 is the initial region of Π_H.*

5.3 Model Checking as Reachability

For the purpose of verification by means of model checking, we need to describe the properties. As it has been said in the previous section, the qualitative properties are often classified into reachability, safety and liveness properties. However, when the time becomes a critical factor to react in the environment, then the concept of safety and liveness properties should be refined. We are going to review these types of properties [Olderog and Dierks, 2008] and show how to specify these properties by RCTL, and hence encode them into the CLP queries for the purpose of model checking. In order to put model checking within our framework, we will concentrate only on the reachability requirements. Indeed, many properties of interest can be specified as a form of reachability, as we will see in the sequel. We will start specifying reachability of properties.

5.3.1 Reachability Properties

The reachability of a property Ψ means that there is a possibility to reach a state where Ψ holds. In other words, the reachability of the property Ψ asserts that starting from an initial state, is there a region along a run in which Ψ is satisfiable. This can be specified in RCTL as follows:

$$init \rightarrow \exists \Diamond \Psi$$

where *init* is the predicate characterizing the set of initial states and is defined as conjunctions of atomic propositions from L and constraints from $\Phi(\mathbb{X})$. This predicate expresses that the run to be considered are those that start from the initial state.

In terms of the CLP, the reachability of a certain region that satisfies the formula Ψ is done by performing forward reachability analysis from the system's initial state, and then checking whether the conjunction of Ψ with the possible reached regions is satisfied. Assuming for example *init* has been assigned to the set of initial states, the following is the CLP query to check the safety requirements.

```
?- reachable(init,Reached),
       member(Ψ1,Reached),φ.
```

In the previous query, the formula Ψ is rewritten as a conjunction of two formulas Ψ_1 and ϕ, where $\phi \in (\Phi(\mathbb{X}) \cup \Phi(\mathbb{T}))$ is an atomic the constraint appearing in the formula Ψ. Indeed, any RCTL formula can be rewritten as $\Psi = \Psi_1 \wedge \phi$, this is for the reason that at the most ϕ can be set to be true.

To demonstrate the reachability of a formula in a concrete example, let us return to the train gate controller example described in Chapter 4. Supposing one wants to check the possibility of reaching a region whose state satisfies that the *train* is at *near* within distance less than 10 *meters* and the *gate* is *closed*. First the initial state of the systems is given by:

$$init : train.far \wedge gate.open \wedge controller.idel \wedge x = 1000 \wedge g = 0 \wedge z = 0.$$

The intended formula is specified as

$$init \rightarrow \neg\exists\Diamond(x \leq 10 \wedge train.near \wedge gate.closed)$$

As shown, the set of atomic propositions L describes the possible locations of hybrid automata. Since locations of different automata may have the same names, we should identify them somehow. To do this, we will refer to each atomic proposition with the form $A.q$ meaning that the automaton A is at location q, as it has been shown in the previous specification.

CLP of the previous formula can be verified by asking the following query:

```
?-reachable((far,[1000]),(open,[0]),(idle,[0]),Reached),
      member((near,close,_,Time,_,X,),Reached), X $=< 10.
```

The successful answer to this query indicates reachability of the specified formula.

It is often that in certain cases we may be interested in the reachability of a certain property either before or after a time deadline has expired called *Time bounded reachability*. For example, the possibility of a formula Ψ to be reached within the bounded time α is specified in RCTL as

$$init \rightarrow \exists\Diamond\ (t.\Psi \wedge t \leq \alpha)$$

Demonstrating this by the previous example with $\alpha = 19$, We are going to check the reachability of the previous example within 19 unite of time.

```
?-reachable((far,[1000]),(open,[0]),(idle,[0]),Reached),
    member((near,close,_,Time,_,X,),Reached),
    X $=< 10, Time $=<19.
```

5.3.2 Safety as Reachability

A safety property states that *something bad must never happen*. The bad thing represents a critical property that should never occur. Let Ψ represent this critical property, then the safety property is specified using RCTL as

$$init :\rightarrow \forall\Box\neg\Psi.$$

Starting from the initial states, the previous formula states that the critical formula Ψ is never reached. Generally, a safety property can be violated within bounded time, which means that the exhibition of the previous formula by a single state within a region suffices to show that the safety property does not hold. Thus, safety property can be reduced to reachability property. In other words, since $\forall\Box$ and $\neg\exists\Diamond$ are dual, we can specify the same property as the following:

$$init :\rightarrow \neg\exists\Diamond\Psi.$$

The previous specification asserts that after executing the initial state *init*, the requirement characterized by Ψ will not be reached. To illustrate the safety property with an example, assuming one wants to check that the state, where the train is at the intersection—the train is at *near* location—with a distance `X=0` and the gate is *open* is a disallowed state. Even a stronger condition can be investigated, namely that the state, where the train is at the intersection and the gate is *down*, is forbidden. This safety requirement can be specified as

$$init \rightarrow \neg\exists\Diamond(x = 0 \wedge train.near \wedge gate.down)$$

This formula asserts that during the run of the system, starting from the initial state, there is no reached state where the train is near at distance $x = 0$ and the gate is at down state. Checking the safety property means that one checks the un-reachability of the following query:

```
?-reachable((far,[1000]),(open,[0]),(idle,[0]),Reached),
      member((near,down,_,Time,_,X,),Reached), X $= 0.
```

The constrain solver answers *No* for the previous query.

5.3.3 Additional Requirements

We showed that safety properties can be reduced to the reachability problem. As it is known, a safety property asserts what may or may not occur, but do not require that anything ever does happen. In the train gate example, closing the gate permanently can maintain the safety of the system, but it is unacceptable for the waiting cars or pedestrians in front of the gate. For this reason, the liveness property is needed to specify such requirements, which asserts that some property of interest will always occur. It should be noted that these type of properties can not be falsified in bounded time. Since the occurrence of some state does not say how long it will take for this state to occur, we can not sure that the liveness property is violated. For this reason, these types of properties are not strong enough in the context quantitative properties. Here one would like to see a time bound when the good state occurs. This leads to the next kind of properties.

Bounded Response Properties

A bounded response property is one of the most important classes of quantitative requirements used to specify many important applications. It asserts that something will happen within a certain limit of time. A typical application of bounded response property is the specification of worst case performance; that is the specification of an upper bound α on the termination of a system S: if started at time t, then S is guaranteed to reach a final state no later than $\alpha + t$ unit time. In the logistic scenario, for example, specifying that any received order is guaranteed to be delivered within 5 days is a bounded response property. In communication protocols, specifying that every request will be acknowledged within 3 seconds is a bounded response property. In the train gate example, a desired property is to specify that once the approach of a train is detected, the gate needs to be closed within a certain time bound in order to halt cars and pedestrian traffic before the train reaches the crossing intersection.

The following is the RCTL specification of a bounded response property between two events $event_1$ and $event_2$:

$$init \rightarrow \forall\Box(t_1.event_1 \rightarrow \forall\Diamond(t_2.event_2 \wedge t_2 \leq \alpha + t_1)).$$

The previous formula states that whenever there is a request $event_1$ occurs at time t_1, then it is followed by a response $event_2$, at time t_2, such that t_2 is at most $\alpha + t_1$.

It should be mentioned that this property can be falsified within time bound. Therefore this property can be specified as a kind of safety requirement represented as reachability. For this reason, proving the previous property means proving that it is not possible to reach a state where $event_2$ is not reached from $event_1$ within $t_2 \leq \alpha + t_1$. In other words, starting from $event_1$, finding a reachable state satisfies $event_2$, within α time bound, is sufficient to check the reachability of the property. In terms of the CLP, the previous property can be encoded into the following steps. Firstly, we get all possible reachable states from $event_1$ within $t_1 + \alpha$ as L. Secondly, we check that reachability of $event_2$ has not occurred. A positive answer of the reachability indicates a negative answer to the original problem, and vice versa. The following is a CLP query encoding the previous specification

```
?- reachable(Ψ0,Reached),
   reached_from(L,event1,Reached),
   reached_within(Target, α,L),
   \+ member((_,..,_,event2),Target)
```

We should say that the traditional way to verify this kind of properties in real-time system tools— like UPPAAL [Bengtsson et al., 1996] and Hytech [Henzinger et al., 1997]–is to translate that property to what is called a testing automata A, and then check whether the parallel composition of the underlying model together with A can reach a designated violation state. As we said earlier, the reason behind this translation is that there is no direct use of events in the model. The use of events is limited to construct only the parallel composition of automata. In contrast to our adopted approach, the direct use of events with the model allows us to avoid this translation process. This shows that RCTL is more expressiveness, particularly in our setting, than many others quantitative temporal logics.

In real-time systems, specifying the behavior on the discrete case, in some cases, is not satisfactory. Suppose for example that one needs to specify that a part of a certain region can be reached in some time bound interval. To do so, we present the bounded invariance properties.

Bounded invariance Properties

Like the bounded response property, bounded invariance property is one of the most important classes of quantitative timing requirements. It asserts that once an event has been triggered, a certain condition will continuously hold for a certain amount of time. It is often used to specify that something will not happen for a certain period of time. Formally, specifying that a certain property hold continuously for a certain amount of time in RCTL is like the following

$$init \rightarrow \forall\Box(t_1.event \rightarrow \forall\Box(t_2.\Psi \wedge t_2 \leq \alpha + t_1)).$$

where α is the duration at which the formula Ψ must be continuously held. For instance, whenever the train approaches the gate, the distance of the train is always larger than 100 meters for the duration of 20 time units. The property $\Psi = X > 100$ in this case represents the distance of the train, and *app* is the triggered event.

The bounded invariance property can be checked as a safety property. Starting from time t_1, finding a non-reachable violating state for the formula Ψ, within α time bound, is sufficient to check the reachability of the property. This can be encoded into CLP as the following

```
?- reachable(Ψ0,Reached),
   reached_from(L,event1,Reached),
   reached_within(Target, α,L),
   member((_,..,X,_,Target), X$≤100.
```

A satisfactory solution to the previous query violates the original property.

The way used to specify the bounded invariance properties can be used to specify what is the so-called *minimal event separation* [Henzinger et al., 1995] too, i.e no $event_2$ can occur earlier than α time units after an occurrence of $event_1$. This property can be specified as

$$init \rightarrow \forall\Box(t_1.event_1 \rightarrow \forall\Box(t_2 < t_1 + \alpha \rightarrow \neg t_2.event_2)).$$

5.4 Related Quantitative Languages

As mentioned in the introduction, several quantitative specification languages have been proposed based on temporal logics. The distinction among those languages depend on various parameters. First of all, the types of the models of computational; that is whether it is linear or branching model. Additionally, the accessibility of time; that is whether the time is implicit or explicit in the temporal logics. Another discrimination concerns the types of time domain. Choosing time to be a set of natural numbers gives us what is the so-called the discrete time model. In this model, the change of states can only happen at the integer time values. Choosing time to be a set of real numbers, gives what is the so-called the continuous/dense time model. In this model, the change of states is assumed to happen at an arbitrary point in time over the real line. Another important distinction among real-time models is whether one assumes that the system under consideration is observed at every instant in time leading to an interval based semantics [Alur et al., 1996a], or whether one only records a countable sequence of snapshots of the system leading to point-based semantics[Alur and Henzinger, 1993, 1994].

In this section, we focus on the other quantitative temporal logics that are used to extend the classical temporal logic with notation of time. Like the conventional view of temporal logics, we devide the extension of temporal logic into linear time and branching time logic.

5.4.1 Linear Time Logics

Linear time logics extends the traditional linear temporal logic by admitting time constraints on definitions of the formulas. In the following, we give overview about these linear quantitative languages—they are also called real time temporal logics.

Metric Temporal Logic

One of the earliest and most popular suggestions for extending temporal logics with quantitative setting is to extend the temporal operators by subscripting the modal operator with time interval. The idea of this extension is traced back to metric tense logic [Burgess, 1984] of superscripting or subscripting temporal logic. A successful and prominent example of this type of logic is *Metric Temporal Logic* (MTL)[Koymans, 1990; Alur and Henzinger, 1993], which extends linear temporal logic by constraining the temporal operators $\Box$, and $\Diamond$ with time intervals. For example, the formula $\Diamond_{[2,6]}\Psi$ means that Ψ is eventually true within 2 to 6 time units from the current time. The timed bounded response property— that is to specify that every p-state is followed by a q-state within 3 time units—can be specified using MTL by the formula $\Box(p \rightarrow \Diamond_{[0,3]}q)$.

The formulas of MTL are built from propositions using Boolean connectives and a time constrained version of until operator U_I. The formulas of MTL are interpreted over time state sequences of integer domain, which provide an interpretation for the propositions at every time instant. For example, the formula $\Psi_1 U_I \Psi_2$ holds at time t of a timed state sequences iff there is a later $t' \in (t+I)$ such that Ψ_2 holds at time t' and Ψ_1 holds throughout the interval (t,t').

Metric Interval Temporal Logic (MITL)[Alur et al., 1996a] is a variant of MTL employing dense time domain, instead of integer domain. Moreover, the bounded operator syntax is used with restriction such that temporal operators must not be bounded by singular interval—i.e. interval of the form $[a,a]$.

Explicit Clock Logic

Another type class of quantitative logics, is to extend the temporal logic with explicit notation of time. In this approach, time is defined with both a sequence of states and a sequences of temporal instants. The key idea behind this approach refers to use a dynamic state variable T and global variables over the time domain. The variable T represents the time of each state, i.e. it is considered as a global clock of a system. Due to the direct use of the global variables, the temporal logic is called *Explicit Clock Temporal Logic*. Examples of this approach can be found in [Harel et al., 1990; Pnueli and Harel, 1988; Ostroff and Wonham, 1990].

The time bounded response property can be specified by the Explicit clock approach as the formula $\forall x.\Box((p \wedge T = x) \rightarrow \Diamond(q \wedge T \leq x+3))$. The global variable x is bound to the time of every state in which p is observed.

Freezing Quantifier

Timed Propositional Temporal Logic (TPTL) [Alur and Henzinger, 1994] is a real time logic, which extends the propositional temporal logic with time notations in order to specify the quantitative properties of real time systems. The key idea of TPTL is to use what is called the *freeze quantifier* ”$x.$”, which binds the associated variable x to the time of the current temporal context. Therefore, the formulas of TPTL are defined similarly to the formulas of propositional temporal logic, but with the additional formula $x.\Psi$, which freezes the time at which the formula Ψ holds. The TPTL formula is interpreted over timed observational sequences. The formal $x.\Psi(x)$ holds at time t if $\Psi(t)$ does. Therefore the formula $\Diamond x.\Psi$ means that the time variable x is bound to the time of the state at which Ψ is eventually true. In this way, and by admitting atomic formula that relate times of different states, one can write the time bounded response property with TPTL as $\Box x.(p \to \Diamond y.(q \wedge y \leq x+3))$. The previous formula means that whenever there is a request p, and the variable x is frozen to the current time, the request is followed by a response q at time y, such that y is at most $x+3$.

5.4.2 Branching Time logics

In this section we show some of the formal specification languages that are used to extend the computational tree logic CTL with time constraints. Generally, the branching time logics adopt the same ideas of extending linear time logics.

Real Time Computation Tree Logic

Real-time Computation tree logic (RTCTL) is a propositional branching timed logic, which has been proposed by Emerson et al. [1992] to extend the temporal logic CTL with real time constraints. The extension allows to the model operators to be bounded with time interval ranging over integer domains. The use of integer domain simplify assumption of modeling real time systems, whose events occur with the ticks of a global clock. The formulas in RTCTL are generated from CTL formulas together with a rule that adds a natural number that abound on the modalities on the formula, such as $\forall(p\, U^{\leq k}\, q)$.

Timed Computation Tree Logic

Timed Computation Tree Logic (TCTL) [Alur et al., 1993] is another propositional branching timed logic that extends the qualitative logic CTL to the quantitative logic Real-time. The syntax of TCTL is very similar to that of RTCTL, but with less restrictive semantics. Precisely, TCTL is a bounded operator extension of CTL with point based real time semantics. Thus, TCTL uses timed automata [Alur and Dill, 1994] as timed state transition graph model in order to define the semantics.

Duration Temporal Logic

There are specification languages that specify quantitative properties based on the concept of duration [Chaochen et al., 1991]. *Duration Temporal Logic* (DTL)[Bouajjani et al., 1993] is one among of these languages permiting to reason about the duration of state properties (formulas). That is, given a finite interval on a run of a system, the duration of some state property in this interval is the time during which the property is true. Namely, the global time spent by the system in a run interval is simply the duration of the formula is true. DTL is a branching time logic with duration variables that can be associated with state formulas, and then used to express constraints on their duration. Thus, the formulas of DTL are built from the formulas of CTL together with a duration formula of the form $[x : \phi_1].\phi_2$, which associates the duration variable x with the formula ϕ_1 and binds x in ϕ_2. DTL considers simple timed graphs defined in [Nicollin et al., 1992] as a model for real time systems.

Integrator Computation Tree Logic

Integrator Computation Tree Logic (ICTL) [Alur et al., 1996b], similar to the approach presented in this chapter, is a quantitative temporal logic for specifying properties based on hybrid automata. The notation of time, however, is not explicitly defined within the model of computation. Instead, the model of computation is augmented with special clocks called integrators whose function is to measure the accumulated time delay inside control locations. These variables are used later to specify quantitative properties. Thus, ICTL extends CTL by admitting these integrators on CTL formulas. The key idea of integrators is inspired by duration temporal logic DTL [Bouajjani et al., 1993]. Each integrator has a type of sub-locations I from the set Q of locations of hybrid automaton. The integrator evolves continuously only inside I, and its value increases with a rate at which time advances whenever the control location in I and its value stays unchanged elsewhere. Therefore, the formulas of ICTL are constructed from CTL formula together with the reset quantifier formula $(z : I).\phi$. The previous formula binds the formula ϕ to the integer z, declares its type to be I, and sets its value to 0. Generally, ICTL can successfully specify duration properties, which can be verified afterward by means of Hytech [Henzinger et al., 1997]. ICTL, however, has some points that should be taken into consideration. Firstly, for the purpose of verification, a model should be extended to contain integrators, which in our case are not necessary. Secondly, ICTL can not specify properties that depend on events. Instead it has to follow an indirect way. For example, to specify that $event_2$ is a response to $event_1$ within α time units, one has to augment the model under consideration by an automaton A, whose *idle*, *wait*, and *violate* are considered as its control locations and t as its integrator. Initially, the control location of A is in the *idle*. When a trigger $event_1$ occurs, control pass to *wait* location and the integrator t is reset. The response $event_2$ causes the control to return to the *idle* location. The location *violate* is only enabled when $t \geq \alpha$. With the parallel composition of the original model with the automaton A, the specification of bounded response property can be specified as the un-reachability of the location *violate*.

6 Experimental Results and Related Work

The aim of this chapter is to evaluate the approach presented in the previous Chapters. The Chapter is doing so by obtaining several examples taken from the context of hybrid automata. It begins with describing these examples and their hybrid automata models. It conducts experiments to check such models against safety requirements. The evaluation of these examples are compared with Hytech. Furthermore, the Chapter discusses works that are related to the presented approach. The main results of this Chapter have already been presented in [Mohammed and Furbach, 2009a].

6.1 Benchmarks

In order to use the approach presented in the previous chapters to model and verify systems—particularly multi-agent systems—by means of hybrid automata, we have to demonstrate its feasibility by running experiments on examples taken from the hybrid automata context. We will refer to standard examples of verification of hybrid automata, which will be used to evaluate our presented approach. We use these examples to verify their safety properties. Firstly, the safety property of *scheduler* example [Halbwachs et al., 1994] is to check whether a certain task (with number 2) never waits. Secondly, in the *temperature control* example [Alur et al., 1994], the safety property must guarantee that the temperature always lies in a given range. Thirdly, in the *train gate controller1* example [Henzinger et al., 1995], the safety property has to be ensured that the gate is closed whenever the train is within a distance less than 10 meter toward the gate. In the *water level* example [Halbwachs et al., 1994; Alur et al., 1994], the safety property is to make sure that the water level is always between given thresholds (1 and 12). A non-linear version of both train gate controller—this non-linear version has been described throughout chapter 4—and the thermostat are taken from [Henzinger et al., 2000]. The safety property of the former one is to prove the similar safety property of the linear version. In the later one, the safety property is to prove that the temperature always lies between 0.28 and 3.76. The safety property of *Fisher's mutual exclusion protocol* [Henzinger et al., 1995] has to guarantee that two processes are never in the critical section at the same time. Last but not least, in the nuclear *Reactor1* example [Alur et al., 1996b], the safety property is to ensure that only one of the rods of the reactor can be put in. *Reactor2* is an approximated version of *Reactor1* which is found in the verification examples of Hytech [Henzinger et al., 1997].

In the following, we present the details of these examples and show their prospective hybrid automata models. Additionally, we show the specification of the safety requirements in terms of RTCL presented in the previous Chapter.

Scheduler Example

In the scheduler example, Fig. 6.1, there are two classes of tasks, activated by two different interrupts I_1 and I_2. Interrupt I_1 occurs at most once each 10 time units, whereas interrupt I_2 occurs at most once each 20 time units. The interrupt I_1 is responsible to activates the first class of tasks, which takes 4 time units. On the other hand, The interrupt I_1 is responsible to activate the second class of tasks, which takes 8 time units. Tasks of the second class have priority, and can preempt other tasks. The goal is to show that a task of the second class never waits.

For the purpose of specifying the model, there are two timers, namely c_i, for $i = 1,2$, to count the delay elapsed since the last interrupt I_i. There are two timers, namely x_i, to count the execution time of tasks, and two counters k_i, to count the number of pending tasks in each class. These counters are discrete variables, which means that their derivative is supposed to be 0 in any location.

The initial condition *init* of the scheduler is given as:

$$\begin{aligned} init : interrupt.start \ \wedge\ task.idle \ \wedge\ x_1 = 0 \\ \wedge\, x_2 = 0 \ \wedge k_1 = 0 \ \wedge k_2 = 0 \ \wedge c_1 = 0 \ \wedge\ c_2 = 0 \end{aligned}$$

The safety requirement, which is *the second tasks never wait*, is specified by RCTL as

$$init \rightarrow \neg\exists\Diamond(k_2 \geq 1 \wedge task.task1)$$

The previous formula states that starting from the initial state of the model, it is not possible to reach a state, where the first task is being processed and the pending of the second type of tasks is greater than 1.

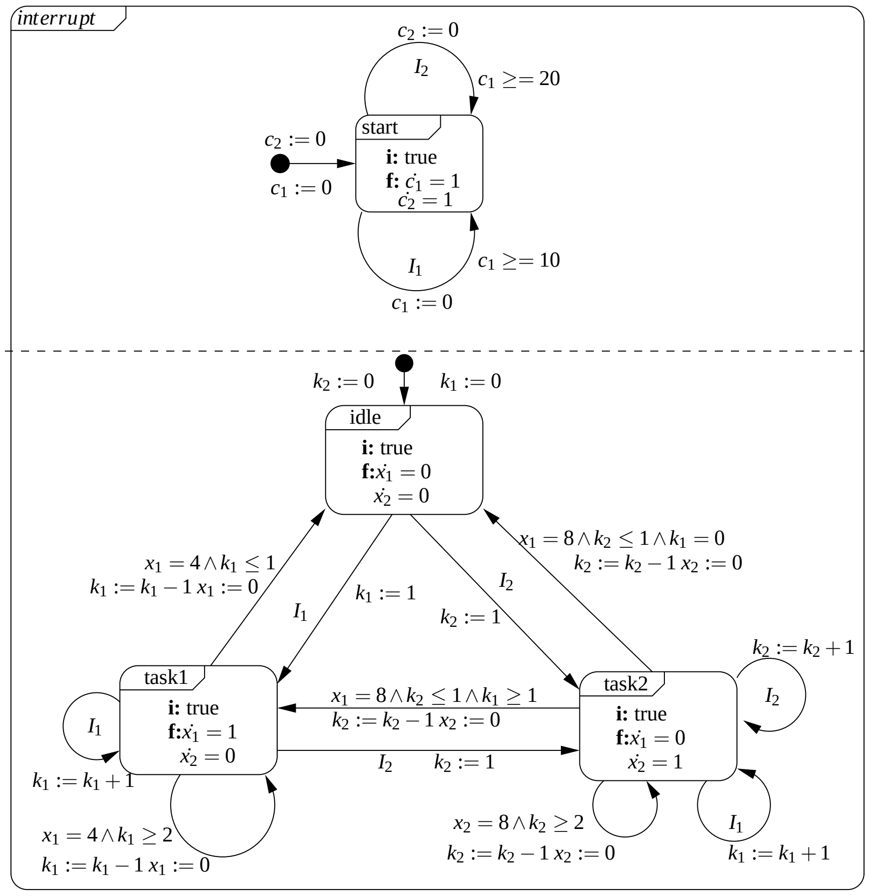


Fig. 6.1. Scheduler automata.

Temperature Control Example

In the temperature control example, a system controls the coolant temperature in a reactor tank by means of moving two independent control rods. The main goal of the system is to maintain the coolant between two temperatures 250 and 1100, so that when the temperature reaches its maximum value of 1100, the tank must be refrigerated with one of the rods. The temperature x rises at a rate of 34, and decreases at rates 25 or 10 depending on which rod is being used. A rod can be moved again only if 80 time units have elapsed since the end of its previous movement. If the temperature of the coolant cannot decrease because there is no available rod, a complete shutdown has to be performed. Figure 6.2 shows the specified model of this example: variable θ measures the temperature and the values of clocks x_1 represents the time elapsed since the last use of first rod, whereas x_2 represents the time elapsed since the last use of the second rod.

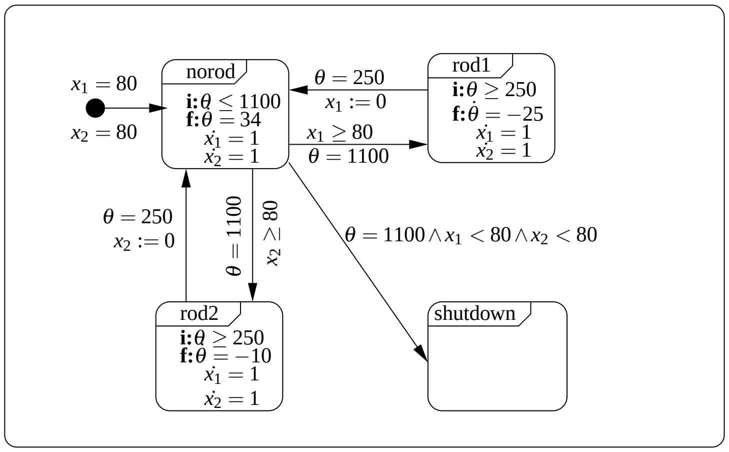


Fig. 6.2. Temperature control automaton.

The initial condition *init* of the system is given as

$$init : temp.norod \ \wedge \ x_1 = 80 \ \wedge x_2 = 80$$

The safety property, which is to check that the shutdown is never reached , is specified by RCTL as

$$init \to \neg\exists\Diamond\ temp.shutdown$$

Train Gate Example

The train gate controller example has been demonstrated as a running example throughout Chapter 4. There are two versions of this example. The key distinction between the two versions is the type of dynamics of the continuous function. In particular, the version presented in [Henzinger et al., 1995] is a rectangular version – see Fig. 6.3–, whereas the version presented in

[Henzinger et al., 2000] is a non-linear version. In both versions, the safety property is to guarantee that the gate must be closed whenever the train is within a distance of less than 10 meter toward the gate.

Recall again, the initial condition *init* of the system is given as:

$$init : train.far \wedge gate.open \wedge controller.idel \wedge x = 1000 \wedge g = 0 \wedge z = 0$$

the safety requirement can be rewritten as:

$$init \rightarrow \neg\exists\Diamond(x \leq 10 \wedge train.near \wedge gate.closed)$$

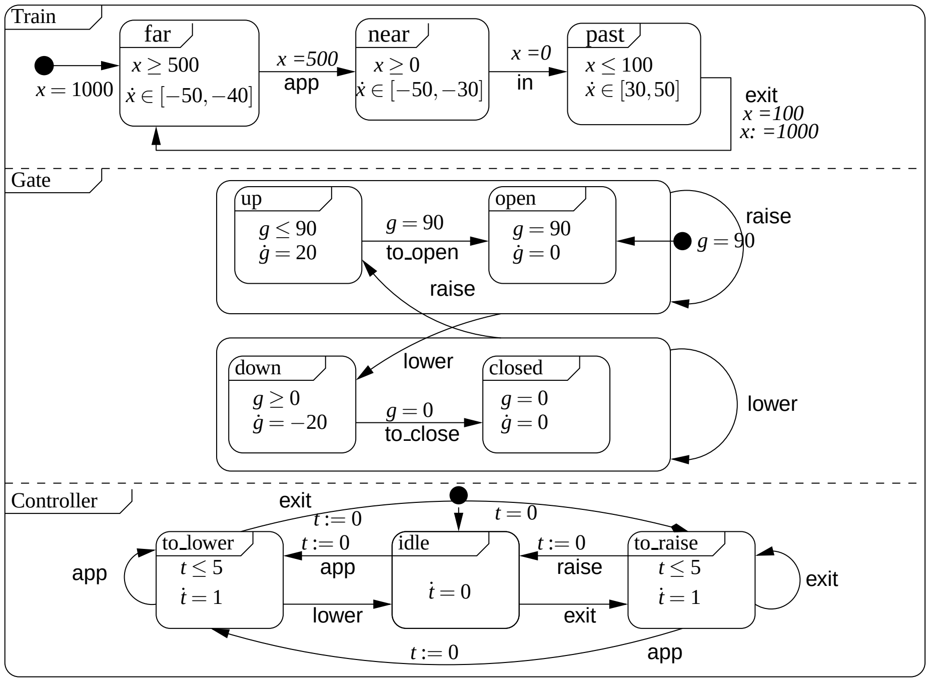


Fig. 6.3. Train gate example.

Water Level Monitor Example

The water level in a tank is controlled through a monitor, which continuously senses the water level and turns a pump on or off. The water level changes as a linear function over time, so that when the pump is off, the water level, denoted by the variable y falls by 2 inches per second and when the pump is on, the water level rises by 1 inch per second. As an initial state, the water level is 1 inch and the pump is turned on. The goal of the water tank is to keep the water between 1 and 12 inches. From the time the monitor signals to change the status of the pump to the time that the changes becomes effective, there is a delay of 2 seconds. The monitor must signal to turn the pump on before the water level falls to 1 inch and it must signal to turn the pump off before the water level reaches 12 inches. The hybrid automaton of Fig. 6.4 describes a water level monitor that signals whenever the water level passes 5 to 10 inches, respectively.

The initial condition *init* of the system is given as:

$$init : water.on1 \ \wedge \ y = 1$$

In terms of RCTL, the safety property is specified as:

$$init \rightarrow \forall\Box(1 \leq y \leq 12)$$

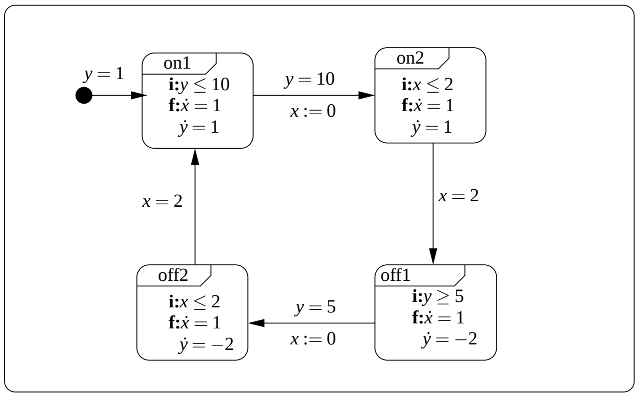


Fig. 6.4. Water level automaton.

Thermostat Example

A thermostat, Fig. 6.5, is a controller with delay: after the thermometer detects that the temperature is too low or too high, there may be a delay of up to one time unit before the appropriate control action is taken, i.e. turn the heater on or off, respectively. The variable x measures the temperature. Initially, $x = 2$ and the heater is on. The temperature rises according to the differential equation $\dot{x} = -x + 4$. The temperature eventually reaches 3; after a delay of one time unit, the thermostat sends a *turnoff* signal to the heater. The delay is measured using a variable z. Then the temperature falls according the equation $\dot{x} = -x$ until $x = 1$. One time unit after the temperature reaches 1, the thermostat sends a *turnon* signal to the heater. The goal of the thermostat is to prove that the temperature always lies between 0.28 and 3.76.

The initial condition *init* of the thermostat is specified as

$$init : thermostat.on \wedge x = 2$$

The safety property is specified as

$$init \rightarrow \forall\Box(0.28 \leq x \leq 3.76)$$

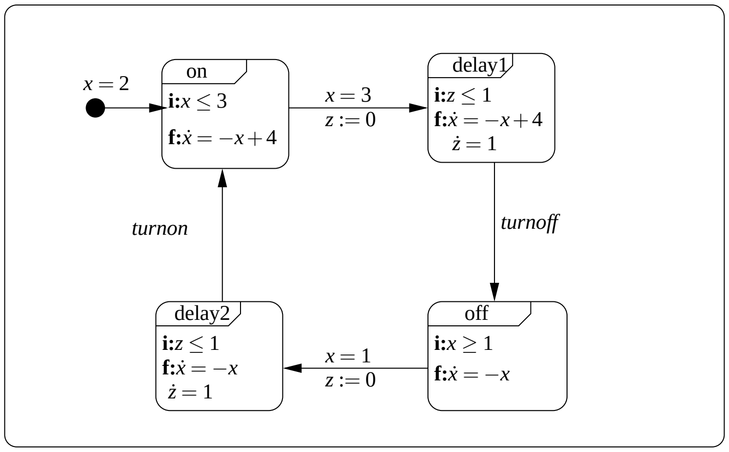


Fig. 6.5. The thermostat automaton.

Fisher's Mutual Exclusion Example

A mutual exclusion protocol is a system consisting of two process P_1 and P_2 each performing atomic read and write operations on a critical section of a shared memory variable k. Each process has a critical section. At any time instant one of the two processes is allowed to be in its critical section at most. Fisher's protocol ensures the mutual exclusion by modeling the execution of each process $P_i, i = 1, 2$ as the following pseudo-code:

```
P_i :  repeat
           repeat
               await k = 0
               k :=i delay b
           until k =i
           Critical Section
           k :=0
       forever
```

The two processes P_1 and P_2 share a variable k and each process P_i is allowed to enter its critical section iff $k = i$. Each process has a private clock. The statement *delay b* puts off a process for at least b time units as measures by the process's local clock. Each process takes a time unit at most measured by the process's clock, in order to make a single write access to the shared memory variable k, i.e. the assignment $k := i$ occurs . The values of a and b are the only information we have about the timing behavior of processes.

Fig. 6.6 shows the hybrid automata that model the mutual exclusion protocol of the two process $P_i, i = 1,2$. Given particular values to a and b, the safety property is to ensure that the two process are never in the critical section at the same time.

The specification of the initial condition of the system *init* is given as:

$$init : P_1.init \ \wedge \ P_2.init \ \wedge k = 0$$

The mutual exclusion requirement is specified by the RCTL formula:

$$init \rightarrow \neg\exists\Diamond(P_1.cs \ \wedge P_2.cs)$$

Reactor Example

In the reactor example Fig. 6.7, the temperature is represented by a non-linear variable x. The temperature of the reactor is initially 510 degrees and both the control rods are outside the reactor. In this case, the temperature rises according to the differential equation $\dot{x} = \frac{x}{10} + 50$. In order to prevent the reactor to shutdown, one of the two control rods can be put into the reactor core. Control rod 1 decreases the reactor temperature according to the differential equation $\dot{x} = \frac{x}{10} - 56$, whereas control rod 2 has a stronger effect and decreases the temperature according to the differential equation $\dot{x} = \frac{x}{10} - 60$. When a control rod is removed from the reactor, it cannot be put back into the reactor core for 15 seconds. This requirement is enforced by the clock variable x_i $(i = 1,2)$, which measures the elapsed time since the control rod i has been

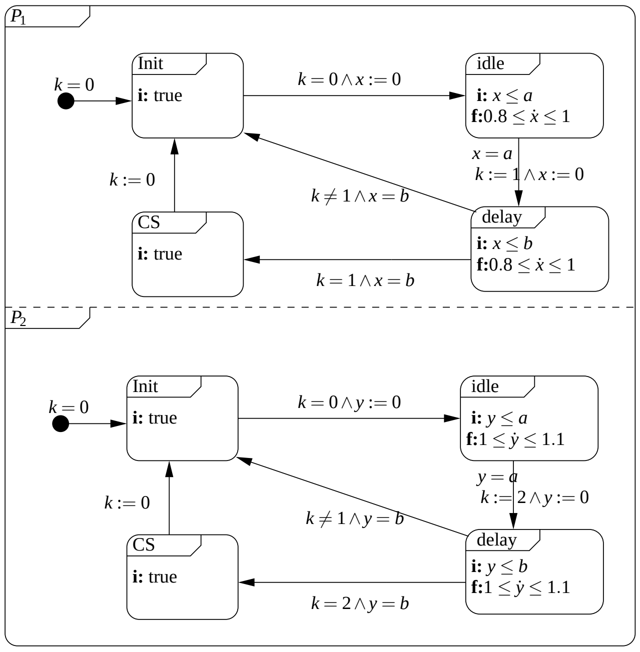


Fig. 6.6. Fischer mutual exclusion.

removed from the reactor core. The safety requirement asserts that one of the rods must be put in the reactor, if the reactor temperature reaches 550 degree.

The specification of the initial condition of the system *init* is given as:

$$init : reactor.norod \wedge rod1.out1 \wedge rod2.out2 \wedge x = 510 \wedge x1 = 15 \wedge x2 = 15$$

The safety property is specified as:

$$init \rightarrow \neg\exists\Diamond(x = 550 \wedge rod1.out1 \wedge rod2.out2)$$

6.2 Evaluation and Discussion

This section compares the evaluation of the benchmarks demonstrated in the previous section using our proposed approach and Hytech [Henzinger et al., 1997]. We have chosen Hytech as a reference tool as it provides the most general input language by supporting the full scope of linear hybrid automata. It tackles also the verification procedure based on reachability analysis similar to our adopted approach. In contrast to our approach, Hytech treats the continuous dynamics by using a polyhedral manipulation library [Halbwachs et al., 1994].

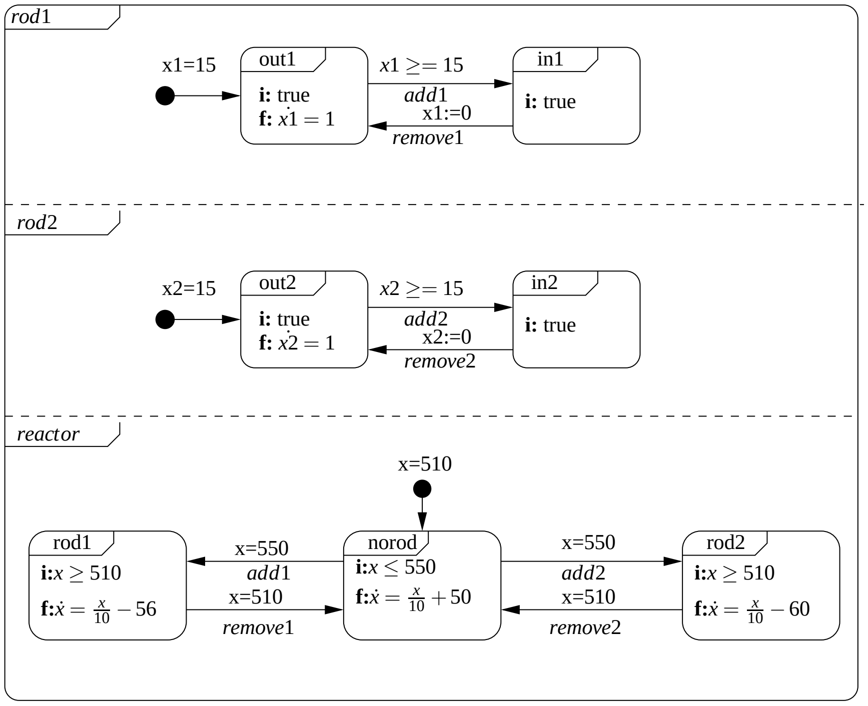


Fig. 6.7. Scheduler automata.

Example	**HyTech**	**CLP**
Scheduler	0.12	0.07
Temperature Controller	0.04	0.02
Train Gate Controller1	0.05	0.02
Water Level	0.03	0.01
Train Gate Controller2	-	0.02
Thermostat	-	0.01
Fisher protocol	0.11	0.34
Reactor1	0.01	0.01
Reactor2	-	0.01

Fig. 6.8. Experimental results.

Fig. 6.8 illustrates the performance of our CLP and Hytech on running the benchmarks. The performance is given in seconds. The symbol "–" within Hytech column indicates that its underlying example can not be expressed in its direct form within Hytech; we will come to this point in more details. The results revealed that our approach has a slight advantage with respect to the performance regarding the run-time of checking the properties of the benchmarks.

Despite the fact that Hytech has an advantage over others hybrid automata model checking tools concerning checking parametric analysis, i.e. giving the conditions on some parameters, which violate safety requirements. There are shortcoming issues in Hytech, which we take into the consideration in our presented approach. The first issue concerns the expressiveness of the dynamical model. In Hytech, there are no direct means of automatically verifying nonlinear hybrid automata. This is for that reason that HyTech restricts the dynamical model to linear hybrid automata in which the continuous dynamics are governed by differential equations $\dot{x} = a$ or differential inclusion $a \leq \dot{x} \leq b$ for some integers a and b. This illustrates putting the symbol ”$-$” in the column of hytech to some example. To overcome this point, the nonlinear dynamics, e.g. of the form $\dot{x} \bowtie c1 \cdot x + c2$, for some integers $c1, c2$ and $c1 \neq 0, \bowtie \in \{<, \leq, >, \geq, =\}$, are firstly approximated either by a linear phase portrait or clock translation method [Henzinger et al., 1998b]. In the former method, the approximation method is obtained manually by partitioning the state space of each control location into a set of control locations. Within each partitioned location, the continuous flow is approximated using linear flow, such that the nonlinear variable x in a location L is approximated by a differential inclusion $a \leq \dot{x} \leq b$, where the integer constant a and b specify the minimal and maximal rate of change of the variable x in the location L and are obtained from the differential equation, the location invariant and location initial state. In the location *norod* in Fig. 6.7 for example, if we take the invariant and initial value into consideration, we find that the derivative of $\dot{x}$ is bounded below by 1 and above by 5. Thus, the continuous dynamics of the location *norod* is approximated to $101 \leq \dot{x} \leq 105$. Henzinger et al. [1998b] showed that this method may cause a substantial blow-up ofverification procedure the state space. On the other hand, the idea behind the clock translation method is to replace a nonlinear variable x by a clock t_x, if the value of x can be determined uniquely from the value of t_x at all the time. This happens if t_x measures the time that has elapsed since the value of x was last changed by a discrete transition, if the value of x after that change is recorded, and if x has followed a unique flow since that change.

In both methods of approximation, the verification phase is carried out on the approximated model, so that every run of the approximated nonlinear system is a run of the approximating linear hybrid automata. On the other end of the spectrum, our implemented *CLP* approach is more expressive, as it allows the direct use of more general dynamics. In particular, *CLP* can directly handle dynamics expressible as a combination of polynomials, exponentials, and logarithmic functions explicitly without approximating the model.

An additional shortcoming issue of Hytech deals with the type of properties which can be checked withing a hybrid automata model. HyTech cannot verify simple qualitative properties that depend on the occurrence of events, despite of the fact that events are used to construct the composition of different parts of hybrid automata. On the other hand, simple duration properties between events can be verified using HyTech. To do so, the model must be specified by introducing auxiliary variables to measure delays between events or the delay needed for a particular conditions to be hold.

Other simple quantitative properties like time bounded response and minimal separation time between events are further properties that can be verified using HyTech. These properties, however, can only be checked after augmenting the model under consideration with what is called a *monitor* or *observer* automaton (cf. [Henzinger et al., 1995]) whose functionality is to observe the model without changing its behavior. It records the time as soon as an event occurs. Before the model is verified, the monitor automaton has to be composed with the original model, which may add further complexity to the model. For example, in order to check that the event $event_1$ is allways followed by the event $event_2$ within α time unit in hybrid automata H, the monitor automata of Fig. 6.9 should be composed first with H. Checking the time bounded response property is translated into checking the reachability of the control location *viol*. As it has been demonstrated in our approach, however, there is no need to augment the model with an extra automaton. This is for the reason that during the run, not only the states of variables are recorded, but also the events and the durations of time. Consequently, constraints solvers can be used to reason about the respective property

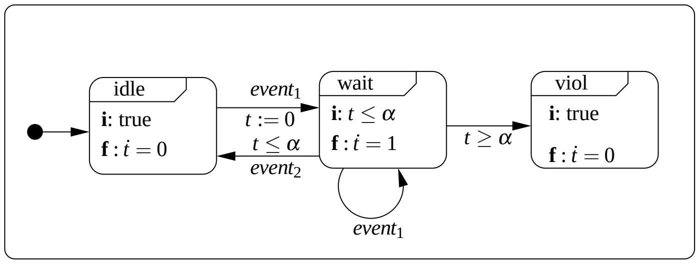


Fig. 6.9. A monitor automaton for the time bounded response property.

6.3 Related Works

In this section, we will review related work that model, specify and check systems by means of hybrid automata and their restricted classes. Basically, we classify this works into two categories. In the first categories, we discuss the algorithmic approaches in which several tools exist for the purpose of modeling, specifying and analysing of systems. In the second category we relate our work to those works which adopt CLP as a framework for hybrid automata. In the following we will discuss these two categories.

6.3.1 Algorithmic Approaches

There are several formalisms and tools for hybrid automata and their restricted cases, e.g. timed automata. In this section, we briefly introduce those lines of works that are more or less closely related to our presented approach.

As already shown, *Hytech* is a tool for modeling and automatic verification of linear hybrid automata. A system is modeled as concurrent hybrid automata that must be parallel composed prior to the verification phase. Similar to *Hytech*, *PHAVer* [Frehse, 2005] is a tool supporting to analyze linear hybrid automata. Basically, *PHAVer* is emerged to overcome the arithmetic overflow errors of *Hytech* resulting from the limited digits of the exact arithmetic operations. To cope with this, *PHAVer* enhances fix-point computation algorithm for reachability with operators for partitioning locations and simplification of sets of states. The partitioning process of the reachable locations is done during the analysis. This process is performed by splitting locations recursively until a minimum partition size is reached. The purpose of partitioning locations is to improve the accuracy on the dynamics. Despite its enhancement, *PHAVer* is a quite similar to *Hytech* from various prospectives. First, *PHAVer* does not specify properties using a kind of formal specification languages. Instead, it handles an algorithmic language built up from commands that manipulate set of states. Additionally, *PHAVer* computes all the states that are connected to the initial states by a run. Furthermore, the composition of hybrid automata has to be done prior to the verification phase. However, the process of splitting locations in *PHAVer* is restricted to specify larger systems as it adds extra complexity to the state space.

Fränzle and Herde [2007] present an approach for verification of hybrid systems. This approach applies what is called bounded model checking (BMC) [Biere et al., 1999] to linear hybrid automata encoded into predicative formulas suitable for BMC. For this reason, a tool has been developed called *HySAT* that combines a SAT solver with linear programming. In

HySAT boolean variables are used for encoding the discrete components, and real variables represent the continuous component. The linear programming routine is used to solve a large conjunctive system of linear inequalities over reals, whereas the SAT solver is used to handle disjunctions. However, modeling systems as concurrent hybrid automata is not taken into consideration in this approach.

In addition to the tools of hybrid automata, several restricted dynamics model checking tools have been developed in the last two decades. For instance, *Uppaal* [Bengtsson et al., 1996; Behrmann et al., 2004; Larsen et al., 1997] is one of those tools which is widely used to model and verify timed systems. *Uppaal* implements forward search algorithms, in which the state space is explored in a breadth first manner. It models systems as a network of timed automata and supports communication via shared variables. The network of timed automata is composed using on-the-fly technique and the model checking procedure is performed using a symbolic representation of the infinite state space by sets of linear constraints. The computation of clock constraints is managed with a data structure known as *Difference Bound Matrices* (DBMs) [Bengtsson and Yi, 2004]. For the specification of the properties, *Uppaal* uses a fragment of TCTL [Alur et al., 1993] with restriction to the properties that can be checked with reachability analysis.

Kronos [Yovine, 1997] is another well known verification tool for timed automata. It implements a symbolic model checking algorithm for the timed temporal logic TCTL developed in [Henzinger et al., 1994]. It incorporates also both forward and backward algorithms for the reachability analysis. *Kronos* allows us to express and verify not only reachability properties but liveness properties as well. It can express full TCTL model checking and the invocation of the model checker will select whether forward or backward analysis will be performed. Like *Uppaal*, a system is modeled as a set of concurrently operating time automata. *Kronos* can perform model checking using a symbolic representation of the infinite state space by sets of linear constraints. To improve the exploration of the state space, *Kronos* also implements on the fly technique. Additionally, the symbolic computations are also managed with the DBM data structure.

The time *Cospan* [Alur and Kurshan, 1996] is a very restricted form of timed automata. It supports verification based on automata language of coordinating processes with timing constraints. A system to be verified is modeled as a collection of coordinating processes described as a finite automata with timing constraints. These timing constraints are expressed by associating lower and upper bounds on the time spent by a process in some local state. *Cospan* is considered a single language framework in the sense that both the model and specification of a system are expressed using automata. Thus the verification procedure asks for checking whether the language of the product of the model and the property is empty. Checking the emptiness is performed by searching through the reachable state of the model. *Cospan* includes two types of search: an on the fly enumerative search, and symbolic binary decision diagrams [McMillan, 1993].

Similar to time *Cospan*, there are further tools which lay between timed automata and automata augmented with timing constraints. *Timed HSIS* [Balarin and Sangiovanni-Vincentelli, 1994] and *VERITI* [Dill and Wong-Toi, 1995] are example of such tools.

6.3.2 Constraints Based Approaches

Constraints based approaches have been used generally as practical implementations platform for automatic verification [Delzanno and Podelski, 2001; Nilsson and Lübcke, 2000; Ramakrishnan et al., 2000]. In addition, these approaches have been applied to modeling and analysis hybrid systems. Similar to the algorithmic approaches, constraints based approaches ranging from simple to more general dynamics. In the following we survey these approaches.

Urbina [1996] presents a pioneer approach CLP(R) [Jaffar et al., 1992] to model and analyze linear hybrid automata. In his approach, he translates hybrid automata into equivalent CLP(R) programs, where discrete transitions, invariants, flows and initial conditions are encoded as CLP(R) constraints. In addition, he adopts the quantitative logic ICTL [Alur et al., 1996b] to specify requirements of hybrid automata. A reachability analysis is the fundamental verification technique in his approach. In contrast to our presented approach, his approach does not provide an automatic mean to construct the composition of hybrid automata. Instead, the composition has to be explicitly encoded manually by a user before applying his CLP implementation. This is a tedious task, especially in the case of MASs where a group of agents exists.

Banda and Gallagher [2008] show how reachability analysis for linear hybrid automata can be done by means of CLP too. They present a scheme that translates linear hybrid automata into CLP clauses. The composition of automata is constructed using the product construction of clauses with synchronization on shared events which are handled as constraints too. The analysis of the CLP program is checked against constraints existence. In contrast to our approach, the way in which they construct the composition of the CLP program leads to an exponential increase in the number of clauses in general.

Additionally, they do not provide any validation techniques into the specification of intended requirements. The analysis of requirements is also restricted to finding a constraints that obey or violate a certain state. Therefore, real time requirements are not expressed in their approach.

Ciarlini and Frühwirth [2000] present another CLP approach for the verification of hybrid automata. In their approach, a model of hybrid automata is described as CLP where they derive test data for conditions of interest from the output of the symbolic execution of CLP. These conditions are specified declaratively in the form of first order temporal logic. They develop an algorithm that takes the resulting constraints from the valid run of CLP to obtain test data for the automata by projecting outputs constraints onto the conditions of interest. As a result from the projection process, domains of values of the constraints of interest are obtained which are considered as the test data for the automata. In turn, these test data can be used as the validation of the hybrid automata model. In this approach, however, there is no means to prove qualitative or quantitative requirements systematically. In addition, the approach has not taken the compositions of the concurrent automata in consideration. Instead, the symbolic execution of the approach takes the possible interleaving run of hybrid automata.

In contrast to our approach, various works approach to model a behavior of a hybrid system as an automaton using CLP, but they do not handle concurrent hybrid systems. For example, Hickey and Wittenberg [2004a] present an approach to model hybrid systems using CLP(F) [Hickey and Wittenberg, 2004b]. They show that nonlinear dynamics can be model with CLP(F). However, modeling concurrent systems are not expressed in their approach. Furthermore, they provide no means to handle model checking. Instead, they show techniques for satisfaction of constraints within some regions of interest.

Another approach on model checking of hybrid systems is presented in [Gulwani and Tiwari, 2008]. There, an analysis technique is proposed which is able to derive verification conditions, i.e. finding the constraints that hold in reachable states. These conditions are universally quantified and transformed into purely existentially quantified conditions, which is more suitable for constraint solver. An implementation in Lisp is available employing a satisfiability modulo theories (SMT) solver.

In addition to those CLP approaches that model and analyze of hybrid automata, there are works proposing CLP to restricted classes of hybrid automata. For example, the works of [Gupta and Pontelli, 1997; Jaffar et al., 2004] describe schemes for modeling timed (safety) automata as CLP programs. These works do not construct the overall behavior prior to modeling. Instead, they model model each automaton separately, but the run of the overall model takes all possible paths which result from the product of each component into consideration. Likewise, this leads to unnecessary computation.

Another restricted CLP approach of hybrid automata has been developed by Delzanno and Podelski [1999]. In their approach, they have only modeled discrete transition systems. Furthermore, they have showed how to encode CTL temporal operators into CLP, in order to check temporal properties of systems. Thus, their approach can be considered as a special case of our presented approach.

Part III

Extensions to the Framework

7

Deliberative Multi-agent Planning

Hybrid automata can be used to formally model and coordinate plans of reactive multi-agent systems. In most cases, reactivity in dynamic environments is not satisfactory. It is favorable for agents to plan their behaviors according to some preference function. Most current verification tools of hybrid automata are inadequate to model such agents' plans. this chapter goes toward extending the decisions making of hybrid automata by incorporating the preference on transitions. A scenario taken from supply chain management is demonstrated to show the Chapter's approach. Analysis of agents' plans are examined using CLP. The main contribution of this chapter has been published in [Mohammed and Furbach, 2009a].

7.1 Introduction

Planning to reach some goal is an essential requirement for multi-agent systems. A classical planning task is generally defined by an initial state I, a final state G and a set of actions A. The solution of the planning is to find the action sequence leading from I to G. In the last few decades, several planning approaches have been developed. Automated planning [Nau et al., 2004] is one of those approaches that has received attention. In this approach, formal methods are attractive to guarantee the reliability of the solution. In particular, several works have adopted model checking to solve the planning problem. This is known as *planning as model checking* [Giunchiglia and Traverso, 2000]. The key idea behind this approach is that the planning domains are formalized as semantic models and the planning goals are specified by formulas of temporal logics. Planning is performed by verifying whether temporal formulas true in a semantic model.

Multi-agent planning [de Weerdt et al., 2005] has been raised as a motivation to solve complex planning problems. In this setting, the planning problem is divided into sub-problems, which are distributed to agents. One key feature of multi-agent planning is the nature of the environment in which the agents are involved. In realistic problems, the environment tends to be dynamic and the behaviors of the agents change continuously therein. When unexpected events, threatening the plan, arise in the environment, then agents should react to those events in a proper way. Planning in such a case is called continual planning [DesJardins et al., 2000]. DesJardins et al. have described the situations in which agents should engage in continual planning. One of these situations occurs, if agents' objectives can evolve over time. In this case the purpose of the planning is to set a target that can be achieved under several constraints at a given time.

Reacting to the unexpected events, in real-time, can avoid any risk that might occur during the planning. Agents should not only react to change those events that threaten the execution of the plan, but also coordinate opportunities to improve the future development of the plan. This can be done by selecting the most favorable course of actions based on utility functions, e.g. cost, quality. Hence, it seems to be favorable to provide a formal way that is capable to model and analysis the multi-agent planning in dynamical environments which combines in the same framework both aspects of planning.

Hybrid automata can be used to model plans of multi-agent systems that are defined through their capability to continuously react in dynamic environments while respecting some time constraints. As presented in Chapter 3, there are works adopting hybrid automata to formally model reactive mutli-agent systems. Examples of that works include the work presented in [El Fallah-Seghrouchni et al., 2003] and [Egerstedt, 2000]. There are authors, such as [Hutzler et al., 2005], who have approached timed automata to model reactive agents. In reactive agents, decision making depends entirely on the occurrence of events so that the agents base their next states on their current sensory events. In contrast to reactive agents, deliberative/rational agents try to find the plan which utilizes a certain objective function. Making deliberative decisions are inadequately expressive to hybrid automata. In various situations, one needs such type of decision making. In a logistic scenario, for example, changing the current route plan of a working truck to perform a new plan might utilize the profit of the company rather than allocating a new truck to perform such a new plan. In soccer-agents scenario, running one agent toward the ball—particularly the closest agent to the ball—and spreading other teammates on the field will increase the utility of the team behaviors, instead of running several agents toward the ball at once. To our knowledge, the current formal model of hybrid automata and their tools do not provide means for modeling these types of situations. Therefore, it seems to be useful to extend hybrid automata in a way that allows them to combine both reactive and deliberative decision making. This combination can avoid catastrophic failures and provide better quality of decisions in time constrained dynamical environments. Consequently, the formal verification of hybrid automata, by means of reachability analysis can be used as planning-problem solver where a plan can be achieved, iff the final plan is reachable. Hence the trajectory from the initial state to the reachable goal will accommodate the solution of this plan.

This chapter contributes to use hybrid automata as a conceptual model for planning and it goes toward enhancing the decision making of the hybrid automata in order to improve the future outcomes of models. This can be accomplished by letting discrete transitions occur on the basis not only of reactive decisions of the continuous evolution of the variables, but also of particular preference functions. The expressiveness of the CLP prototype presented in Chapter 4 facilitates to implement this extension. To demonstrate the idea of this chapter, we present an example taken from supply chain management in continuous dynamic environment. As far as we know, this is the first attempt to use hybrid automata for planning multi-agent systems whose decisions rely on a performance measurement.

The rest of this chapter is organized as follows: In Sec.7.2 we first review the planning, show its relation to model checking framework, and show the planning using deliberation. Sec.7.3 Introduces the scenario, which illustrates the approach of this chapter. Then formal definitions of extended hybrid automata are discussed in Sec.7.4. Finally, Sec.7.5 shows how to specify and analyze the planning requirements.

7.2 Planning

Planning in artificial intelligence is *decision making* about actions to be taken. Generally, the classical planning problem can be formulated as follows: given

- a description of the known part of the initial state of the world denoted by I,
- a description of the goal, denoted by G, and
- a description of the possible actions that can be performed,

then, the solution of the planning determines the sequence of actions in order to reach G from I under achievement a certain objective.

In the last decade, the term multi-agent planning [de Weerdt et al., 2005] has been introduced as an approach to the planning problem with complex goals that divides the problem into sub-problems and allows each agent to deal with each sub-problem. The solutions to the sub-problems have to be combined and coordinated afterwards to achieve a coherent and feasible solution to the original problem. According to de Weerdt et al., multi-agent planning is defined as: given a description of the initial state, a set of global goals, a set of at least two agents and for each agent a set of its capabilities and its private goals, find a plan for each agent that achieves its private goals, such that these plans are jointly coordinated and the global goals are met as well.

7.2.1 Planning as Model Checking

In the last few years, several research has approached planning with formal methods based on model checking [Giunchiglia and Traverso, 2000]. The key idea behind this approach refers to the strong relation between the framework of model checking and planning. The framework of model checking consists of a formal model M of a system, an initial state s_0 of the system, and a formal specification of a property ψ to be verified in this system. The model checking aims at verifying if ψ is satisfied in M, i.e. $M, s_0 \models \psi$. Basically, the model checker is an algorithm that takes (M, s_0, ψ) as input and systematically visits the states of the model M, in order to verify if the property ψ holds. The model checkers returns success if M satisfies the property ψ; otherwise, it returns a counter-example, that is a state in the model M where the property ψ is violated. In this framework, the planning problem can be formally described in a way that the model M describes the planning environment's dynamics, s_0 describes the initial state of the environment and the property ψ describes the goal to be achieved. So, if $M, s_0 \models \psi$, the planner returns a *plan*, i.e. the behavior which allows the systems to achieve its goal; otherwise, the planner returns failure. Fig. 7.1 shows the relation between model checking and planning.

Using the framework of model checking, the solution of the plan is the trajectory holding the sequences of reached states (actions) from a starting state to a goal state. Another way to find the plan is to use the counter example—generated from a model checker—as a solution of the planning problem. The negation of the goal is stated as safety property and introduced to the model checker. If the problem is found to be reached, one of the powerful points of model checkers is to generate a counter-example, which can be used to provide a solution to the problem. This idea has been adopted by several research, such as [Giunchiglia and Traverso, 2000; Pistore and Traverso, 2001; Pereira and Barros, 2008].

The classical way to solve the planning problem has been focused on finding any solution plan without careful consideration of quality of the plan. For many practical problems, the problem is not only to find a plan, but also to achieve a certain objective at the end of the plan. The objective of the planning, according to [Nau et al., 2004], can be specified in several different ways as follows:

- The simplest specification consists of a goal state G, and the objective is achieved by any sequence of state transitions that reaches the goal states. In a logistic scenario, for example, the objective to have a shipment reached to its final destination.
- The objective is to satisfy some condition over the sequence of states followed by the system. For example, one might want to require states to be avoided during the planning, e.g. reaching after deadline .
- The specification of objective based on a utility function with penalties and rewards. The goal is to optimize some function of these utilities, e.g. sum, maximum, minimum, over the sequence of the states followed during the planning.

Thus, the assessment of the planning objective is crucial to determine the quality of the plan.

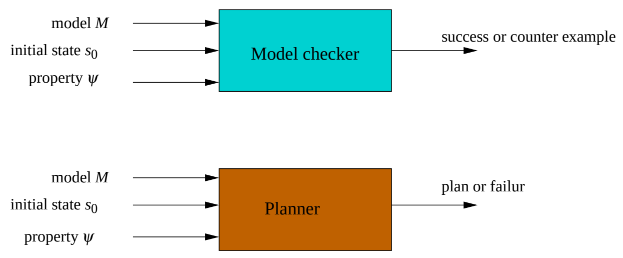


Fig. 7.1. Planning versus Model checking.

7.2.2 Deliberative Actions

During planning, if things do not work as expected, agents must be able to react and reconsider the plan. For instance, if an agent runs into unexpectedly high traffic on its chosen route through the city, then it must be able to consider changing the plan. When there are alternative actions to react to the unexpected changes during the plan, the agent should deliberate to select the best alternative way. The deliberation process generally focuses in the ways to achieve a goal and the decisions of which goal to be achieve. Deliberation is particularly useful in hazardous environments where the correct action selection is crucial. Decker and Lesser [1998] have stated that an agent should deliberate, if one of the following conditions is met:

- The agent has a choice of actions and the choice affects performance.
- The order in which activities are carried out affects performance.
- The time at which actions are executed affects performance.

If the agent deliberates rationally, it will try to find the best ways it can perform the actions. In other words, when the agent performs a certain action *A*, it will try to do so in a way that maximizes the expected utility of *A*. So to decide whether to perform *A*, the agent should assume that it will be performed in the best way, i.e. the value of expected utility of *A* should be the maximum value for all the ways in which the agent can think of for performing *A*. The expected value of an action is defined, according to the decision theory [c.f Bermúdez, 2009] to be the expected value of the environment when the action is performed. In the decision theory, standard models of decision-making involve calculations of the expected utility of each available action. Starting with each possible outcome, multiplying the utility of that outcome by the probability of the condition of the environment in which it will come about. Summing of the values that obtained for each of the possible outcomes of a certain action, gives the expected utility of that action. Standard models of decision-making identify the rational resolutions of decision problems as those that maximize expected utility.

7.3 Planning Scenario

As logistics competency becomes a more critical factor in creating and maintaining competitive advantage, logistics measurement becomes increasingly important because the difference between profitable and unprofitable operations becomes more narrow. In recent years, several research such as Fox et al. [2000] has viewed the supply chain as composed of a set of intelligent (software) agents, each responsible for one or more activities in the supply chain and each interacting with other agents in planning and executing their responsibilities.

In many logistic domains, some of the transport orders are only known in the short time, traffic is often unpredictable and unexpected events might

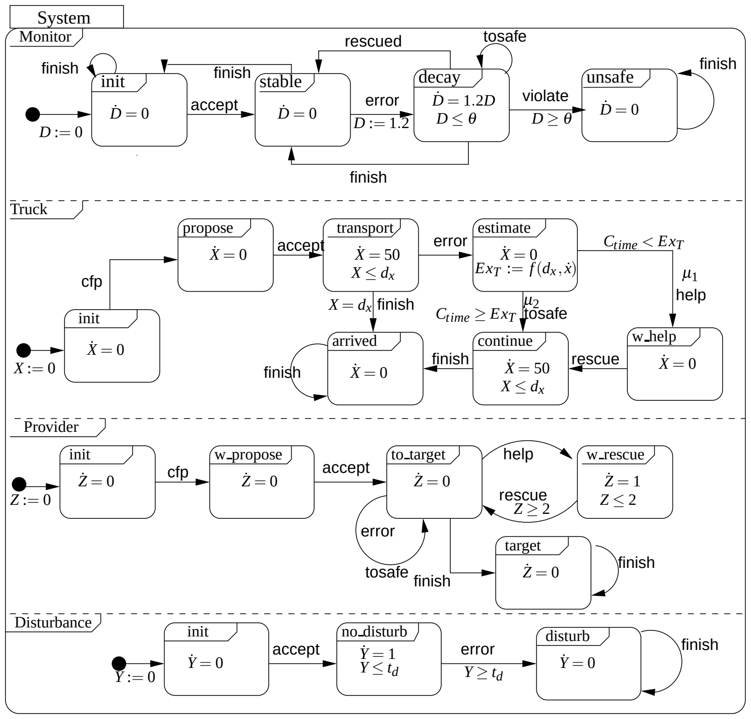


Fig. 7.2. Specification of a logistic scenario as hybrid automata.

occur. Thus, plans need to be revised all the time. Often a significant cost reduction can arise when transportation companies coordinate their actions well. For example, a company (agent) may assign a subtask to some agent either because it can do it more efficiently—as it might be already in the neighborhood—or because the the other agent cannot perform the task at all. Consider this scenario, a customer has a shipment of freight items which is subjected to be decayed. This shipment has to be transported to a certain destination point. Therefore, she/he contacts a transportation service provider for this mission. Then, the transportation service provider assigns a transportation truck to convey the shipment. Assuming that the customer signs a contract with the service provider so that the freight items have to delivered with a certain threshold θ of items' quality, e.g. at most 20% putrefaction of the freight items. Otherwise, the provider has to compensate the customer with a convenient deal. Therefore, for quality assurance and provider's profitable service constraints, the quality of freight items has to be monitored in the truck during the transportation. In case of an exception, e.g. cooling temperature breaks down, the truck has to find a suitable plan to deal with this exception taking into account to utilize its transportation provider business.

In Fig. 7.2, the specification of the previous multi-agent scenario is depicted as hybrid automata. The multi-agent scenario constitutes four agents, *Monitor, Truck, Provider*, and *Disturbance*. The agent *Monitor*, plugged into the truck, observes the occurrence of exceptional errors as well as the putrefaction of the items. The items are putrefied according to the exponential decay function, given as $\dot{D} = 1.2 * D$. When an exceptional error occurs during the transportation, stimulated by the *Disturbance* agent after some time t_d, the Monitor agent alarms the *Truck* with the occurrence of this error. The *Truck* in turn has to make an appropriate decision before the decayed items reach a certain threshold θ. The decision is estimated using the variable Ex_T, according to the remaining distance to the destination point. Here, Ex_T is determined based on the dynamic of distance of the truck to the target. If the expected delivery time is beyond a given critical time C_{time}, then the *Truck* requests help from the transportation service provider, who sends a rescue truck within two hours. However, if the truck estimation is below the critical time C_{time}, then it should continuously transport the shipment according to the current conditions. At the end of transportation, both the customer and the provider check the result of the previous plan.

The objective of the previous scenario is to check that the agents, particularly the truck, will choose the right plan during the course of execution in a way that utilizes the profit of its provider company.

7.4 Planning Model

This section shows the basics components of the conceptual planning model that we use to formulate the planning problem. The model relies on extending the syntax and semantics of hybrid automata. The definition of the model is the same as the definition of hybrid automata defined in Chapter 4, except it contains new decision variables that are used to evaluate the decisions-making. The definition also contains a utility function that assigns cost on transitions. In the following, we show the basic components of the model.

Definition 7.1 (Extended Hybrid Automata). *An extended hybrid automaton is a tuple*
$H = (Q, \mathbb{V}, Inv, Flow, E, Jump, Reset, \Upsilon, Event, Event_H, q_0, v_0)$ *where:*

- *Q is a finite set of control locations.*

- $\mathbb{V} = \mathbb{X} \cup \mathbb{A}$ *is a set of variables, where* $\mathbb{X}$ *is a finite set of n real-valued variables that model the continuous dynamics, whereas* $\mathbb{A}$ *is a set of auxiliary variables that are used as a performance measure to make decisions.* For example, the *Truck* automaton has $X \in \mathbb{X}$ and $Ex_T \in \mathbb{A}$.
- $Inv : Q \to \Phi(\mathbb{X})$.
- $Flow : Q \to \mathbb{D}(\mathbb{X} \cup \dot{\mathbb{X}})$.
- $E \subseteq Q \times Q$.
- $Jump : E \to \Phi(\mathbb{X})$*.*
- *Reset* $: \mathbb{V} \to \mathbb{R}$ *is the updating function, which resets the variables before the control of a hybrid automaton goes from location* q_1 *to location* q_2*. The updating of the variable* $x \in \mathbb{V}$ *is denoted as* $x := Reset(x)$.
  Graphically, one can distinguish between two types of updating depending on types of variables $x \in \mathbb{V}$. Case $x \in \mathbb{X}$, i.e. updating continuous variables then the update is annotated graphically on the transitions $e = (q_1, q_2)$. For example, $D := 1.2$ is the updating of the continuous variable D between location *stable* and *decay* in the automaton *Monitor*. Updating the variables on transitions are omitted, if the value of the variables at end of location q_1 are the same at the beginning of location q_2. Case $v \in \mathbb{A}$ (i.e. updating auxiliary variables), then the update is annotated inside location q_1. The reason is that these variables will be used afterwards as indicators for decision-making on transitions. For example, in the location *estimate* of the *Truck* automaton, $EX_T := f(d_x, \dot{x})$ is updating the auxiliary variable EX_T to the estimated remaining time to deliver the shipment to the target based on the current remaining distance to the target, where $f(d_x, \dot{x}) \in \mathbb{R}$. Semantically, both types of updates are the same. This is because both of them will eventually be executed before the control goes to location q_2 immediately.
- $\Upsilon : E \to \mathbb{R}$ *is the cost function which captures the preference of an agent over e.*

  For example, in the location *estimate*, the *Truck* has preferences to go to either location *w_help* or *continue*, with utilities μ_1 and μ_2 respectively. The utility cost is omitted if there is no preference on the edge *e*.
- $Event_H$ *is a finite set of events.*
- $Event : E \to Event_H$.
- $q_0 \in Q$ *the initial location of the automaton.*
- v_0 *the initial values of the variables* $\mathbb{X}$*.*

As said previously, decision theory is a tool for assessing and comparing the expected utility of different courses of action in terms of the probabilities and utilities assigned to the different possible outcomes. Therefore, we define the preference of an agent based on utilities. We assume, for simplicity, the probabilities of the possible results are equal.

Definition 7.2 (Preference). *Let $q \in Q$ be a control location, and $S = \{e_i = (q, q_i) | 1 \leq i \leq n\}$ be the set of possible alternative transitions connected from q, with respective utilities $\Upsilon(e_i) = \mu_i$. We say e_m is the best preference transition to q iff $\mu_m = Max\{\mu_i | 1 \leq i \leq n\}$.*

The semantics of the planning model is defined in the same way as the semantics of hybrid automata presented in Chapter 4, but with a slight modification on the operational semantics, that is on the definition of the discrete changes of the behavior which is described as the following.

Definition 7.3 (Discrete Changes). *A discrete transition rule between two admissible states $\sigma_1 = \langle q_1, v_1, t_1 \rangle$ and $\sigma_2 = \langle q_2, v_2, t_2 \rangle$ is enabled iff $e = (q_1, q_2) \in E$, $t_1 = t_2$ and $v_1 \models Jump(e)$, and $v_2 \models Inv(q_2)$, where v_2 is the valuation of variables $\mathbb{X}$ as a result from the reset function $Reset(\mathbb{X})$ such that. Additionally, q_2 is the best preference of q_1 in this case an event $a \in Event_H$ occurs.*

7.5 Planning as Reachability Analysis

Having defined the basic extensions of hybrid automata to deal with the planning in a proper way, we can use our CLP presented in 4 to investigate the planning analysis. In particular, we use reachability analysis to analyze the behaviors of the multi-agent team.

Let *Reached* represent the set of reached regions. In terms of CLP, the reachability analysis can be generally specified by checking whether `Reached` $\models \Psi$ holds, where Ψ is the constraint predicate that describes a property of interest. As shown in Chapter 5, the reachability analysis is specified in RCTL as:

$$init \rightarrow \exists \Diamond \Psi$$

init is the predicate characterizing the set of initial states. In the context of planning, the reachability question is equivalent to a plan existence, where Ψ represents the goal of the plan to be reached. Concerning the CLP model, the following encodes the planning query: For example, one can check that there is no existing bad plan, where the shipment arrived to its destination unsafely, i.e. the ratio of decayed items is below 20%. This can be investigated by showing that the location *unsafe* in the *Monitor* agent will not be reached.

```
 ?- reachable(init,Reached),
%%find the plan to reach Goal in Reached
 append(Plan,[Goal|_],Reached).
```

Using the CLP implemenation model and the standard Prolog predicate *append*/3, executing the following query reveals the answer *no* as expected.

```
?- reachable((init1,[0]),(init2,[0]),(init3,[0]),(init4,[0]),
                                                    Reached),
append(Plan,[Goal|_],Reached),
Goal=(Monitor,_truck,_cargo,_disturbance,D,X,Z,Y,Time,Event),
Monitor = unsafe .
```

We are not only interested to find a plan, but also to find the plan that utilizes certain tasks in case of an exceptional error. In the supply chain example, one can check that the truck will choose the best plan that utilizes its company business and at the same time fulfill the customer demands. This can be accomplished by investigating the reachability of the shipment to its destination point with a certain percentage of putrefaction D. For this purpose, the following query should be invoked:

```
?- reachable((init1,[0]),(init2,[0]),(init3,[0]),(init4,[0]),
                                                    Reached),
append(Plan,[Goal|_],Reached),
Goal=(_monitor,Truck,_cargo,_disturbance,D,X,Z,Y,Time,Event),
Truck=arrived.
```

The success of this query means that *append*/3 returns the intended *Plan* to reach *Goal* in the set of possible reached states *Reached*. However, there are several constraints which influence the outcome of this query, such as the time of the unexpected error generated by the *Disturbance* agent and the remaining distance to the destination during the transportation. For example, setting the disturbance time $t_d = 8$ in the supply chain model, the previous query gives the $D \simeq 1.626\%$ upon the truck's arrival to the destination, whereas setting $t_d = 24$, the query gives $D \simeq 5.542\%$. In both cases, the customer's demand is not violated according to the deal with the provider. The contrast between the two values of D results from the truck's decision based on the constraints appeared in the environment. In the first case of t_d, the truck requested a rescue from the provider.In the second case, the truck keeps transporting the shipment without requesting help. The previous analysis can be checked using the following query:

```
?- reachable((init1,[0]),(init2,[0]),(init3,[0]),(init4,[0]),
                                                     Reached),
append(Plan,[Goal|_],Reached),
Goal=(_monitor,Truck,_cargo,_disturbance,D,X,Z,Y,Time,Event),
Truck=arrived,
member(State,Plan),
State=(_monitor,_truck,_cargo,_disturbance,_,_,_,_,_,Event),
Event = rescue.
```

This query checks whether there is a state at which the event *rescue* can be reached in plan. In other words, the query means *does the truck need a rescue?* In the first case of t_d, the query returns with the answer *Yes*, but with *No* in the second case. The interesting thing in hybrid automata is that we can check the timed constraints that occurs during the plan. This type of constraints can be used as an aspect in the decision making, where the agents take suitable actions that comply with a deadline.

8 Hierarchical Model

Hybrid automata may add complexity by specifying multi-agent systems. This is because hybrid automata not only describe the internal behaviors of agents, but also the external interaction among agents. This demands for structured and systematic methods for the specification of MASs, which are able to cope with the complexity of structures. *Statecharts* in this case are helpful. They have the clear advantage of allowing hierarchical specification on several levels of abstraction but are limited to describe the behavior of discrete reactive systems. To bring the advantage of statecharts together with hybrid automata, this chapter combines both formalisms within the same framework. The Chapter presents the formal semantics for this combination and shows how to systematically analyze the dynamic behaviors of systems with this combination. In principle, a straightforward way to analyze a hierarchical machines is to flatten them and to apply verification techniques to the resulting ordinary finite state machines. We show how this flattening can be avoided by providing an implementation with help of constraint logic programming. The implementation serves as a model and verification engine for the proposed combination. The contribution of this chapter has been presented in [Mohammed and Stolzenburg, 2008; Mohammed et al., 2010], which stems from original work of [Furbach et al., 2008].

8.1 Introduction

So far, we have used hybrid Finite State Machines (FSMs) to specify and verify a group of agents. Classical FSMs unfortunately lack of support for modularity, which is very important when modeling complex systems that contain similar subsystems—for the setting of this chapter, we use the term finite state machine and automaton synonymously. All states are equally visible and are considered to be at the same level of abstraction, which makes modeling cluttered and illegible. In practice, to describe complex systems using FSMs, several extensions can be useful to overcome their structural limitations. The most important extension is hierarchy, or what is the so-called *hierarchical (nested) FSM*. Hierarchical FSMs have descriptive advantages over ordinary FSMs. Firstly, super-states offer a convenient structuring mechanism that allows us to specify systems in a gradual refinement manner, and to look at it at different levels of granularity. Such structuring is particularly essential for specifying large FSMs by means of a graphical interface. Secondly, by allowing sharing of component FSMs, one needs to specify components only once and then can re-use them in different contexts leading to modularity and succinct system representations.

One of the existing specification formalisms, which adopts the notation of hierarchy, is *statecharts* [Harel, 1987]. Statecharts have been originally proposed to describe complex reactive systems. The behavior of a reactive system is described as a sequence of discrete events that cause changes in the state of the system. In order to cope with those reactive systems that exhibit continuous timed behaviors, it seems to be advantageous to extend statecharts with continuous actions inside states. This extension allows complex/multi-agent systems to be modeled with different levels of abstraction and provides a formal way to analyze the dynamical behavior of the modeled systems. There are two possibilities of combinations to do so, namely combining statecharts with differential equations or extending hybrid automata with hierarchy. Therefore, both terms hierarchical hybrid automata (HHA) and hybrid statecharts can be used interchangeably.

Modeling, and extending statecharts to include differential equation can be straightforward. However, for the purpose of formal analysis, an important thing is that we need executable models of those systems which can be described in terms of hybrid statecharts. Modeling languages are extremely useful, if they can prove properties of systems being described. The straightforward way to analyze hierarchical state machines is to flatten them to obtain an ordinary FSMs. The flattening process is done by recursively substituting each super-state with its associated FSM. In turn, model checking tools are applied on the resulting ordinary FSM. Such a flattening, however, can cause a blow-up, particularly when there is a lot of sharing. This chapter shows that this flattening can be avoided.

This chapter contributes in extending the statecharts with continuous dynamics to model complex multi-agent systems situated in a dynamical environment. The Chapter also gives an executable model based constraint logic

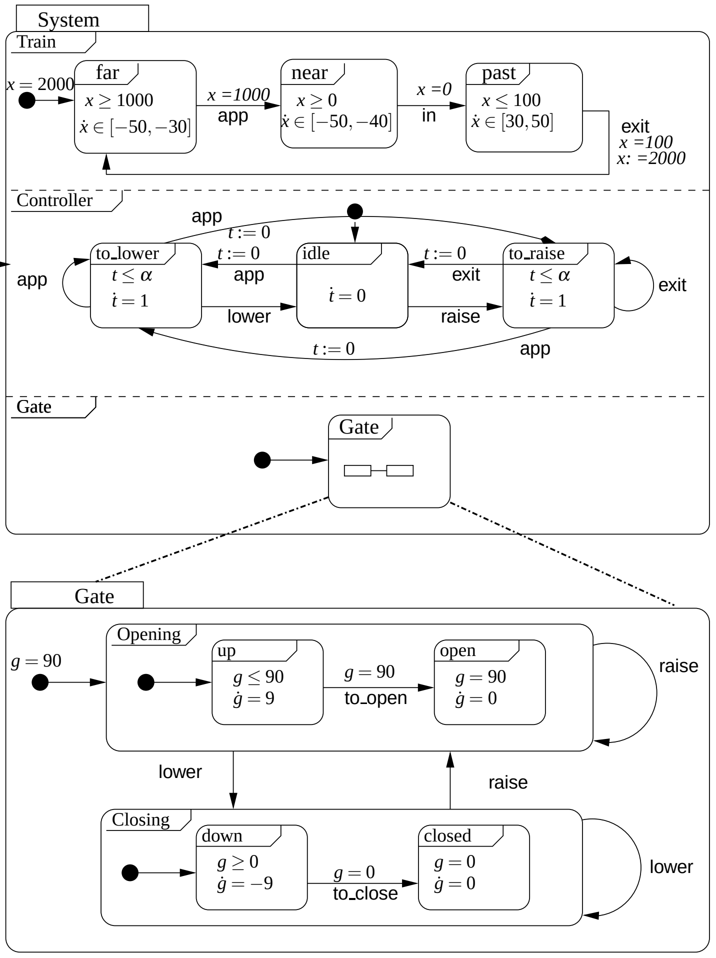


Fig. 8.1. State Hierarchy of train gate controller example.

programming, where the size of the corresponding CLP program is only straight proportional to the size of the given hierarchical hybrid automaton description.

8.2 Statecharts Basics

FSMs have been used extensively in the specification and analysis of reactive systems. In practice when they are applied to larger problems, the models lack to support modularity and become cluttered and illegibal. Statecharts [Harel, 1987] have been introduced to overcome these limitations.Basically, statecharts extend FSMs with several capabilities including hierarchy, and concurrency. Hierarchy is the ability to group states into a super-state, or synonymously an OR state. Hierarchy serves several purposes such as state refinement used mostly for the purposes of top-down design and reduction of transition clutter and transition state dependence. Graphically, hierarchy is usually represented using two drawing techniques namely *explicit nesting* and *coarse states*. Explicit nesting is used when one draw explicitly a lower level state inside a higher level state, such as state *train* inside *System* in Fig. 8.1. A coarse state is a state whose contents are drawn on a separated drawing, such as the state *Gate* shown in Fig. 8.1.

The notation of hierarchical was popularized not only with the introduction of statecharts, but with also other specification formalisms such as *modecharts* [Jahanian and Mok, 1994]. It has become a central component of various object-oriented software development methodologies developed in recent years, such as *OMT* [Rumbaugh et al., 1991], and it has become a part of the unified modeling language (UML) [UML, 2009]. It is commonly available also in commercial software engineering tools, such as *Statemate* and *Rational rose*.

Concurrency in the statecharts denotes orthogonal sub-systems called together a concurrent state. These sub-systems are independent of each other and are therefore drawn separately. Each sub-system can be conceptually regarded as a statecharts in its own. An concurrent state is visually depicted by dashed lines splitting a state. When a system is in a concurrent state, it will be in all if its sub-systems.

Statechart transitions are annotated with events, conditions, and actions. Thus, a transition in a statechart takes the form $event[condition]/action$. Such a transition is shown as a directed edge from a state s1 to state s2. The (informal) semantics of a transition means that if the system is in state s1 and an event occurs and some condition holds, then the system executes an action and changes to state s2 .

8.3 Hybrid Statecharts

In this section we present the definitions and formalism for *hybrid statecharts*. Before we begin the description of their formal syntax and semantics, we should note that we replace the notation of *states* in statecharts with the notation *locations*. This is because, a state in a hybrid automaton describes the evaluation of the continuous variables at a particular time instance at a certain location. Therefore, we will use the term *location* to avoid any confusion that may happen. It should be noted that we do not attempt to handle the maximal fragment of the statecharts languages. Instead, we focus on a representative fragment of hierarchy.

The locations in hybrid statechars are generalized into a set Q of locations, which is divided into three disjoint sets: Q_{simple}, Q_{comp}, and Q_{conc} called *simple*, *composite* and *concurrent* locations. There is one designated start location which is the topmost location in the hierarchy. In essence, the locations of plain hybrid finite state machines correspond to simple locations in the hybrid statecharts. Based on this, we will now introduce the concepts of hybrid statecharts. In the following, we adopt and slightly change the basic definitions of [Furbach et al., 2008].

8.3.1 Syntax

Similar to the definition of syntax of hybrid automata, hybrid statecharts contain the basic components of hybrid automata including the set of real variables $\mathbb{X}$ representing the continuous flows, invariants inside locations, jump conditions, and the initial state. However, the hierarchy of locations is the key difference to hybrid automata. Therefore, we will only concentrate on the locations hierarchy[1] which will be defined in the following.

Definition 8.3.1 (Hierarchy components) *The basic components of hybrid statecharts are the following disjoint sets:*

Q : *a finite set of locations, which is partitioned into three disjoint sets: Q_{simple}, Q_{comp}, and Q_{conc}—called simple, composite and concurrent locations, containing one designated* start location $q_0 \in Q_{comp} \cup Q_{conc}$.

In order to introduce a concrete example of the previous definition, let us look at the hierarchical train gate controller example of Fig. 8.1. The locations *far*, *idle*, and *down* are example of simple locations. The location *System* is a concurrent location and the start location of the model too. The locations *Train*, *Controller*, *Gate*, *Opening* and *Closing* are composite locations.

Definition 8.3.2 (Location hierarchy) *Each location q is associated with zero, one or more* initial locations $\alpha(q)$*: a simple location has zero, a composite location exactly one, and a concurrent location more than one initial location. Moreover, each location $q \in Q \setminus \{q_0\}$ is associated to exactly one superior state $\beta(q)$. Therefore, it must hold $\beta(q) \in Q_{conc} \cup Q_{comp}$. A concurrent state must not directly contain other concurrent ones and all transitions*

[1] see Chapter 4 for the basic components of a hybrid automaton

(q_1, q_2) *must keep to the hierarchy, i. e.* $\beta(q_1) = \beta(q_2)$. *Variables* $x \in \mathbb{X}$ *may be declared locally in a certain state* $\gamma(x) \in S$. *A variable* $x \in \mathbb{X}$ *is* valid *in all states* $s \in S$ *with* $\beta^n(s) = \gamma(x)$ *for some* $n \geq 0$ *(i.e. in all states below* $\gamma(x)$ *in the state hierarchy), unless another variable with the same name overwrites it locally.*

For the example in Fig. 8.1, according to the previous Def., it holds e.g.:

$$\begin{array}{ll}
\alpha(Train) = far & \alpha(Gate) = opening \\
\alpha(opening) = up & \alpha(controller) = idle \\
\alpha(System) = \{Train, Controller, Gate\} & \beta(near) = Train \\
\beta(Train) = System & \gamma(x) = Train \\
\gamma(g) = Gate & \\
\gamma(t) = controller &
\end{array}$$

The function β from the previous definition naturally induces a location tree with q_0 as root. This tree is formed as a result of the semantics between states which we will define as the following.

8.3.2 Semantics

As known, the semantics of hybrid automaton are described in terms of alternating sequences of states. A state is a control location and the valuation of the real variables at each time instance. Different to hybrid automata, the control location of statecharts may be composite or concurrent locations. Therefore, state machines, which describe the behaviors of systems can not be described by simple sequences of states, but by configurations, which are trees of locations. While processing the behavior of the state machines, each composite location only contains one active control location. More specific, whenever a location is in a configuration and it is composed location, then each of its direct sub-automata must also contribute to the configuration and vice versa. In the case of concurrent location, each of the sub-automata contributes to the configuration, if their parent is in that configuration, i.e. one location of respective automata belong to the current configuration. In our example Fig. 8.1, this means that whenever the model in a location *System*, also *Train*,*Gate*, and *Controller* are active.

Fig. 8.2 shows the configuration tree of the example of Fig. 8.1. A configuration of the given statecharts is indicated by the thick lines. Let us now define the notion configuration more formally.

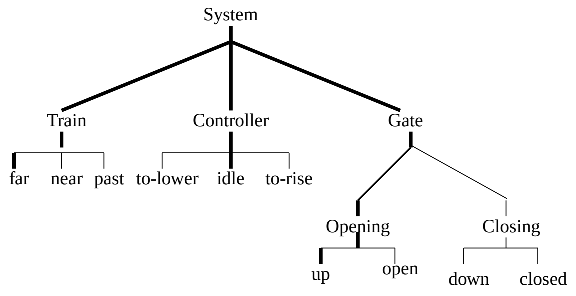


Fig. 8.2. Location hierarchy and configuration tree (thick lines).

Definition 8.3.3 (Configuration and Completion) *A* configuration *c is a rooted tree of locations where the root node is the topmost initial location q_0 of the overall state machine. Whenever a location q is an immediate predecessor of q' in c, it must hold $\beta(q') = q$. A configuration is* completed *by applying the following procedure recursively as long as possible to leaf nodes: if there is a leaf node in c labeled with a location q, then introduce all $\alpha(q)$ as immediate successors of q.*

As presented in Chapter 4, a hybrid automaton may change in two ways: *discretely*, from location q_1 to another location q_2, when the transition $e \in E$ between the two locations is enabled (i.e., the jump condition holds) and *continuously* within a control location $q \in Q$, by means of a finite (positive) time delay t. The semantics of hybrid statecharts can now be defined by alternating sequences of discrete and continuous steps between configurations. we assume that discrete state changes happen in zero time, while continuous steps (within one state) may last some time.

Definition 8.3.4 (Operational Semantic) *The state machine starts with the* initial configuration, *i.e. the completed topmost initial state s_0 of the overall state machine. In addition, an initial condition must be given as a predicate with free variables from $\mathbb{X}$. The current* situation[2] *of the whole system can be characterized by a triple (c,v,t) where c is a configuration, v a valuation (i.e. a mapping $v : \mathbb{X} \to \mathrm{IR}^n$), and t the current time. The* initial situation *is a situation (c,v,t) where c is the initial configuration, v satisfies the initial condition, and $t = 0$. The following steps are possible in the situation (c,v,t):*

discrete step: *a discrete/micro-step from one configuration c of a state machine to a configuration (c',v',t) by means of a transition $(q,q') \in E$ with*

[2] situation are used instead of state to describe the time instance of a configuration

some jump condition in the current situation (written $c \to c'$) is possible iff:

1. *c contains a node labeled with q;*
2. *the jump condition of the given transition holds in the current situation (c, v, t);*
3. *c' is identical with c except that q together with its sub tree in c is replaced by the completion of q';*
4. *the variables in X are set by executing specific assignments.*

continuous step: *a continuous step/flow within the actual configuration to the situation (c, v', t') requires the computation of all $x \in \mathbb{X}$ that are valid in c at the time t' according to the conjunction of all state conditions (i.e. flow conditions plus invariants) of the active locations $q \in c$, where it must hold $t' > t$.*

From the previous semantics, a state machine is initially in a configuration derived from the initial top most location. This derivation is performed in a top-down manner; that is the root of the state machine contributes to the initial configuration by its initial location. If some location in the configuration is refined to further automata, then these automata must contribute their initial states to the initial configuration as well.

It should be noted that invariants of the definition of hybrid automata presented in Chapter 4 are merged here with the flow conditions in continuous steps (see Def. 8.3.4). In particular, while jump conditions are checked during a discrete transition, flow and invariant conditions are only tested at the beginning and at the end of a continuous flow within one configuration, i.e. only at the boundaries.

8.4 Hierarchy Implementation with CLP

Now we will show how to implement an abstract state machine for the previous hybrid statecharts. In this implementation, hierarchies and concurrency are treated more explicitly [Mohammed and Stolzenburg, 2008; Mohammed et al., 2010]. This leads to a lean implementation of hybrid automata, where efficient CLP solvers are employed for performing reachability analysis.

Fig. 8.3 shows parts of the abstract state machine in Prolog, namely the code for completion and for performing discrete and continuous steps according to Def. 8.3.4 and 8.3.3. Discrete steps take zero time. Continuous steps remain within the same configuration but the variable values may differ. The

```
complete(T,Rest,State,[State:Var|Complete]):-
 init(T,State,[Var|Rest],Init,_),
 maplist(complete(T,[Var|Rest]),Init,Complete).

discrete(T,Rest1,Rest2,[State1:Var1|_],[State2:Var2|Conf]):-
 trans(T,State1,[Var1|Rest1],State2,[Var2|Rest2]),
 complete(T,Rest2,State2,[State2:Var2|Conf]).
discrete(T,Rest1,Rest2,[Top:Var1|Sub],[Top:Var2|Tree]) :-
 Sub \= [],
 maplist(discrete(T,[Var1|Rest1],[Var2|Rest2]),Sub,Tree).

continuous(T1,T2,Rest1,Rest2,[State:Var1|Sub],
                                       [State:Var2|Tree]):-
 flow(T1,T2,State,[Var1|Rest1],[Var2|Rest2]),
 maplist(continuous(T1,T2,[Var1|Rest1],[Var2|Rest2]),Sub,Tree).
```

Fig. 8.3. CLP code of the abstract state machine.

flow conditions of active locations (in the configuration) must be applied, as time passes. In this context, configurations are encoded in Prolog lists, where the head of a list corresponds to the root of the respective configuration tree. In addition, each location is conjoined by a colon (`:`) with its list of local variables. According to Def. 8.3.3, the completed start configuration will be represented as shown below. The event and the delay α—represented by the variable `Alpha`—are treated as global variables of the whole system.

```
[system:[none,Alpha],
   [train:[2000],[far:[]]],
   [gate:[90],[open:[]]],
   [controller:[0],[idle:[]]]]
```

The corresponding configuration is also shown as a tree in Fig. 8.4 (left). Of course, trees could be represented more efficiently, i.e. consuming less space, rather than by Prolog lists as shown above. But the use of lists is straightforward and allows us to implement the abstract state machine for hybrid statecharts (Fig. 8.3) within only a dozen lines of CLP/Prolog code. By this way, explicit composition of automata is avoided. For each state, its initial states have to be declared in addition to their continuous flow conditions. For all discrete transitions, the jump conditions have to be stated. Local variables are expressed by a nested list of variables valid in the respective state. Since the abstract state machine is of constant size and the abstract machine computes complex configurations only on demand, there is a one-to-one correspondence between the elements of the hybrid statecharts and their CLP/Prolog implementation. Thus, the program size is linear in the size of the model.

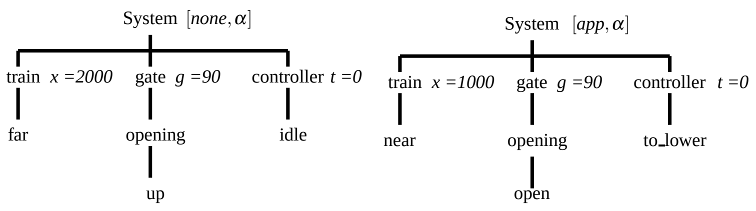


Fig. 8.4. Configuration trees of the running example.

In the concrete implementation of the example, the overall start state s_0 is indicated by the predicate `start`, while `init` defines the initial states for each state (α values according to Def. 8.3.2). The flow and the jump conditions have to be expressed by means of the predicates `flow` and `trans`. The reader can easily see from Fig. 8.5 [3] that the size of the CLP program is only straight proportional to the size of the given hybrid statecharts because there is a one-to-one correspondence between the graphical specification and its encoding in Prolog, whereas computing the composition of concurrent automata explicitly leads to an exponential increase. Since the overall system behavior is given by the abstract state machine (Fig. 8.3), this approach is completely declarative and concise.

The reachability analysis of the abstract state machine uses iterative deepening search strategy. After one continuous and one discrete step according to Def. 8.3.4, the configuration shown below (see Fig. 8.4, right) will be reached after 0.0–25.0 s. The event *app* occurs, when the train has traveled 1000 m. Then, the simple states *near* and *to_lower* in the composite states *train* and *controller* are entered respectively.

8.4.1 Testing Hierarchy

As far as we know, there is no standard benchmark to test the hierarchy. Instead, a flat version of hybrid automata should be given to model checkers for the purpose of verification. Thus, to be able to check the feasibility of our approach, we use flat benchmarks. We experiment the standard benchmarks presented in Chapter 6 to test the HHA. Querying these benchmarks check safety properties (cf. Fig. 8.6). Firstly, in the *scheduler* example, the safety

[3] See appendix A for the rest of the example

```
%%% system
start(system).
init(T,system,[[Event,Alpha]],[train,gate,controller],_) :-
        Event = none.
flow(T1,T2,system,[[Event,Alpha]],[[Event,Alpha]]).


%%% train
init(T,train,[[X]|_],[far],system) :-
        X $= 2000.
flow(T1,T2,train,_,_).

init(T,far,[[]|_],[],train).
flow(T1,T2,far,[[],[X1]|_],[[],[X2]|_]) :-
        X2 $>= 1000,
        X2 $>= X1-50*(T2-T1),
        X2 $=< X1-40*(T2-T1).
trans(T,far,[[],[X],[Event1,Alpha]],far,[[],[X],
                                        [Event2,Alpha]]):-
        Event2 = lower ; Event2 = raise.
trans(T,far,[[],[X],[Event1,Alpha]],near,[[],[X],
                                       [Event2,Alpha]]):-
        Event2 = app,
        X $= 1000.
```

Fig. 8.5. A part of the HHA implementation of the train example.

```
[system:[app,Alpha],
   [train:[1000],[near:[]]],
   [gate:[90],[open:[]]],
   [controller:[0],[to_lower:[]]]
```

property is to check whether a certain task (with number 2) never waits. Secondly, in the *temperature control* example, it has to be guaranteed, that the temperature always lies in a given range. Thirdly, in the *train gate controller* example, the safety property has to make sure that the gate is closed whenever the train is within a distance less than 10 meter toward the gate. The second version of the train gate controller example is used to calculate a parameter analysis, i.e. finding a condition to be hold on a parameter/variable which guarantees the satisfaction of the safety property. In the train gate controller example, parameter analysis aims at finding the condition on the parameter α. Last but not least, in the *water level* example, the safety property is to make sure that the water level is always between given thresholds (1 and 12).

The benchmarks can be solved by all considered implementations, namely HyTech, and the HHA implementation with CLP, within milliseconds. Fig. 8.6 shows the concrete run-time results (in milliseconds), by comparing Hytech with HHA. It reveals that the CLP/HHA implementation allows the briefest problem formulations because of the use of the abstract state machine, but since the time points of performing discrete steps are not computed explicitly, it is susceptible to rounding errors. In order to guarantee termination of the CLP implementations, the search depth is fixed in advance. For the CLP/HHA implementation, the number of continuous plus discrete steps is given. These limits are also listed in the table.

Example	**HyTech**	**CLP/HHA**	
	seconds	seconds	steps
Scheduler	0.12	0.34	12
Temperature Controller	0.04	0.02	12
Train Gate Controller	0.05	0.03	12
Train Gate Controller 2	0.10	0.02	9
Water Level	0.03	0.02	8

Fig. 8.6. Experimental results.

In the run of the abstract state machine, the way of setting the depth might restrict the reachability analysis such that the reached configurations could be incomplete to check the reachability of certain queries. Hence, one might get negative results. A possible solution to this problem is to set the depth to be big enough. But this raises the problem of the performance of run-time. A suitable way to handle the reachability analysis is to find exact reachability of configurations—similar to the way of computing the reachability of regions presented in Chapter 4—by running the abstract state machine until reaching to fixed configurations, i.e. finding cycles. In [Schwarz et al., 2010; Mohammed and Schwarz, 2009] the hierarchical implementation has been refined to perform that process.

8.5 Related work

For the advantages to model the dynamical behaviors of MASs in hierarchical manner, several successful approaches have been proposed to provide hierarchy. In the following we show some of those approaches that are relates to the the work presented in this chapter.

On of the early approaches discussing model checking of hierarchical state machines is presented in [Alur and Yannakakis, 1998]. In this approach the verification of hierarchical finite machines are performed without flatting the hierarchy. Several algorithms are presented for the model checking problem. In particular, this approach adopts a depth first search algorithm that performs the reachability analysis. This approach, however, does not consider the continuous behaviors within the hierarchical state machines. A similar work to this approach is presented in [Gnesi et al., 1999]. This work presents a simple model-checking approach to verify UML statechart diagrams. However, this approach is very simple and restricted in the sense that it does not handle the hierarchical notations of statecharts. Moreover, the continuous dynamics are not be considered.

Modecharts [Jahanian and Mok, 1994] is one of the early extensions of the statecharts which extends statecharts with timing notations. The formal semantics of *Modecharts* are defined in terms of real time logic of [Jahanian and Mok, 1986] whose time is restricted to discrete domain. A similar work is presented in [Kesten and Pnueli, 1991]. This work suggests an extension of statecharts to accommodate continuous and discrete event behaviors. As a result of this extension a language called *timed statecharts* is presented in which each transition of statecharts is annotated by a time interval $[l,u]$ denoting the lower and upper time bounds of that transition. Also, this work proposes what is called *hybrid statecharts* as a further extension to statecharts, which allows to annotate a basic state of statecharts with differential equations. The semantics of those extensions are discussed, but there is no automatic mean to execute these hybrid statecharts.

In contrast to this chapter, there are researches that use components to model hierarchy instead of statecharts. In these researches, atomic components are used to build more complex components in hierarchical manners. Interaction between components takes place by means of shared variables. Synchronization by means of actions is, however, not supported. One of the works adopting this approach is presented in [Henzinger, 2000]. In this work, a language called *Masaccio* is used as a formal model of hybrid dynamic systems. In *Masaccio*, systems are built from two atomic components, namely discrete and continuous components. A model in *Masaccio* is structured in a way which permits hierarchical definition of components. Both types of components can be arbitrarily nested and composed by means of parallel and serial operators. Data can enter and exit a component through variables. Control enters and exits through locations. *Masaccio* supports the assume guarantee principle, where one can separately verify the correctness of each component by assuming that the rest of the components of the systems behave according to their specification. By using this technique, a large verification problem can be decomposed into many smaller verification problems. In this approach, however, there is no formal verification on the model as a whole. *Charon* [Alur et al., 2000, 2001] is quite similar to this work, which also addresses the hierarchical issues within hybrid systems. In the framework of *Charon*, the basic building block is represented by an agent that communicates with its environment by means of shared variables. Agents can model distinct components of the system whose executions are all active at the same time. The agents in this approach are classified into two types: primitive and composite agents. The primitive agents form the primitive types or basic building blocks of the architectural hierarchy. The composite agents are derived by parallel composition of the primitive agents. The internal behavior of each agent is represented by modes, which represent the discrete and continuous activities of the agent. Each agent consists of one or more distinct modes that describe the flow of control inside an agent. In addition to variables, continuous dynamics, invariants, and guards, modes contain control points which provide entry and exit points to the flow of modes. Although *Charon* can perform some kind of formal analysis, particularly checking invariants at run time and reporting an error when an invariant is violated and no transition is enabled, it only focuses on simulation rather then formal analysis. *SHIFT* [Deshpande et al., 1997] is a similar simulation approach allowing hierarchical specifications of hybrid systems.

Other works proposes to model hierarchical machines with hybrid automata or with less expressive subclasses of hybrid automata. However, to analyze behaviors of hierarchical models, they have to be flattened into ordinary state machines and then model checking tools are applied on the flattened parts. An example of these works is presented in [Mller et al., 2003]. In this work, a hierarchical specification of timed automata is presented. To verify a hierarchical model, it has to be transformed to flat-timed automata, which in turn can be used as input for the model checker tool Uppaal [Behrmann et al., 2004]. Similarly, Ruh in [Ruh, 2007] presents a translator tool that automatically converts hybrid hierarchical statecharts, defined as an ASCII-formatted specification, into an input format for the model checker Hytech [Henzinger et al., 1997]. In contrast to these works, we have shown that the hierarchical hybrid automata can be analyzed without getting involved in the flattening process.

9 From Graphical Modeling to Formal analysis

So far, we have shown a framework to model and verify multi-agent systems by means of concurrent and hierarchical hybrid automata. Both the model and the requirements of the system under consideration have to be written in CLP. However, specifying the complete systems using CLP is definitely a tedious, error-prone and undesirable task, particularly when specifying safety-critical or larger systems. To facilitate the process of specification, graphical notations of software engineering are helpful to model systems, but they provide little support for systems analysis. Therefore, to bring the advantages of graphical notations together with automatic verification of formal methods. For this purpose, this chapter aims at simplifying the specification and verification process by introducing *HieroMate* a tool environment with a constraint logic programming core that allows us to specify multi-agent systems graphically and verify them automatically. The Chapter demonstrates this on a multi-agent system scenario taken from the Robocup rescue. The contribution of this chapter has been published in [Mohammed and Schwarz, 2009; Schwarz et al., 2010].

9.1 Introduction

In the previous Chapters 4 and 8, we have shown how to formally specify and automatically verify systems at different levels of abstraction using hybrid automata. We have presented two structural views of systems, namely concurrent/flat and hierarchical views. It is well known that the formal specification targets a precise and unambiguous description of the behavior of systems under design, whereas the automatic verification by model checking aims at verifying a desired specification.

To automatically verify a certain model, one needs to translate such model into an executable format written in a language executed by model checkers. Generally, to specify and verify a certain model, two alternatives can be used to achieve this: either designing the model prior to put it in a textual representation format convenient to a model checker, or starting to specify the scenario directly with the suitable description languages, which is definitely a tedious and undesirable work, particularly when specifying safety-critical systems. An example of these textual languages adopted throughout this thesis is CLP by which we encode both models and specification of properties of hybrid automata. However, writing models together with the specification of properties with a textual language in general or with CLP in more specific is a difficult, cumbersome and error-prone task for several reasons. To model and specify a certain problem, one needs to write hundreds of lines of CLP, which becomes difficult to grasp the meaning of the whole system. To avoid side effects that may result in unwanted behavior, CLP code has to be done carefully. Consequently, this may require a longer specification time and expert personnel. Nevertheless, the possibilities of error occurrence are highly increased and difficult to discover. Additionally, the direct use of logic to build a model is often claimed to be an obstacle for systems engineering. To cope with these limitations, the graphical representation taken from software engineering can be helpful. Intuitively, graphical representations have advantages over textual representations. They are less syntactical language from users' prospectives and easy to develop. They do not require highly experienced users, and hence are favored by lot of users. However, for their informal specification, they provide little support for analysis systems. To bridg this gap, this chapter combines the advantages of the graphical notations together with formal verification. In particular, this chapter presents a tool environment *HieroMate* with a constraint logic programming core that allows us to specify and hence verify multi-agent systems. With *HieroMate* the process of specifying a certain model is done in the form of graphical state transition diagram annotated with mathematical formalisms, which in turn can be verified directly. The informal graphical notations are invisible converted into formal executable specifications. In this way, it is sufficient for the user to focus only on the specification process rather than focusing on both specification and the CLP implementation. This can reduce mistakes which may occur due to the CLP implementation. The *HieroMate* tool accepts specifications and properties to be proven by a visual interaction. Then it generates an intermediate CLP, which can be verified by means of reachability analysis using appropriate constraints solvers and state machine, written in CLP as well. *HieroMate* supports different views of model specification, namely concurrent/flat and hierarchical view. To our knowledge, there is no tool that supports the integration of graphical notations and formal verification of hybrid automata with these views. To convey how the various parts of *HieroMate* are used, the Chapter provides a multi-agent system example taken from the RoboCup rescue.

The rest of this chapter is organized as follows: Sec.9.2 describes and specifies a Robocup rescue multi-agent system scenario, which is taken as an a running example to demonstrate the tool *HieroMate*. This scenario is a modified version of that one existing exists in [Furbach et al., 2008]. Sec.9.3 goes through the details of the tool. Finally, Sec.9.4 shows related work.

9.2 Robocup Rescue Scenario

In the RoboCup rescue simulation league [Tadokoro et al., 2000], a large scale disaster is simulated. The simulator models part of a city after an earthquake. Buildings may be collapsed or are on fire, and roads are partially or completely blocked. A team of heterogeneous agents consisting of police forces, ambulance teams, a fire brigade, and their respective headquarters is deployed. The agents have two main tasks, namely finding and rescuing the civilians and extinguishing fires. An auxiliary task is the clearing of blocked roads, such that agents can move smoothly. As their abilities enable each type of agent to solve only *one* kind of task, e.g. fire brigades cannot clear roads or rescue civilians, the need for coordination and synchronization among agents is obvious in order to accomplish the rescue tasks.

Consider the following simple scenario. When a fire breaks out somewhere in the city, a fire brigade agent is ordered by its headquarters to extinguish the fire. The fire brigade moves to the fire and begins to put it out. If the agent runs out of water it has to refill its tank at a supply station and returns to the fire to complete its task. If the fire is not out within a certain period of time, it will get out of control and cannot be extinguished by the brigade anymore. If enough water could be added to the fire, it will be extinguished and the fire brigade agent is idle again. An additional task the agent has to execute is to report any discovered injured civilians. The whole scenario is modeled as hybrid automata in Fig 9.1. It includes models of the fire, civilians, a fire station and a fire brigade agent.

The fire will initially start in the first 10 minutes of the scenario, the concrete time point is not defined. This is modeled by the location *nofire* and the clock variable *boom* which is restricted to values less then 600. The transition

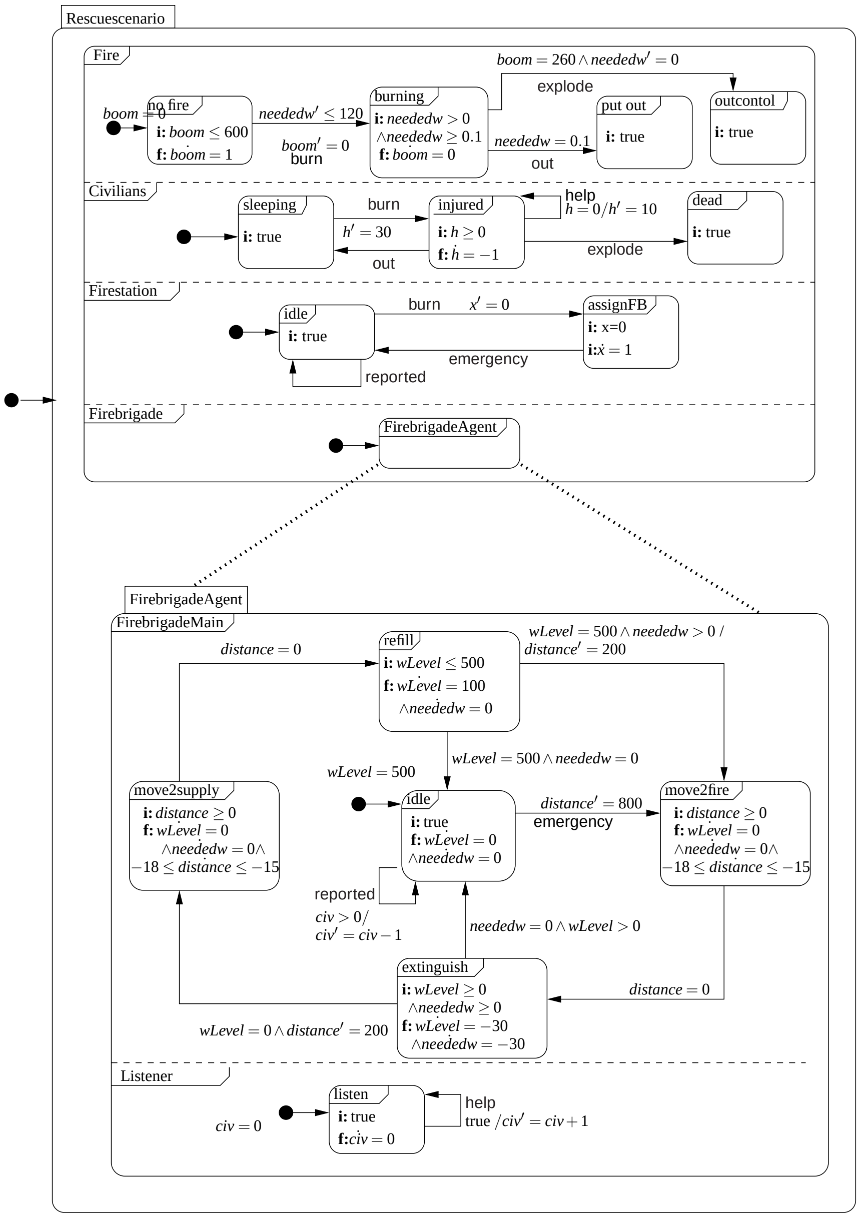


Fig. 9.1. The specification of the rescue scenario.

from of this location may fire at any time and will lead to the location *burning* which models the state of the fire where it could be extinguished. The amount of water that is needed to put it out is modeled by the variable *neededw* that is set to some value less then 1200 when entering the location. This means that in the beginning it needs less than 1200 liters of water to extinguish the fire. There are two possible follow-up locations for *burning* namely *putout* that is reached if enough water was added by the fire brigade and *outofcontrol* that is reached if that is not the case before some timer runs out.

In this simple scenario the civilians are modeled to be sleeping initially. When the fire breaks out, they will wake up and call for help every 30 seconds. If the fire is put out they will sleep again, if the fire gets out of control, they will die.

The task of the fire station is to assign a fire brigade to a fire. As soon as the fire is discovered, the fire station assigns a fire brigade to extinguish it. In this simple example there is only one brigade agent, so the task of the fire station is rather trivial.

As depicted in Fig. 9.1, The specification of the fire brigade consists of the main control structure *FirebrigadeMain* which models the behavior of the agent and a *Listener* that records the number of discovered civilians. The behavior of the agent *FirebrigadeMain* consists of five control locations corresponding to movements (*move2fire*, *move2supply*), extinguishing (*extinguish*), refilling the tank (*refill*), and an idle location (*idle*). The behavior of *FirebrigadeMain* starts in the *idle* location and jumps to the *move2fire* location, when it is assigned to a fire by the fire station. The location *move2fire* models the movement of the fire brigade towards the fire. The distance between fire and the the fire brigade is modeled by the variable *distance* which is set to be less then 800 meters in the beginning. The fire brigade moves with some speed between 15m/s and 18m/s towards the fire. This is modeled by bounding the derivative of the variable *distance* between -18 and -15. After it arriving on the site of the fire, the fire brigade tries to extinguish it. This is modeled by decreasing the value of *wLevel*, which models the water level in the tank, and the water needed to put out the fire, *neededw*, by the same rate. If the water in the tank runs out, the fire brigade has to move to the next refill station that is set to be 200 meters away. The movement is modeled analogously to the movement to the fire. After the tank is refilled, the fire brigade moves towards the fire again. After the fire is put out or is out of control, the fire brigade becomes idle again and reports any found civilians.

It should be obvious that even in this simple case with very few components, it is difficult to see if the agent behaves correctly. Important questions like:

- Does the fire brigade agent try to extinguish without water?
- Will every discovered civilian (and only those) be reported eventually?

depend on the interaction of all components and cannot be answered without an analysis of the whole system.

Fig. 9.1 shows the RoboCup scenario depicted as a graphical state transition diagrams. Generally, state transition diagrams have been applied successfully for MAS, particularly in the RoboCup, a simulation of (human) rescuer with real or virtual robots [cf. Arai and Stolzenburg, 2002; da Silva et al., 2004], in particular for the teams *RoboLog Koblenz* (two-dimensional simulation league) and *Harzer Rollers* (standard four-legged league) [Murray et al., 2002; Ruh and Stolzenburg, 2008]. In what follows, we will demonstrate the use of this scenario with the tool *HieroMate* and present some exemplary model checking tasks.

9.3 The HieroMate tool

The aim of *HieroMate* is to use graphical notations to make the process of specifying system easier or more approachable to average users. The idea of *HieroMate* relies on translating a graphical model into a hidden CLP specification, which is verified using a proper state machine encoded with CLP too. A normal session with *HieroMate* is as follows: After creating a model with hybrid automata, and specifying the requirements graphically, the designer invokes *HieroMate* to verify it. The tool automatically converts the model into constraint logic programming specification. Then the verification is performed using the reachability analysis.

This section describes the tool *HieroMate* in more detail by showing the internal architecture view. The section demonstrates step by step how to describe a normal session with *HieroMate*. Finally, it shows how several properties of the robocup scenario can be checked.

9.3.1 HieroMate at a Glance

The tool *HieroMate* is composed of three layers, namely the graphical user interface , an internal constraint logic program, and state machine with constraints solver back-end. The overall architecture of *HieroMate* environment is shown in Fig. 9.2. The different layers are separated with dashed lines. The first layer contains a graphical user interface of *HieroMate*, which serves the user to do several activities. First, it enables the user to construct or edit a certain model. Moreover, it helps the user to edit and specify properties of the model. Furthermore, it informs the user with the answer of checking the model against properties of interest.

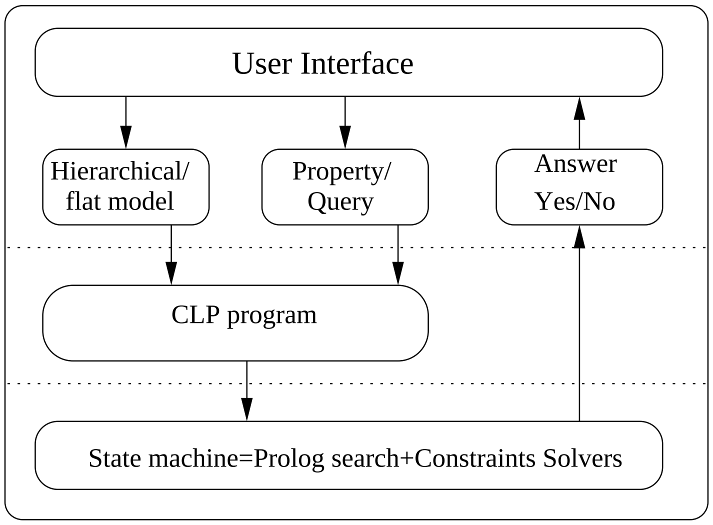


Fig. 9.2. The Architecture view of HieroMate environment.

The second layer contains an internal constraint logic program. Inside this layer, the graphical model together with the specification of properties, constructed in the graphical user interface layer, are converted into a constraint logic program. Usually this layer is hidden to the user but the tool facilitates to give a view for the constructed constraint logic program.

Finally, the lowest layer contains a state machine, written in CLP, which uses the prolog search together with constraints solvers for checking the CLP program created from the previous layer. Like its predecessor layer, this layer is hidden to the user. Hiding the layers to the user helps to reduce the errors, which might occur as the direct access of the CLP program with the user.

Since the graphical user interface is the only visible layer to the *HieroMate* users, in the following we will show a normal session to invoke this interface. We depict it with the example described in Sec.9.1.

9.3.2 The Graphical Interface

The graphical user interface of the *HieroMate* enables the user to create and edit a graphical model of hybrid automata. A model under construction appears on a workstation display as shown in Fig. 9.3. In this workstation, the pull-down menus (*File*, *Edit*, *Automaton*) at the top of the display contain commands for storing, retrieving and editing models of hybrid automata. With the pull-down menu *Automaton*, one can choose the type of the structural view of the model under construction; that is whether it is flat or hier-

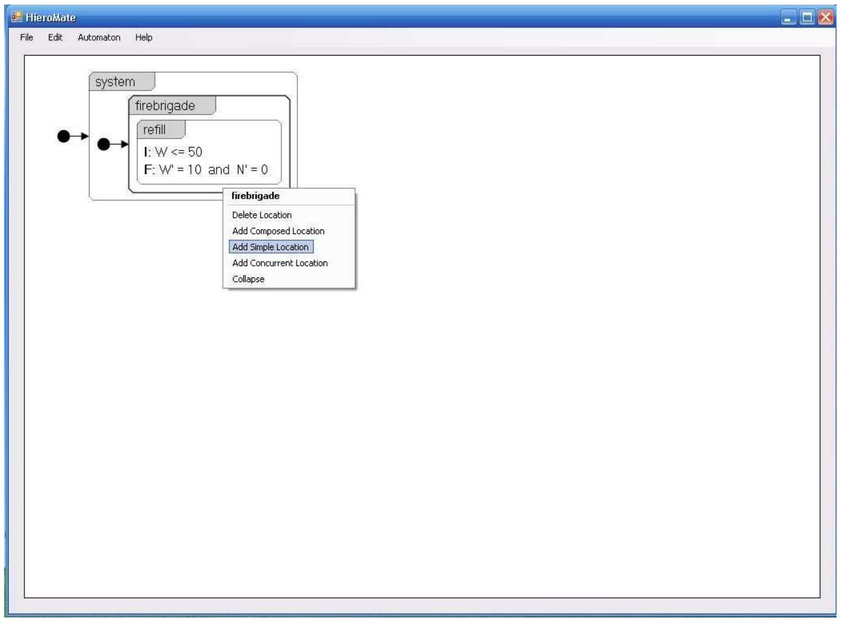
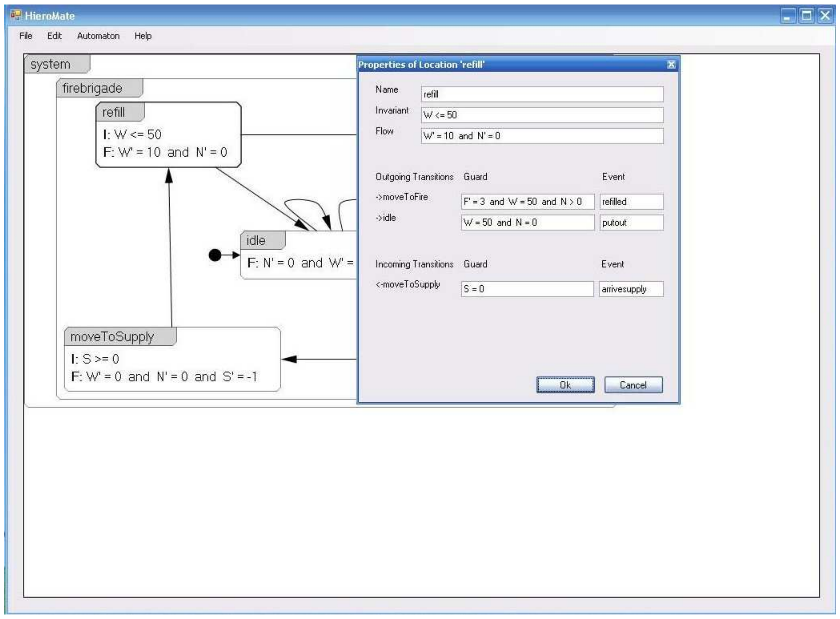


Fig. 9.3. The graphical interface of HieroMate.

archical. Additional function of the menu is that it contains command *Verify* which invokes the window of specifying the properties for the model checking purposes.

The first step to create a model is to start a new working area, then right clicking on any empty working area, a pop-up menu appears containing several items, which help the user to create and manage his/her model. Among of these items, as shown in Fig. 9.3, the user has the ability to begin creating simple, composed or concurrent locations of the model. After creating the locations of the model, the user can create a transition between any two locations by drawing a directed edge from one location to another.

Once both locations and transitions have been created, their properties can be edited. Clicking on a particular location, for example, causes a property window to appear, by which the user can edit the name, invariant, and flow of the location, as illustrated in Fig. 9.4. It should be noted that most of these actions are context-sensitive so only legal options are shown and executed. While editing the flow of a variable x in a non-prime form, i.e x', the tool marks this specification with red colored font, which means there is a violation of the syntax made by the user. In addition to edit the properties of locations, the user can edit the jump condition and synchronization label of transitions.



Properties of Location 'refill'

Name	refill	
Invariant	W <= 50	
Flow	W' = 10 and N' = 0	
Outgoing Transitions	Guard	Event
->moveToFire	F' = 3 and W = 50 and N > 0	refilled
->idle	W = 50 and N = 0	putout
Incoming Transitions	Guard	Event
<-moveToSupply	S = 0	arrivesupply

Fig. 9.4. Editing locations in HieroMate.

Having specified all locations and transitions, the complete model can be ready for checking against properties of interest. Fig. 9.5 shows the specification of the robocup rescue scenario with *HieroMate*. Now this model can be stored or modified. Its appearance can also be changed by rearranging the locations and transitions by means of drag-and-drop or collapse operation.

For specifying and checking requirements of interest, *HieroMate* includes an interface that helps in achieving these aims. Fig. 9.6 shows this interface as a window entitled with *Verification*. This window is partitioned into three parts: Visual tree of locations, the generated query, and the result of model checking. The visual tree contains all the possible locations of the model being checked. From that tree, the user can mark locations, for the purpose of model checking. There are two points should be noted, while selecting locations. Firstly, marking more than one child of a certain concurrent location means that during checking the requirement, all these locations have to be reached in the same time. Secondly, marking more than one child of a composed location means that at least when of these children has to be reached.

In addition to the visual tree part, the interface contains the generated query part, which contains the automatically CLP query resulted from marking the locations of interest in the first part. This part allows the user to textual

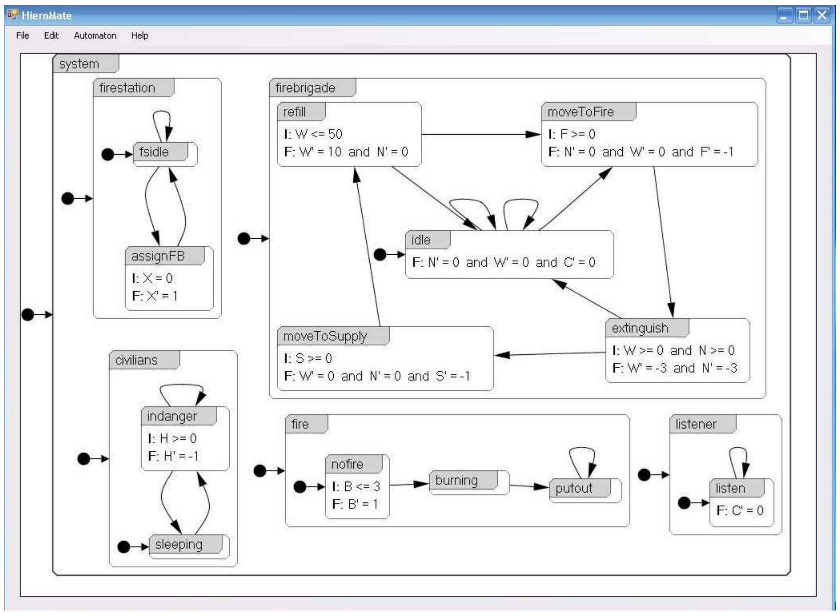


Fig. 9.5. The rescue scenario in HieroMate.

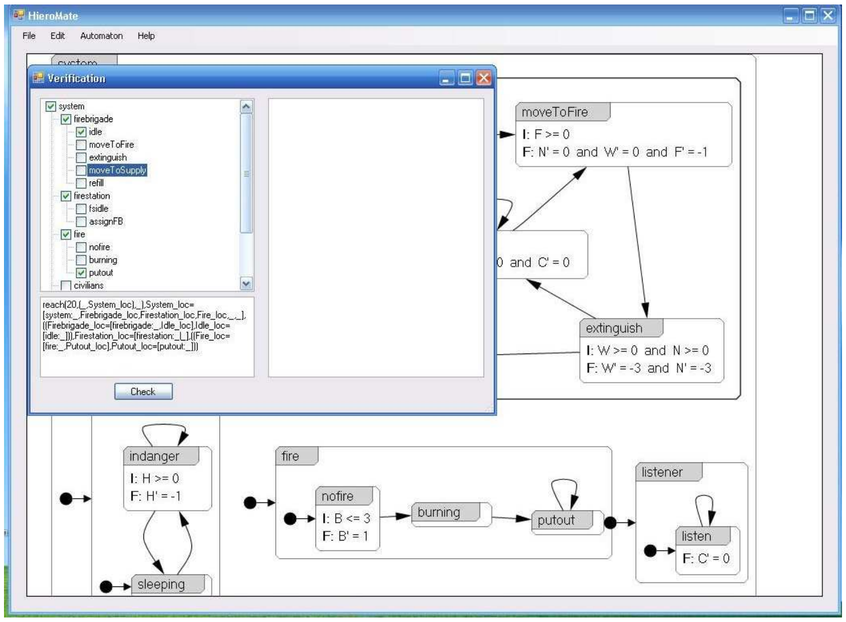
edit the current query to specify more complex requirements, such as specifying the reachability of certain values of particular variables.

Having defined the properties, the interface can check the reachability of theses properties by pressing the *Check* button afterwards. The answer to checking the query is returned in the output part. At the moment, the current output of the tools gives a positive or a negative answer to the query under investigation, but in future it might be shown additional information such as a trace which leads to the specific configuration of the model or the value of certain parameters.

9.3.3 Examples with Model checking

As we already mentioned, the graphical notations are translated into executable specifications, which can be checked by model checking. In *HieroMate* the term of *model checking* refers to *reachability* testing, i.e. the question whether some (unwanted) state is reachable from the initial configuration of the specified system. For this purpose, some exemplary model checking tasks for the rescue scenario can be investigated.

For the behavior specification shown in Fig. 9.1, we conducted several experiments with *HieroMate*. The tool performs reachability tests on the state



```
reach(20,[_,System_loc],_),System_loc=
[system:_,Firebrigade_loc,Firestation_loc,Fire_loc,_,_],
([Firebrigade_loc=[firebrigade:_,Idle_loc],Idle_loc=
[idle:_]]),Firestation_loc=[firestation:_|_],([Fire_loc=
[fire:_,Putout_loc],Putout_loc=[putout:_]])
```

Fig. 9.6. Specifying and verifying properties in HieroMate.

space of the model. This is done by computing all reachable states from the initial state/configuration, and then checking the resulting set for the needed properties. In the following, we present some exemplary model checking tasks for the rescue scenario.

Is it possible to extinguish the fire? When the state of the automaton modeling the fire changes from *no fire* to *burning*, the variable *neededw* stores the amount of water needed for putting out the fire ($neededw = 120$ in the beginning). When the fire is put out, i.e. $neededw = 0$, the automaton enters the state *put out*. Thus the fire can be extinguished, iff there is a reachable configuration c_{out} where fire is in the state *put out*. It is easy to see from the specification that this is indeed the case, as *neededw* is only decreased after the initial setting, and so the transition from *burning* to *put out* is eventually forced.

Does the agent try to extinguish with an empty water tank? To answer this question, we should check the reachability of certain intervals in the continuous valuation of the automaton. The fact that the fire brigade agent tries to put out the fire without water corresponds to the simple state *extinguish* being active while $wLevel < 0$. Note that we must not test for $wLevel \leq 0$, as the state *extinguish* is only left when the water level is zero, so including a check for equality leads to false results.

Won't the fire brigade move to the fire if it is not burning? This is a kind of question that needs to check the reachability of composed locations at the same time. This can be checked by investigating that no location where *fire-brigade* is in location *move2fire* and *fire* is in location *nofire*, or *putout* is reachable

Does the agent report all discovered civilians? We can check properties about the history of a certain state and the reachable states from a given state, this allows more complex questions like this question. Actually, this question contains two properties to be checked:

(a) all discovered civilians are reported eventually and
(b) the agent does not report more civilians than found.

The property (a) corresponds to the fact that from every reachable state there is a state reachable where all discovered civilians have been reported. This again means that the number of transitions labeled with *help* equals the number of transitions labeled with *reported*. Property (b) holds if in the history of each reachable state the number of transitions labeled with *help* is always greater or equal to the number of transitions that are labeled with *reported*.

All properties described above could be successfully proven using our framework.

9.4 Related Work

The graphical notation is becoming more and more accepted, as it is expected that designers will be more familiar with graphical notation. Therefore, several researchers approach to specify behaviors of MASs using graphical notations, namely UML statechart. Murray in [Murray, 2004], for instance, presents an statechart editor called *StatEdit* that is used to graphically specify behaviors MASs with a layered structured. *StatEdit* was intended to design behavior of agents in the RoboCup simulation league. With this editor statecharts can be created and exported to a variety of formats for further processing. Similar to *StatEdit*, there are software engineering tools, such as *Statemate* [Harel et al., 1988] and *Rational rose*, which can be used to graphically specify graphically behaviors of MAS. Neither continuous activities, nor model checking are allowed in these tools however .

In order to combine the formal verification with graphical models, there already exists a number of tools supporting verification of the state machines view of an un-timed UML graphs, particularly statecharts. The tools are doing so by translating graphical models into input languages of existing model checkers. For example, Lilius and Porres in [Lilius and Porres, 1999] present the tool *vUML* for model checking systems, which are modeled by UML statecharts. They use the model checker *SPIN* [Holzmann, 1997] as the underlying verification engine in their tool. Similar to their work, Mikk et al. in [Mikk et al., 1998] present a translator that translates statecharts into, the modeling language of the *SPIN* model checker, *Promela*. The same approach is adopted in Mota et al. [2004]. In their approach, they present a tool that integrates UML models with formal verification. In their tools, a graphical model is translated into an intermediate representations, before the model is checked using the model checker *NuSVM* [Cimatti et al., 2002]. In contrast to our work, all the previous approaches are restricted to model discrete reactive systems.

To graphically specify and hence verify real time systems, several tools are existing. For instance, *Modechart* [Jahanian and Mok, 1994] is a tool which captures graphically time-based requirements. Systems are expressed graphically as concurrent finite state diagrams with delays and deadlines which are specified in modes of the systems. The tool includes a simulator allowing an interactive execution of modechart for consistency and completeness checker. Additionally, the tool includes a verifier which determines whether a timing assertion can be derived from a set of modechart specifications. Another successful graphical tool is *Uppaal* [Behrmann et al., 2004], which is a tool suite for automatic verification of safety and bounded liveness properties of real time systems modeled as networks of timed automata. *Uppaal* consists of a graphical user interface based on *autograph* [Roy and Simone, 1991], which allows a system description to be defined graphically and then verified with a model checking procedures. However, hierarchical structures are not allowed in *Uppaal*. To cope with this, several works propose to specify real-time systems by incorporating the full advantages of the UML models. In particular, there are works extending the standard UML models with time notation [Graf et al., 2006]. For this purpose, several tools have been developed to verify the timed UML models by mapping them to input languages of timed automata, which in turn are verified using existing model checkers of timed automata. For example, Del Bianco et al. in [Del Bianco et al., 2002] use the model checker *Kronos* [Yovine, 1997] to verify their systems, whereas Knapp et al. in [Knapp et al., 2002] use *Uppaal* [Bengtsson et al., 1996] for the same purpose. As we have said, before translating the graphical specifications into suitable representations to model checkers, the previous tools, however, have to flatten any hierarchical specifications. This of course increases the complexity of models. Additionally, those tools together with their underlying verification tools, do not support more hybrid dynamics—e.g. linear hybrid automata.

Several works introduce tools that contain graphical user interfaces serving as graphical input languages to model hybrid systems. However, those tools provide no means of formal verification; instead, they are emerged for simulation purposes of highly complex hybrid systems. Examples of these tools include, *HyVisual* [Cataldo et al., 2003], *Charon* [Alur et al., 2000, 2001], and *Stateflow* [Sahbani and Pascal, 2000].

Part IV

Conclusion

10

Final Remarks

10.1 Summary and Future Work

Multi-agent systems (MASs) are reactive systems consisting of distributed reactive components/agents located in some environment in which they jointly work and interact to achieve their goals. Reactive systems react to stimuli received from the environment by generating corresponding responses. They often appear in safety-critical applications where failure is unacceptable. Their behaviors must be carefully designed with a high degree of precision in order to avoid any undesirable behaviors. The use of rigorous formal methods not only provide ways to precisely describe behaviors of such systems through formal specification, but also to analyze them through formal analysis.

One of the formal approaches that is extensively used to describe behaviors of reactive systems, particularly MASs, is to use state transition systems. A reactive system usually behaves according to a reasoning process of external or internal actions. whenever an action occurs, the behavior of the system moves from one state to another. Finite automata or finite state machines have been successfully used as a medium to model such transition systems. One advantage of state transition systems is that they can be formally analyzed by means of model checking.

Hybrid systems are special forms of reactive systems that continuously react to their environment according to time dependent physical rules. The behavior of such systems involves continuous and discrete actions. The continuous actions of the behaviors arise as an evolution of the systems according to differential equations describing some physical rules, whereas the discrete actions result from the change from one continuous action to another. The classical finite automata are not sufficient to model such types of behaviors, as they can only model the discrete behaviors. Finite automata have been extended to deal with such type of behaviors. This has led to the birth of hybrid automata, which are mathematical formalisms that can formally capture the behavior of hybrid systems. Their formal semantics allow us to prove desirable features and the absence of unwanted properties in the specified systems.

This thesis aimed at approaching the framework of hybrid automata to model and verify behavior of MASs. We have contributed in a number of ways to achieve this aim. In one way of contribution, we have presented a convenient approach which allows us to specify and verify behaviors and requirements of MASs. In this approach we have shown how to attack the state space complexity raised from composition of automata by providing a way that dynamically constructs the composition during the verification phase. Additionally, we have presented a specification language based on extending the well know temporal logic CTL to specify both qualitative and quantitative properties. We have also implemented this approach with the help of constraint logic programming. In this implementation, a model of hybrid automata is converted to an equivalent model of constraint logic program. The specifications of the requirements are converted to suitable queries, which are checked within the constraint logic program by means of reachability analysis.

In another way of contributions, we have provided several extensions concerning the expressiveness of specifying behaviors of MASs. We have introduced a simple approach toward extending the decision making of hybrid automata to deal with deliberative agents' plans. Moreover, we have presented a combination of hybrid automata formal semantics with hierarchical notations. This combination allows us to model and analyze behaviors of MASs under several levels of abstraction. We have presented a tool that facilitates the specification process by permitting the graphical notations while specifying behaviors and requirements.

One possible area of future research is to reason about the behavior of MASs under uncertainty. Decision theories have proposed a number of decision rules for decision-making under uncertainty [Bermúdez, 2009]. One should integrate such types of rules into our approach.

Reasoning about knowledge [Fagin, 2003] has always been a core concern in AI and MASs. It is well known that knowledge is a key concept to model intelligent and rational activities. The usual approach to reason about knowledge assumes time to be discrete. Thus, another topic worth investigating would be extending model checking to reason not only about temporal properties but also about epistemic properties of agents.

A

Appendix

This Appendix lists the complete CLP for the train gate example demonstrated in Chapter 4 and 8.

The concurrent CLP

```
:- lib(ic).
:- lib(ic_symbolic).
:-lib(scattered).
:- local domain(events(app,in,exit,lower,raise,to_close,to_open)).
%%% train
train(far,[Y0],[Y],T0,T):-
      Y $>= Y0-50*(T-T0),
      Y $>=500, T $>=T0.

train(near,[Y0],[Y],T0,T):-
      Y $=(exp(-(T-T0)/25))*(Y0+750)-750,
      Y $>=0, T $>= T0.

train(past,[Y0],[Y],T0,T):-
      Y $= (exp((T- T0)/5) )*(Y0+150)-150,
      Y $=<100, T $>= T0.

gate(open,[G0],[G],T0,T):-
      G $=G0+0*(T-T0),
      T $>=T0.

%%%gate
gate(close,[G0],[G],T0,T):-
      G $=G0+0*(T-T0),
      T $>=T0.

gate(up,[G0],[G],T0,T):-
```

```
      G $=G0+20*(T-T0),
      T $>=T0, G $=<90.

gate(down,[G0],[G],T0,T):-
      G $= G0-20*(T-T0),
      T $>=T0, G $>=0.

%%controller
controller(idle,[Z0],[Z],T0,T):-
      Z $=Z0+0*(T-T0),
      T $>=T0.

controller(to_lower,[Z0],[Z],T0,T):-
    Z $=Z0+(T-T0),
    T $>=T0, Z $=<5.

controller(to_raise,[Z0],[Z],T0,T):-
    Z $=Z0+(T-T0),
    T $>=T0, Z $=<5.


evolve(Automaton,(State,Value1),(State,Value2),T0,T,Tn,Event):-
    continuous(Automaton,(State,Value1),(State,Value2),T0,T,Tn,Event),
    Tn $>=0.

evolve(Automaton,(State,Value1),(Nextstate,Value2),T0,T,Tn,Event):-
    discrete(Automaton,(State,Value1),(Nextstate,Value2),T0,T,Tn,Event).

discrete(train,(far,[X0]),(near,[XX0]),T0,T,Tn,Event):-
    Event &::events,Event &=app,
    train(far,[X0],[XX0],T0,Tn),
    XX0 $=500.

discrete(train,(near,[X0]),(past,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=in,
   train(near,[X0],[XX0],T0,Tn),
   XX0 $=0.

discrete(train,(past,[X0]),(far,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=exit,
   train(past,[X0],[100],T0,Tn),
   XX0 $=2000.

discrete(controller,(idle,[X0]),(to_lower,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=app,
   XX0 $=0,
   controller(idle,[X0],[X],T0,Tn).

discrete(controller,(idle,[X0]),(to_raise,[XX0]),T0,T,Tn,Event):-
```

```
   Event &::events,Event &=exit,
   XX0 $=0,
   controller(idle,[X0],[X],T0,Tn).

discrete(controller,(to_lower,[X0]), (idle,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=lower,
   XX0 $=X0,
  controller(to_lower,X0,X,T0,Tn).

discrete(controller,(to_lower,[X0]),(to_raise,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=exit,
   XX0 $=0,
   controller(to_lower,[X0],[X],T0,Tn) .

discrete(controller,(to_raise,[X0]),(idle,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=raise,
   XX0 $=X0,
   controller(to_raise,[X0],[X],T0,Tn).

discrete(controller,(to_raise,[X0]),(to_lower,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=app,
   XX0 $=0,
   controller(to_raise,[X0],[X],T0,Tn).

discrete(gate,(open,[X0]),(down,[XX0],T0,T,Tn,Event):-
   Event &::events,Event &=lower,
   XX0 $=X0,
   gate(open,X0,X,T0,Tn).

discrete(gate,(close,[X0]),(up,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=raise,
   XX0 $=X0,
   gate(close,[X0],[X],T0,Tn).

discrete(gate,(down,[X0]),(close,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=to_close,
   XX0 $=0,
   gate(down,[X0],[0],T0,Tn),

discrete(gate,(down,[X0]),(up,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=raise,
   XX0 $=X0,
   gate(down,[X0],[X],T0,Tn).

discrete(gate,(up,[X0]),(open,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=to_open,
   XX0 $=90,
   gate(up,X0,90,T0,Tn).
```

```
discrete(gate,(up,[X0]),(down,[XX0]),T0,T,Tn,Event):-
   Event &::events,Event &=lower,
   XX0 $=X0,
   gate(up,[X0],[X],T0,Tn) .

continuous(train,(far,[X0]),(far,[X0]),T0,T,Event):-
   Event &::events, Event &\=app, Event &\= in,Event &\=exit,
   train(far,[X0],[X],T0,T), \+ (X $=500).

continuous(train,(near,[X0]),(near,[X0]),T0,T,Event):-
   Event &::events, Event &\=app, Event &\= in,Event &\=exit,
   train(near,[X0],[X],T0,T), \+ (X $=0).

continuous(train,(past,[X0]),(past,[X0]),T0,T,Event):-
   Event &::events, Event &\=app, Event &\= in,Event &\=exit,
   train(past,[X0],[X],T0,T), \+ (X $=100).


continuous(controller,(idle,[X0]),(idle,[X0]),T0,T,Event):-
  Event &::events,Event &\=app, Event &\=exit,Event &\=raise,
  Event &\=lower, controller(idle,[X0],[X],T0,T).

continuous(controller,(to_lower,[X0]),(to_lower,[X0]),T0,T,Event):-
   Event &::events,Event &\=app,Event &\=exit,Event &\=raise,
   Event &\=lower, controller(to_lower,[X0],[X],T0,T), \+ (X $=5).

continuous(controller,(to_raise,[X0]),(to_raise,[X0]),T0,T,Event):-
   Event &::events,Event &\=app, Event &\=exit,Event &\=raise,
   Event &\=lower,
   controller(to_raise,[X0],[X],T0,T), \+ (X $=5).

continuous(gate,(open,[X0]),(open,[X0]),T0,T,Event):-
   Event &::events,Event &\=lower, Event &\=to_open,
   Event &\=to_close,Event &\=raise,
   gate(open,[X0],[X],T0,T).

continuous(gate,(down,[X0]),(down,[X0]),T0,T,Event):-
  Event &::events,Event &\=lower,
  Event &\=to_open,
  Event &\=to_close,Event &\=raise,
  gate(down,[X0],[X],T0,T),\+ (X $=0).

continuous(gate,(up,[X0]),(up,[X0]),T0,T,Event):-
  Event &::events,Event &\=lower,
  Event &\=to_open,
  Event &\=to_close,Event &\=raise,
  gate(up,[X0],[X],T0,T),\+ (X $=90).

continuous(gate,(close,[X0]),(close,[X0]),T0,T,Event):-
```

```
  Event &::events,Event &\=lower,
  Event &\=to_open,
  Event &\=to_close,Event &\=raise,
  gate(close,[X0],[X],T0,T).

drive((S1,[X0]),(S2,[G0]),(S3,[Z0]), Starttime,[(S1,S2,S3,Time,Event,X)|L],B):-
  train(S1,[X0],[X],Starttime,Tx),
  gate(S2,[G0],[G],Starttime,Tg),
  controller(S3,[Z0],[Z],Starttime,Tz),
  Tx $=Tg, Tx$= Tz , Time $=Tx,

  evolve(train,(S1,[X0]),(NextS1,[XX0]),Starttime,Tx,Tx1,Event),
  evolve(gate,(S2,[G0]),(NextS2,[GG0]),Starttime,Tx,Tg1,Event),
  evolve(controller,(S3,[Z0]),(NextS3,[ZZ0]),Starttime,Tx,Tz1,Event),
  Tx1 $=Tg1, Tx1 $=Tz1, Tnew $=Tx1,
  \+ member((S1,S2,S3,_,Event,X),B),
  A=[(S1,S2,S3,Time,Event,X)|B],
  drive((NextS1,[XX0]),(NextS2,[GG0]),(NextS3,[ZZ0]),Tnew,L,A) .
  drive(_,_,_,_,[],_):- !.

Reachable((L1,[X0]),(L2,[G0]),(L3,[Z0]),Reached):- drive((L1,[X0]),(L2,[G0]),(L3,[Z0]),0,
```

Hierarchical CLP

```
:- lib(ic).
greater(T2,T1) :- T2 $> T1.

%%% system
start(system).

init(T,system,[[Event,Alpha]],[train,gate,controller],_) :-
Event = none.
flow(T1,T2,system,[[Event,Alpha]],[[Event,Alpha]]).

%%% train
init(T,train,[[X]|_],[far],system) :-
X $= 2000.
flow(T1,T2,train,_,_).

init(T,far,[[]|_],[],train).
flow(T1,T2,far,[[],[X1]|_],[[],[X2]|_]) :-
X2 $>= 1000,
X2 $>= X1-50*(T2-T1),
X2 $=< X1-30*(T2-T1).
trans(T,far,[[],[X],[Event1,Alpha]],far,[[],[X],[Event2,Alpha]]) :-
Event2 = lower ; Event2 = raise.
```

```
trans(T,far,[[],[X],[Event1,Alpha]],near,[[],[X],[Event2,Alpha]]) :-
Event2 = app,
X $= 1000.

init(T,near,[[]|_],[],train).
flow(T1,T2,near,[[],[X1]|_],[[],[X2]|_]) :-
X2 $>= 0,
X2 $>= X1-50*(T2-T1),
X2 $=< X1-30*(T2-T1).
trans(T,near,[[],[X],[Event1,Alpha]],near,[[],[X],[Event2,Alpha]]) :-
Event2 = lower ; Event2 = raise.
trans(T,near,[[],[X],[Event1,Alpha]],past,[[],[X],[Event2,Alpha]]) :-
Event2 = in,
X $= 0.

init(T,past,[[]|_],[],train).
flow(T1,T2,past,[[],[X1]|_],[[],[X2]|_]) :-
X2 $=< 100,
X2 $>= X1+30*(T2-T1),
X2 $=< X1+50*(T2-T1).
trans(T,past,[[],[X],[Event1,Alpha]],past,[[],[X],[Event2,Alpha]]) :-
Event2 = lower ; Event2 = raise.
trans(T,past,[[],[X1],[Event1,Alpha]],far,[[],[X2],[Event2,Alpha]]) :-
Event2 = exit,
X1 $= 100,
X2 $= 2000.

%%% gate
init(T,gate,[[G]|_],[open],system) :-
G $= 90.
flow(T1,T2,gate,_,_).

init(T,open,[[]|_],[],gate).
flow(T1,T2,open,[[],[G1]|_],[[],[G2]|_]) :-
G1 $= 90,
G2 $= G1+0*(T2-T1).
trans(T,open,[[],[G],[Event1,Alpha]],open,[[],[G],[Event2,Alpha]]) :-
Event2 = app ; Event2 = in ; Event2 = raise.
trans(T,open,[[],[G],[Event1,Alpha]],down,[[],[G],[Event2,Alpha]]) :-
Event2 = lower.

init(T,down,[[]|_],[],gate).
flow(T1,T2,down,[[],[G1]|_],[[],[G2]|_]) :-
G1 $>= 0,
G2 $= G1-9*(T2-T1).
trans(T,down,[[],[G],[Event1,Alpha]],down,[[],[G],[Event2,Alpha]]) :-
Event2 = app ; Event2 = in ; Event2 = lower.
trans(T,down,[[],[G1],[Event1,Alpha]],closed,[[],[G2],[Event2,Alpha]]) :-
G2 $= 0.
```

```
trans(T,down,[[],[G1],[Event1,Alpha]],up,[[],[G1],[Event2,Alpha]]) :-
Event2 = raise.

init(T,closed,[[]|_],[],gate).
flow(T1,T2,closed,[[],[G1]|_],[[],[G2]|_]) :-
G1 $= 90,
G2 $= G1+0*(T2-T1).
trans(T,closed,[[],[G],[Event1,Alpha]],closed,[[],[G],[Event2,Alpha]]) :-
Event2 = app ; Event2 = in ; Event2 = lower.
trans(T,closed,[[],[G],[Event1,Alpha]],up,[[],[G],[Event2,Alpha]]) :-
Event2 = raise.

init(T,up,[[]|_],[],gate).
flow(T1,T2,up,[[],[G1]|_],[[],[G2]|_]) :-
  G1 $=< 90,
G2 $= G1+9*(T2-T1).
trans(T,up,[[],[G],[Event1,Alpha]],up,[[],[G],[Event2,Alpha]]) :-
Event2 = app ; Event2 = in ; Event2 = raise.
trans(T,up,[[],[G1],[Event1,Alpha]],open,[[],[G1],[Event2,Alpha]]) :-
G2 $= 90.
trans(T,up,[[],[G1],[Event1,Alpha]],down,[[],[G1],[Event2,Alpha]]) :-
Event2 = lower.


%%% controller
init(T,controller,[[D]|_],[idle],system) :-
D $= 0.
flow(T1,T2,controller,[[D1]|_],[[D2]|_]) :-
D2 $>= 0.

init(T,idle,[[]|_],[],controller).
flow(T1,T2,idle,[[],[D1]|_],[[],[D2]|_]) :-
D2 $= D1+0*(T2-T1).
trans(T,idle,[[],[D],[Event1,Alpha]],idle,[[],[D],[Event2,Alpha]]) :-
Event2 = in.
trans(T,idle,[[],[D1],[Event1,Alpha]],lower,[[],[D2],[Event2,Alpha]]) :-
Event2 = app,
D2 $= 0.
trans(T,idle,[[],[D1],[Event1,Alpha]],raise,[[],[D2],[Event2,Alpha]]) :-
Event2 = exit,
D2 $= 0.

init(T,lower,[[]|_],[],controller).
flow(T1,T2,lower,[[],[D1],[Event,Alpha]],[[],[D2],[Event,Alpha]]) :-
D2 $=< Alpha,
D2 $= D1+1*(T2-T1).
trans(T,lower,[[],[D],[Event1,Alpha]],lower,[[],[D],[Event2,Alpha]]) :-
Event2 = app ; Event2 = in.
trans(T,lower,[[],[D],[Event1,Alpha]],idle,[[],[D],[Event2,Alpha]]) :-
```

```
Event2 = lower.
trans(T,lower,[[],[D1],[Event1,Alpha]],raise,[[],[D2],[Event2,Alpha]]) :-
Event2 = exit,
D2 $= 0.

init(T,raise,[[]|_],[],controller).
flow(T1,T2,raise,[[],[D1],[Event,Alpha]],[[],[D2],[Event,Alpha]]) :-
D2 $=< Alpha,
D2 $= D1+1*(T2-T1).
trans(T,raise,[[],[D],[Event1,Alpha]],raise,[[],[D],[Event2,Alpha]]) :-
Event2 = exit ; Event2 = in.
trans(T,raise,[[],[D],[Event1,Alpha]],idle,[[],[D],[Event2,Alpha]]) :-
Event2 = raise.
trans(T,raise,[[],[D1],[Event1,Alpha]],lower,[[],[D2],[Event2,Alpha]]) :-
Event2 = app,
D2 $= 0.
```

References

Abrial, J.-R. (2009). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

Alur, R., Courcoubetis, C., and Dill, D. (1993). Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34.

Alur, R., Courcoubetis, C., Henzinger, T. A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1994). The algorithmic analysis of hybrid systems. In *ICAOS: International Conference on Analysis and Optimization of Systems – Discrete-Event Systems*, Lecture Notes in Control and Information Sciences 1994, pages 331–351. Springer, Berlin, Heidelberg, New York.

Alur, R., Dang, T., Esposito, J. M., Fierro, R. B., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G. J., and Sokolsky, O. (2001). Hierarchical hybrid modeling of embedded systems. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 14–31, London, UK. Springer-Verlag.

Alur, R. and Dill, D. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235.

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

Alur, R., Grosu, R., Hur, Y., Kumar, V., and Lee, I. (2000). Modular specification of hybrid systems in charon. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 6–19, London, UK. Springer-Verlag.

Alur, R. and Henzinger, T. (1992). Logics and models of real time: A survey. *Real Time: Theory in Practice, Lecture Notes in Computer Science*, 600:74–106.

Alur, R. and Henzinger, T. (1994). A really temporal logic. *Journal of the ACM (JACM)*, 41(1):203.

Alur, R. and Henzinger, T. A. (1993). Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77.

Alur, R., Henzinger, T. A., and Ho, P.-H. (1996b). Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201.

Alur, R. and Kurshan, R. P. (1996). Timing analysis in COSPAN. *Lecture Notes in Computer Science*, 1066:220–231.

Alur, R. and Yannakakis, M. (1998). Model checking of hierarchical state machines. *SIGSOFT Softw. Eng. Notes*, 23(6):175–188.

Apt, K. R. and Wallace, M. (2007). *Constraint Logic Programming Using Eclipse*. Cambridge University Press, Cambridge, UK.

Arai, T. and Stolzenburg, F. (2002). Multiagent systems specification by uml statecharts aiming at intelligent manufacturing. pages 11–18.

Balarin, F. and Sangiovanni-Vincentelli, A. L. (1994). Iterative algorithms for formal verification of embedded real-time systems. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 450–457, Los Alamitos, CA, USA. IEEE Computer Society Press.

Banda, G. and Gallagher, J. P. (2008). Analysis of linear hybrid systems in CLP. In Hanus, M., editor, *Pre-Proceedings of LOPSTR 2008 – 18th International Symposium on Logic-Based Program Synthesis and Transformation*, pages 58–72. Technical University of Valencia, Spain.

Bauer, B., Müller, J. P., and Odell, J. (2001). Agent uml: a formalism for specifying multiagent software systems. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 91–103, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on Uppaal. In Bernardo, M. and Corradini, F., editors, *Proceedings of 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems – Formal Methods for the Design of Real-Time Systems (SFM-RT)*, LNCS 3185, pages 200–236. Springer, Berlin, Heidelberg, New York.

Bellini, P., Mattolini, R., and Nesi, P. (2000). Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42.

Ben-Ari, M., Pnueli, A., and Manna, Z. (1983). The temporal logic of branching time. *Acta Informatica*, 20(3):207–226.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1996). Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In Desel, J., Reisig, W., and Rozenberg, G., editors, *Lectures on Concurrency and Petri Nets*, LNCS 3098, pages 87–124. Springer, Berlin, Heidelberg, New York.

Bermúdez, J. L. (2009). *Decision theory and rationality*. Oxford University press.

Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). Symbolic model checking without BDDs. In *Proceedings of 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS 1579, pages 193–207. Springer, Berlin, Heidelberg, New York.

Bouajjani, A., Echahed, R., and Sifakis, J. (1993). On model checking for real-time properties with durations. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer*, pages 147–159. IEEE Computer Society.

Bouajjani, A., Tripakis, S., and Yovine, S. (1997). On-the-fly symbolic model checking for real-time systems. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 232–243.

Burgess, J. P. (1984). Basic tense logic. *Handbook of philosophical logic*, 2:89–133.

Cabac, L. and Moldt, D. (2004). Formal semantics for auml agent interaction protocol diagrams. In Odell, J., Giorgini, P., and Müller, J. P., editors, *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3382 of *Lecture Notes in Computer Science*, pages 47–61. Springer.

Cansell, D., Abrial, J., et al. (2004). B4free. *A set of tools for B development. Available from: http://www.b4free.com*.

Cassez, F. and Roux, O. H. (2006). Structural translation from time petri nets to timed automata. *Journal of Systems and Software*, 79(10):1456 – 1468. Architecting Dependable Systems.

Cataldo, A., Hylands, C., Lee, E., Liu, J., Liu, X., Neuendorffer, S., and Zheng, H. (2003). Hyvisual: A hybrid system visual modeler. Technical report, UCB/ERL M03/1, UC Berkely. available at http://ptolemy.eecs.berkeley.edu/hyvisual.

Celaya, J., Desrochers, A., and Graves, R. (2009). Modeling and analysis of multi-agent systems using petri nets. *Journal of Computers*, 4(10):981– 996.

Chainbi, W. (2004). Multi-agent systems: A petri net with objects based approach. *Intelligent Agent Technology, IEEE / WIC / ACM International Conference on*, 0:429–432.

Chaochen, Z., Hoare, C. A. R., and Ravn, A. P. (1991). A calculus of durations. *Information Processing Letters*, 40(5):269–276.

Ciarlini, A. and Frühwirth, T. (2000). Automatic derivation of meaningful experiments for hybrid systems. *Proceeding of ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark. Springer.

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model checking*. Springer.

Colmeraue, A. (1984). Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99.

Colmerauer, A. (1990). An introduction to prolog iii. *Commun. ACM*, 33(7):69–90.

da Silva, V., Choren, R., and de Lucena, C. (2004). A UML based approach for modeling and implementing multi-agent systems. pages 914–921.

David, R. (1997). Modeling of hybrid systems using continuous and hybrid petri nets. In *PNPM '97: Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 47, Washington, DC, USA. IEEE Computer Society.

de Weerdt, M., ter Mors, A., and Witteveen, C. (2005). Multi-agent planning: An introduction to planning and coordination. In *Handouts of the European Agent Summer School*, pages 1–32.

Decker, K. S. and Lesser, V. R. (1998). Designing a family of coordination algorithms. pages 450–457.

Del Bianco, V., Lavazza, L., and Mauri, M. (2002). Model checking uml specifications of real time software. page 203.

Delzanno, G. and Podelski, A. (1999). Model checking in CLP. In *Proceedings of 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS 1579, pages 223–239. Springer, Berlin, Heidelberg, New York.

Delzanno, G. and Podelski, A. (2001). Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):250–270.

Deshpande, A., Göllü, A., and Varaiya, P. (1997). Shift: A formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems IV*, pages 113–133, London, UK. Springer-Verlag.

DesJardins, M. E., Durfee, E. H., Ortiz, C. L., Jr., and Wolverton, M. J. (2000). Survey of research in distributed, continual planning. *AI MAG*, 20(4):13–22.

Dill, D. L. and Wong-Toi, H. (1995). Verification of real-time systems by successive over and under approximation. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 409–422, London, UK. Springer-Verlag.

Dincbas, M., Hentenryck, P. V., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. (1988). The constraint logic programming language chip. In *Proceedings of the international conference on fifth generation computer systems*, pages 693–702.

Egerstedt, M. (2000). Behavior Based Robotics Using Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 103–116.

Eker, S., Meseguer, J., and Sridharanarayanan, A. (2002). The maude ltl model checker. *Electr. Notes Theor. Comput. Sci.*, 71.

El Fallah-Seghrouchni, A., Degirmenciyan-Cartault, I., and Marc, F. (2003). Framework for Multi-agent Planning Based on Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 226–235.

Emerson, E. A., Mok, A. K., Sistla, A. P., and Srinivasan, J. (1992). Quantitative temporal reasoning. *Real-Time Syst.*, 4(4):331–352.

Fagin, R. (2003). *Reasoning about knowledge*. The MIT Press.

FIPA (2002). Contract Net Interaction Protocol Specification. Available from: http://www.fipa.org/specs/fipa00029/SC00029H.pdf.

Fox, M. S., Barbuceanu, M., and RuneTeigen. (2000). Agent-oriented supply-chain management. *International Journal of Flexible Manufacturing Systems*, 12(2):165–188.

Fränzle, M. and Herde, C. (2007). HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198.

Frehse, G. (2005). PHAVer: Algorithmic verification of hybrid systems past HyTech. In Morari, M. and Thiele, L., editors, *Hybrid Systems: Computation and Control, 8th International Workshop, Proceedings*, LNCS 3414, pages 258–273. Springer, Berlin, Heidelberg, New York.

Furbach, U., Murray, J., Schmidsberger, F., and Stolzenburg, F. (2008). Hybrid multiagent systems with timed synchronization – specification and model checking. In Dastani, M., El Fallah Seghrouchni, A., Ricci, A., and Winikoff, M., editors, *Post-Proceedings of 5th International Workshop on Programming Multi-Agent Systems at 6th International Joint Conference on Autonomous Agents & Multi-Agent Systems*, LNAI 4908, pages 205–220. Springer.

Gehrke, J. D., Behrens, C., Jedermann, R., and Morales Kluge, E. (2006). The intelligent container - toward autonomous logistic processes. In *KI 2006 Demo Presentations*, pages 15–18. Universität Bremen. Published on CD-ROM.

Ghomri, L. and Alla, H. (2007). Modeling and analysis using hybrid petri nets. *Nonlinear Analysis: Hybrid Systems*, 1(2):141 – 153. Nonlinear Hybrid Control Systems.

Giunchiglia, F. and Traverso, P. (2000). Planning as Model Checking. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 1–20.

Gnesi, S., Latella, D., and Massink, M. (1999). Model checking uml statechart diagrams using jack. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 46–55, Washington, DC, USA. IEEE Computer Society.

Graf, S., Ober, I., and Ober, I. (2006). A real-time profile for UML. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(2):113–127.

Gulwani, S. and Tiwari, A. (2008). Constraint-based approach for analysis of hybrid systems. In Raskin, J.-F. and Thiagarajan, P. S., editors, *Proceedings of 20th International Conference on Computer Aided Verification (CAV 2008)*, LNCS 5123, pages 190–203, Princeton, NJ. Springer, Berlin, Heidelberg, New York.

Gupta, G. and Pontelli, E. (1997). A constraint-based approach for specification and verification of real-time systems. *Proceedings of IEEE Real-time Symposium*, pages 230–239.

Halbwachs, N., Proy, Y., and Raymond, P. (1994). Verification of linear hybrid systems by means of convex approximations. In *Static Analysis – Proceedings of 1st International Static Analysis Symposium (SAS'94)*, LNCS 864, pages 223–223, Namur, Belgium. Springer, Berlin, Heidelberg, New York.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(-):231–274.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, a. (1988). Statemate: a working environment for the development of complex reactive systems. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 396–406, Los Alamitos, CA, USA. IEEE Computer Society Press.

Harel, D. and Pnueli, A. (1985). On the development of reactive systems. pages 477–498.

Harel, E., Lichtenstein, O., and Pnueli, A. (1990). Explicit clock temporal logic. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 402–413. IEEE Computer Society.

Henzinger, T. (1996). The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, NJ. IEEE Computer Society Press.

Henzinger, T., Ho, P.-H., and Wong-Toi, H. (1995). A user guide to HyTech. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 41–71. Springer, Berlin, Heidelberg, New York.

Henzinger, T., Horowitz, B., Majumdar, R., and Wong-Toi, H. (2000). Beyond HYTECH: Hybrid Systems Analysis Using Interval Numerical Methods. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 130–144.

Henzinger, T., Kopke, P., Puri, A., and Varaiya, P. (1998a). What's Decidable about Hybrid Automata? *Journal of Computer and System Sciences*, 57(1):94–124.

Henzinger, T. A. (2000). Masaccio: A formal model for embedded components. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 549–563, London, UK. Springer-Verlag.

Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1997). Hytech: A model checker for hybrid systems. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463, London, UK. Springer-Verlag.

Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1998b). Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554.

Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244.

Hickey, T. J. and Wittenberg, D. K. (2004a). Rigorous modeling of hybrid systems using interval arithmetic constraints. In Alur, R. and Pappas, G. J., editors, *Proceedings of 7th International Workshop on Hybrid Systems: Computation and Control (HSCC 2004)*, LNCS 2993, pages 402–416, Philadelphia, PA, USA. Springer, Berlin Heidelberg, New York.

Hickey, T. J. and Wittenberg, D. K. (2004b). Using analytic CLP to model and analyze hybrid systems. In *Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295.

Hutzler, G., Klaudel, H., and Wang, D. Y. (2005). Towards timed automata and multi-agent systems. In *Formal Approaches to Agent-Based Systems, Third InternationalWorkshop, FAABS 2004, Greenbelt, MD, USA, April 26-27, 2004, Revised Selected Papers*, volume 3228 of *Lecture Notes in Computer Science*, pages 161–172. Springer.

Jaffar, J. and Lassez, J. (1987). Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM New York, NY, USA.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. (1992). The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395.

Jaffar, J., Santosa, A., and Voicu, R. (2004). A clp proof method for timed automata. *Real-Time Systems Symposium, IEEE International*, 0:175–186.

Jahanian, F. and Mok, A. K. (1986). Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(9):890–904.

Jahanian, F. and Mok, A. K. (1994). Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 20(12):933–947.

James, P. (1977). Petri nets. *ACM Computing Surveys*, 9(3):223–252.

Jemni Ben Ayed, L. and Siala, F. (2008). Specification and verification of multi-agent systems interaction protocols using a combination of auml and event b. pages 102–107.

Kesten, Y. and Pnueli, A. (1991). Timed and hybrid statecharts and their textual representation. In *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620, London, UK. Springer-Verlag.

Knapp, A., Merz, S., and Rauh, C. (2002). Model checking-timed UML state machines and collaborations. *Lecture notes in computer science*, pages 395–416.

Kopke, Jr., P. W. (1996). *The theory of rectangular hybrid automata*. PhD thesis, Ithaca, NY, USA. Adviser-Henzinger, Thomas A.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152.

Lilius, J. and Porres, I. (1999). Formalising UML state machines for model checking. page 430.

Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems: Specification*. Springer.

McMillan, K. (1993). *Symbolic model checking: an approach to the state-explosion problem* . Kluwer Academic.

Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. J. (1998). Implementing statecharts in promela/spin. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90, Washington, DC, USA. IEEE Computer Society.

Mohammed, A. and Furbach, U. (2008a). Modeling multi-agent logistic process system using hybrid automata. In Ultes-Nitsche, U., Moldt, D., and Augusto, J. C., editors, *In Proceedings of the 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2008*, pages 141–149, Barcelona, Spain. INSTICC PRESS. Held in conjunction with 10th International Conference on Enterprise Information Systems (ICEIS 2008).

Mohammed, A. and Furbach, U. (2008b). Using CLP to model hybrid systems. In *Proceedings of Annual ERCIM Workshop on Constraint Solving Programming (CSCLP2008)*, Rome, Italy. Published online http://pst.istc.cnr.it/CSCLP08.

Mohammed, A. and Furbach, U. (2009a). From reactive to deliberative multi-agent planning. In Ultes-Nitsche, U., Moldt, D., and Augusto, J. C., editors, *In Proceedings of the 7th International Workshop on Modeling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2009*, pages 67–75, Milan, Italy. INSTICC PRESS. Held in conjunction with 11th International Conference on Enterprise Information Systems (ICEIS 2009).

Mohammed, A. and Furbach, U. (2009b). Multi-agent systems: Modeling and verification using hybrid automata. In *Proceedings of the 7th International Workshop on Programming Multi-Agent Systems (ProMAS 2009), May 10-15, 2009, Budapest, Hungary*. Revised Version, will appear as the workshop Post-proceedings LNCS, Springer.

Mohammed, A. and Furbach, U. (2010a). Extending ctl to specify quantitative temporal requirements. In Sopena, J. G. and l. Capel-Tunon, M., editors, *In Proceedings of the 8th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2010*, pages 70–79, Funchal, Madeira, Portugal. INSTICC PRESS. Held in conjunction with 11th International Conference on Enterprise Information Systems (ICEIS 2010).

Mohammed, A. and Furbach, U. (2010b). Multi-agent systems: Modeling and verification using hybrid automata. In Lars Braubach, J.-P. B. and Thangarajah, J., editors, *Programming*

Multi-Agent Systems:7th International Workshop,ProMAS2009, Budapest, Hungary, May 2009, Revised Selected Papers, LNAI 5919, pages 49–66. Springer, Berlin, Heidelberg.

Mohammed, A., Furbach, U., and Stolzenburg, F. (2010). Multi-robot systems: Modeling, specification, and model checking. In Papic, V., editor, *Robot Soccer*, chapter 11, pages 241–265. IN-TECH.

Mohammed, A. and Schwarz, C. (2009). Hieromate: A graphical tool for specification and verification of hierarchical hybrid automata. In B. Mertsching, M. H. and Aziz, Z., editors, *KI 2009: Advances in Artificial Intelligence, Proceedings of the 32nd German Conference on Artificial Intelligence*, LNAI 5803, pages 695–702. Springer.

Mohammed, A. and Stolzenburg, F. (2008). Implementing hierarchical hybrid automata using constraint logic programming. In Schwarz, S., editor, *Proceedings of 22nd Workshop on (Constraint) Logic Programming*, pages 60–71, Dresden. University Halle Wittenberg, Institute of Computer Science. Technical Report 2008/08.

Mokhati, F., Boudiaf, N., Badri, M., and Badri, L. (2007). Translating auml diagrams into maude specifications: A formal verification of agents interaction protocols. *Journal of Object Technology*, 6(4).

Mota, E., Clarke, E. M., Groce, A., Oliveira, W., Falcão, M., and Kanda, J. (2004). Veriagent: an approach to integrating uml and formal verification tools. *Electr. Notes Theor. Comput. Sci.*, 95:111–129.

Murray, J. (2004). Specifying agents with UML statecharts and StatEdit. 3020:145–156.

Murray, J., Obst, O., and Stolzenburg, F. (2002). RoboLog Koblenz 2001. In Birk, A., Coradeschi, S., and Tadokoro, S., editors, *RoboCup 2001: Robot Soccer World Cup V*, LNAI 2377, pages 526–530. Springer, Berlin, Heidelberg, New York. Team description.

Mller, O., David, A., and Yi, W. (2003). Verification of uml statechart with real-time extensions. pages 218–232.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nicollin, X., Sifakis, J., and Yovine, S. (1992). From atp to timed graphs and hybrid systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 549–572, London, UK. Springer-Verlag.

Nilsson, U. and Lübcke, J. (2000). Constraint logic programming for local and symbolic model-checking. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 384–398, London, UK. Springer-Verlag.

Olderog, E.-R. and Dierks, H. (2008). *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press.

Ostroff, J. and Wonham, W. (1990). A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, 35(4):386–397.

Pereira, S. L. and Barros, L. N. (2008). A logic-based agent that plans for extended reachability goals. *Autonomous Agents and Multi-Agent Systems*, 16(3):327–344.

Pistore, M. and Traverso, P. (2001). Planning as model checking for extended goals in nondeterministic domains. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, pages 479–484, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Pnueli, A. (1977). The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57.

Pnueli, A. and Harel, E. (1988). Applications of temporal logic to the specification of real-time systems. In *Systems, Proceedings of a Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98, London, UK. Springer-Verlag.

Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Dong, Y., Du, X., Roychoudhury, A., and Venkatakrishnan, V. N. (2000). Xmc: A logic-programming-based verification toolset. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 576–580, London, UK. Springer-Verlag.

Rossi, F. (2000). Constraint (logic) programming: a survey on research and applications. *Lecture Notes in Computer Science*, 1865:40–74.

Roy, V. and Simone, R. d. (1991). Auto/autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK. Springer-Verlag.

Ruh, F. (2007). *A translator for cooperative strategies of mobile agents for four-legged robots*. Master thesis, Dept. of Automation and Computer Sciences, Hochschule Harz, Hochschule Harz.

Ruh, F. and Stolzenburg, F. (2008). Translating cooperative strategies for robot behavior. In Nalepa, G. J. and Baumeister, J., editors, *Proceedings of 4th Workshop on Knowledge Engineering and Software Engineering at 31st German Conference on Artificial Intelligence*, pages 85–96, Kaiserslautern. CEUR Workshop Proceedings 425.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Russell, S., Norvig, P., Canny, J., Malik, J., and Edwards, D. (2003). *Artificial intelligence: a modern approach*. Prentice Hall Englewood Cliffs, NJ.

Sahbani, A. and Pascal, J.-C. (2000). Simulation of hyibrd systems using stateflow. In Landeghem, R. V., editor, *14th European Simulation Multiconference - Simulation and Modelling: Enablers for a Better Quality of Life*. SCS Europe.

Scholz-Reiter, B., Windt, K., and Freitag, M. (2004). Autonomous Logistic Processes-New Demands and First Approaches. In *Proceedings of the 37th CIRP International Seminar on Manufacturing Systems, Budapest, Hungaria*, pages 357–362.

Schwarz, C., Mohammed, A., and Stolzenburg, F. (2010). A tool environment for specifying and verifying multi-agent systems. In Filipe, J., Fred, A., and Sharp, B., editors, *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence*, volume 2, pages 323–326. INSTICC Press.

Tadokoro, S. et al. (2000). The RoboCup-Rescue project: A robotic approach to the disaster mitigation problem. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA 2000)*, pages 4089–4104.

UML (2009). *OMG Unified Modeling Language (OMG UML): Infrastructure; Superstructure*. Object Management Group, Inc.

Urbina, L. (1996). Analysis of hybrid systems in CLP(R). In *Proceedings of 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, LNAI 1118, pages 451–467.

Van Benthem, J. and ter Meulen, A., editors (1997). *Handbook of Logic and language*. Elsevier.

Wang, J. (1998). *Timed Petri nets: Theory and application*. Kluwer Academic Publishers.

Wen, W. and Mizoguchi, F. (1999). Analysis and verification of multi-agent interaction protocols. In *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*, page 252, Washington, DC, USA. IEEE Computer Society.

Wielemaker, J. (2008). *SWI-Prolog 5.6 – Reference Manual*. University of Amsterdam, Amsterdam, The Netherlands. Updated for version 5.6.59.

Wood, M. and DeLoach, S. (2001). An Overview of the Multiagent Systems Engineering Methodology. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 207–222.

Wooldridge., M. (2002). An introduction to multiagent systems. *John Willey & Sons, New York*.

Wooldridge, M. and Jennings, N. (1995). Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152.

Yovine, S. (1997). Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133.

Curriculum Vitae

Personal Data

- Ammar Mohammed Ammar
  1.11.1977, Cairo, Egypt.

Education

- M.Sc., Computer Science, March 2005, Department of Computer and Information Science, ISSR, University of Cairo, Cairo, Egypt. Thesis Title: A Multi-Agent System for Solving a Scheduling Problem.

- May 2002: Preliminary Master courses leading to M.Sc registration , May 2002, Department of Computer and Information Science, ISSR, University of Cairo, Egypt.

- B.Sc., Computer Science, May 1999, Faculty of Science, University of Cairo, Egypt Very Good with honor class.

Professional Career

- October 2006–today: Joining AI Research Group, Department of Computer Science, University of Koblenz-landau.

- March 2005–today: Assistant Lecturer, Department of Computer and Information Sceince,Institute of Statistical Studies and Research, University of Cairo, Egypt.

- April 2000–March 2005: Teaching Assistant,Department of Computer and Information Sceince,Institute of Statistical Studies and Research, University of Cairo, Egypt.