A Model for Hierarchical Open Real-Time Systems

by

Md Tawhid Bin Waez

A thesis submitted to the

School of Computing

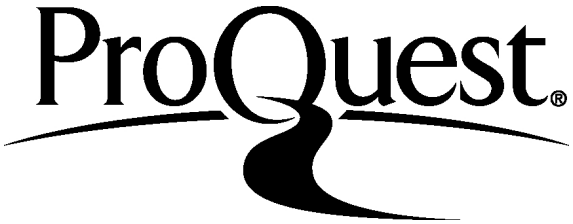
in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

September 2015



Abstract

Introducing automated formal methods for large industrial real-time systems is an important research challenge. We propose *timed process automata* for modeling and analysis of time-critical systems which can be open, hierarchical, and dynamic. The model offers two essential features for large industrial systems: (i) compositional modeling with reusable designs for different contexts, and (ii) automated state-space reduction technique. Timed process automata model dynamic networks of continuous-time communicating control processes which can activate other processes. We show how to automatically establish safety and reachability properties of timed process automata by reduction to solving timed games. To mitigate the state-space explosion problem, an automated state-space reduction technique using compositional reasoning and aggressive abstractions is also proposed. Before working on timed process automata, we did a survey on semantics, decision problems, variants, and tools of timed automata. The insights gained from this survey motivated us to use timed game theory and Uppaal Tiga in a couple of industrial case studies and the development of timed process automata. Both the case studies show that state-space explosion is a severe problem for timed games. Suitable abstractions, however, dramatically improve the scalability of timed games in one case study. These case studies motivate the development of timed process automata and an automatable state-space reduction technique for them based on aggressive abstraction.

Co-Authorship

Chapter 2 was published in a paper and appeared in the journal *Computer Science Review* co-authored with Juergen Dingel and Karen Rudie. The next chapter—synthesis of a reconfiguration service for mixed-criticality multi-core systems—was co-authored with Andrzej Wąsowski, Juergen Dingel, and Karen Rudie and appeared in the 11^{th} *International Symposium on Formal Aspects of Component Software*. Chapter 4 was also co-authored with Andrzej Wąsowski, Juergen Dingel, and Karen Rudie and appeared in the 16^{th} *International Conference on Verification, Model Checking, and Abstract Interpretation*. For all the papers, I am the primary author, and conducted the research under the supervision of Juergen Dingel and Karen Rudie.

Acknowledgments

He who does not thank the people is not thankful to Allah.

Prophet Muhammad (PBUH)

By the grace of almighty Allah, my fortune favored me in an extraordinary fashion during the past six years.

I am extremely lucky to have Karen Rudie and Juergen Dingel as my PhD supervisors. They helped me a lot by providing their valuable feedbacks on uncountable drafts of my writings. Because of their continuous support, I never had to worry about my PhD funding, and I enjoyed the maximal freedom from administrative works, which helped me to concentrate only on my research. Karen and I had conflicting opinions on many issues; however, most of the times she allowed and assisted me to work on my choices. She introduced the accurate implementation (or robustness) problem of timed automata to me. The main idea of this thesis was developed during working on that problem. Timed automata was not an area of her expertise. She, hence, had to work harder to supervise my studies. Juergen favored me in many ways but the best one was the opportunity to observe the actions of a man of great morals at his work, which has been helping me to improve my behaviors and ethics. He is a well-known expert in modeling and formal methods communities so with no wonder he had a remarkable influence on my research. For example, the

idea of compositional reasoning for the analysis of timed process automata was introduced by him in one of our meetings. He brought me under the research umbrella of NECSIS, an outstanding research collaboration on automotive software development. He managed a Visiting Researcher position for me at Real-Time Embedded Software Group, University of Waterloo, ON from September 2011 to August 2012. He introduced me with Andrzej Wąsowski, and also sponsored me for a three-month visit to Anderzej's research groups at IT University of Copenhagen, Denmark. Similarly, I received opportunities to work as a Visiting Researcher at Generative Software Development Lab, University of Waterloo, ON and a Visiting Scientist at a major automotive manufacturing company because of Juergen's tremendous efforts. Experience of these positions enriched my thesis and also helped me to attain a research position on formal methods at Ford Motor Company, MI. Once he also tried to find a job for my wife so that I can stay in Kingston during my PhD studies. My family, indeed, received generous supports from both of my supervisors.

Documents show that Andrzej Wąsowski had been one of my academic collaborators and co-authors—however, that does not enough to describe his contributions. He was my mentor, particularly between summer 2012 and fall 2014. On my PhD projects, he spent numerous hours, including many weekend, holiday, and after-hours meetings. This thesis is hard to imagine without his expertise and input on timed automata, timed game theory, and Uppaal tools. I am grateful to him for hosting me in his research labs MT-Lab and Process and System Models between October 2012 and December 2012, which was one of a kind experience for me to know the European research and studies, closely.

I would like to express my gratitude to Joseph D'Ambrosio for introducing the reconfiguration case study, Soheil Samii for discussions on different reconfiguration models, many others who contributed in the initial phase of this case study, and Thomas Fuhrman and

Ramesh S for a series of meetings on the fault tolerant SMP systems scheduling case study.

I am grateful to my PhD thesis examination committee members Mark Lawford, Admed Ahsan, Hossam Hassanein, Thomas Dean, Lawrence Widrow, Juergen Dingel, and Karen Rudie for their comments, which have improved the thesis in a great way. Thanks to Jim Cordy, Ahmed Hasan, Juergen Dingel, and Karen Rudie for serving in my PhD Supervising Committee. Their valuable comments on my progress reports, breadth proposal, topic proposal, and depth report kept my PhD works in a correct direction. I am thankful to Krzysztof Czarnecki and Sebastian Fischmeister for hosting me in their research groups. I also thank Alexandre David for his help with Uppaal Tiga. Thanks to my direct or indirect funding agencies such as NECSIS, NSERC, and Queen's University.

Many people did not influenced my research technically but helped extraordinarily to obtain my PhD. My father Mohammad Wazuddin, my mother Razia Khanam Quraishi, and my wife Tonima Ferdous sacrificed a lot for my studies. My sisters Tasnima Aziza, Fahima Sadia and their families have been taking care of my old and sick parents; otherwise, I would not be able to continue my studies. I am grateful to my mother-in-law Rokeya Begum for taking care of my family for the last six months of my PhD studies, when Tonima suffered from prenatal and postpartum complications. My late grandfather Rafiq Quraishi's encouragements and memories served as motivations, especially during the difficult spells of my studies. Thanks to my teachers, extended family, friends, Queen's staffs, and Ford Motor Company for their contributions and supports. At the end, love for my son Tamjeed Yaseen Waez with whom I wish to spend more daily hours after the thesis submission.

Statement of Originality

I hereby certify that the research presented in this dissertation is my own, conducted under the supervision of Juergen Dingel and Karen Rudie. Ideas and techniques that are not a product of my own work are cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automata are a prominent group of models in model-based development because they facilitate many important types of formal analyses. *Finite automata* (and their derived models, such as *Kripke structures* [159]) can be considered as the most popular, studied, and applied automata because of their rich theoretical properties and practicability. Properties of some systems, however, do not depend only on exact sequence of actions (or communication) but also the exact *time* of execution. Finite automata, implicitly, can model time information using sample timed data. For example, an action a that executes n seconds after the previous action b can be modeled as n special time tick symbols followed by a. Such implicit modeling of time can result in an exponential blowup of both input data and the size of the model. To avoid this problem, this thesis uses *timed automata (TA)* [14, 15], which can be viewed as finite automata with continuous clocks to record time. Timed automata are used over other real-time formal models (such as timed Petri nets [206], timed transition systems [197], and Modecharts) because real-time reachability and some other important analytical properties were first solved using symbolic semantics *region graph* of TA and after that other models adopt the same approach.

This thesis proposes the first compositional modeling with reuse and automatable-state-space reduction technique for the formal analysis of *dynamic hierarchical open real-time systems*. Dense-time model TA do not support compositional modeling with reuse and automatable-analysis for large dynamic hierarchical open systems. More precisely, timed automata are not suitable for modeling and analyzing industrial dynamic hierarchical open systems. This dissertation explores timed automata and its variants in the literature, applies these dense-time models in a couple of industrial problems, and proposes a novel variant timed automata together with a state-space reduction technique for the compositional modeling and analysis of dynamic hierarchical open real-time systems.

Timed automata are desirable for the development of *open real-time systems* since timed automata can capture both discrete-time controllable behaviors of the system and dense-time uncontrollable behaviors of the environment. Timed automata have no structured support for modeling dynamic hierarchical open systems. This absence may lead to cumbersome design details in a large-scale system having several *control hierarchies*. *Timed game automata* [190, 137, 95]—a variant of timed automata—are a well-known model in the research community for the analysis of open dense-time systems. Dense-time formal methods of timed automata may provide the most accurate analysis, however timed automata, currently, are not suited for open systems in practice mainly because of poor scalability.

A system with which an uncontrollable and unknown environment may continuously interact is an *open system*. A *hierarchical open system* is an open system whose components may be other smaller open systems, which also can be hierarchical open systems. A *dynamic hierarchical open system* is a hierarchical open system whose components change over time. A *ground hierarchical open system* is a hierarchical (open) system that does

not have a component. A non-ground hierarchical open system is a *compound hierarchical open system*. A ground hierarchical open system has a control hierarchy of depth 0. A compound hierarchical open system system_1 has a control hierarchy of depth $n + 1$, where n is the maximum of depths of the control hierarchies of the components contained in system_1. Many hierarchical open systems have *dynamic behaviors*, which are activated components only when needed. Dynamic behaviors are an important feature when resource constraints (such as limited memory) do not allow one to keep all the components active at the same time. Models of industrial dynamic hierarchical open systems can be very detailed because of the hierarchical compositionalability. These details may introduce errors in the design and make automated analysis challenging.

1.1 Motivation

The first goal of this thesis is to develop an automatable synthesis technique for reconfiguration services for cost-effective fault tolerance. The next goal is to develop a timed automata-based modeling paradigm for dynamic hierarchical open systems, where a designer will not need to readjust a design for different compositions. However, the main motivation behind this thesis is to develop an automatable state-space reduction technique for timed automata-based analysis of dynamic hierarchical open systems.

1.1.1 Automated Formal Synthesis of Reconfiguration Techniques

Industrial multi-core systems typically use additional processing cores to provide fault-tolerance. Task-level reconfiguration techniques reduce the number of these additional processing cores—thus reducing costs—by reallocating the loads of the failed cores to the non-additional operational cores. The main challenge for developing a task reconfiguration

technique is to provide formal guarantee that the developed technique or framework can handle all fault scenarios. Automated formal synthesis of such reconfiguration frameworks is highly desirable for industrial use.

1.1.2 Three Representations of the Same Component

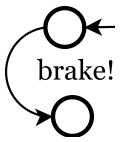


Figure 1.1: A brake actuator as an independent system

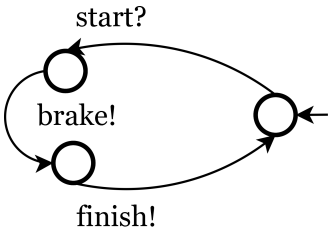


Figure 1.2: A brake actuator as a single copy of a dynamic subsystem

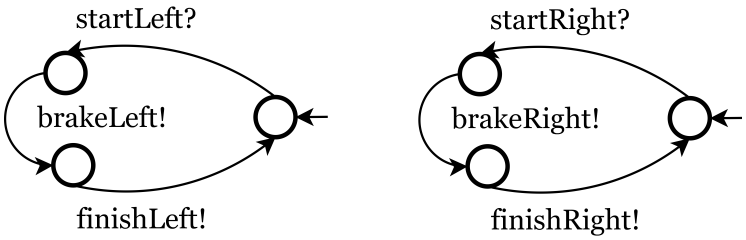


Figure 1.3: Two brake actuators as two copies of a dynamic subsystem

This thesis considers three different kinds of representations of the same component in timed I/O automata-based compositional modeling:

- An independent system.
  - For example, a brake actuator is represented as an independent system in Figure 1.1.
- Single copy of a dynamic subsystem.
  - Figure 1.2, for instance, presents the same brake actuator of Figure 1.1 as a dynamic subsystem by adding additional edges.
- Multiple copies of a dynamic subsystem.
  - Figure 1.3, for example, presents two copies of the same brake actuator of Figure 1.2 as two dynamic subsystem obtained by renaming.

These three manual alterations cause poor reuse and may introduce errors in large industrial designs. For all three scenarios, the thesis aims for only one representation.

1.1.3 State-Space Explosion in Timed Game-Based Analysis: A Case Study

The size of the (monolithic) analysis model of hierarchical systems is exponential in the depth of the hierarchy, due to a product construction and linear in the product of the sizes of all included callee processes. No prior work has been done to improve the scalability of timed games-based analysis of dynamic hierarchical open systems. A generalized automated reduction technique is a necessity for timed games-based analysis of large dynamic hierarchical open systems.

The use of timed automata for the analysis of a real-time control problems in the context of one of our industrial projects failed (even with simplified assumptions) because of the severity of state-space explosion in timed game-based analysis. The goal of the project

was to perform a schedulability analysis of a fault tolerant system using timed games. The effort lasted for around four months. Most of the time was spent to understand the system, the fault model, and the other assumptions.

A *symmetric multi-core processing (SMP)* system has two or more concurrent processing cores, where all the cores are identical. An SMP system periodically executes a finite set of tasks. The worst-case execution time, the deadline, and the release period of every task are known in a fault-free SMP system. The worst-case execution time of a task is not larger than its deadline, and the deadline of a task is equal to its release period. Tasks have priorities.

A fault-free task has three states: suspend, ready, and running. Execution time of a task increases only when the task is in its running state meaning a core is allocated for that task. The deadline of a task decreases in non-suspended states. Every task periodically enters into its ready state at its release period with a constant deadline and zero execution time. No core is allocated to a ready task. A ready task enters into its running state whenever that task starts to execute. A core is occupied by a running task. A running task reenters into its ready state whenever that task is preempted by the scheduler. A running task enters into its suspended state whenever that task terminates. No core is allocated to a suspended task.

We assume that an SMP system can be affected by at most one fault at a time, a fault can occur only at discrete time units, and every fault is permanent. The system is fault-free in its initial system-state. In the other system-states, the system might suffer three types of faults: safety violations by tasks, runaway tasks, and core failures. Tasks may breach safety constraints, for instance, illegal memory access. The scheduler kills and blacklists a task whenever that task violates a safety constraint. A blacklisted task never reenters into suspended, ready, or running state. A task may run away or hang for forever. A runaway

task does not reenter into its suspended state. The scheduler is unable to detect runaway tasks but instantly detects a core failure. Every core of the system may fail, and a task cannot execute on a failed core.

The scheduler ensures that no core is allocated to a task if a higher priority task is in its ready state. The scheduler can preempt a task only at discrete time units. A system is schedulable if and only if no task reaches its deadline before its worst-case execution time.

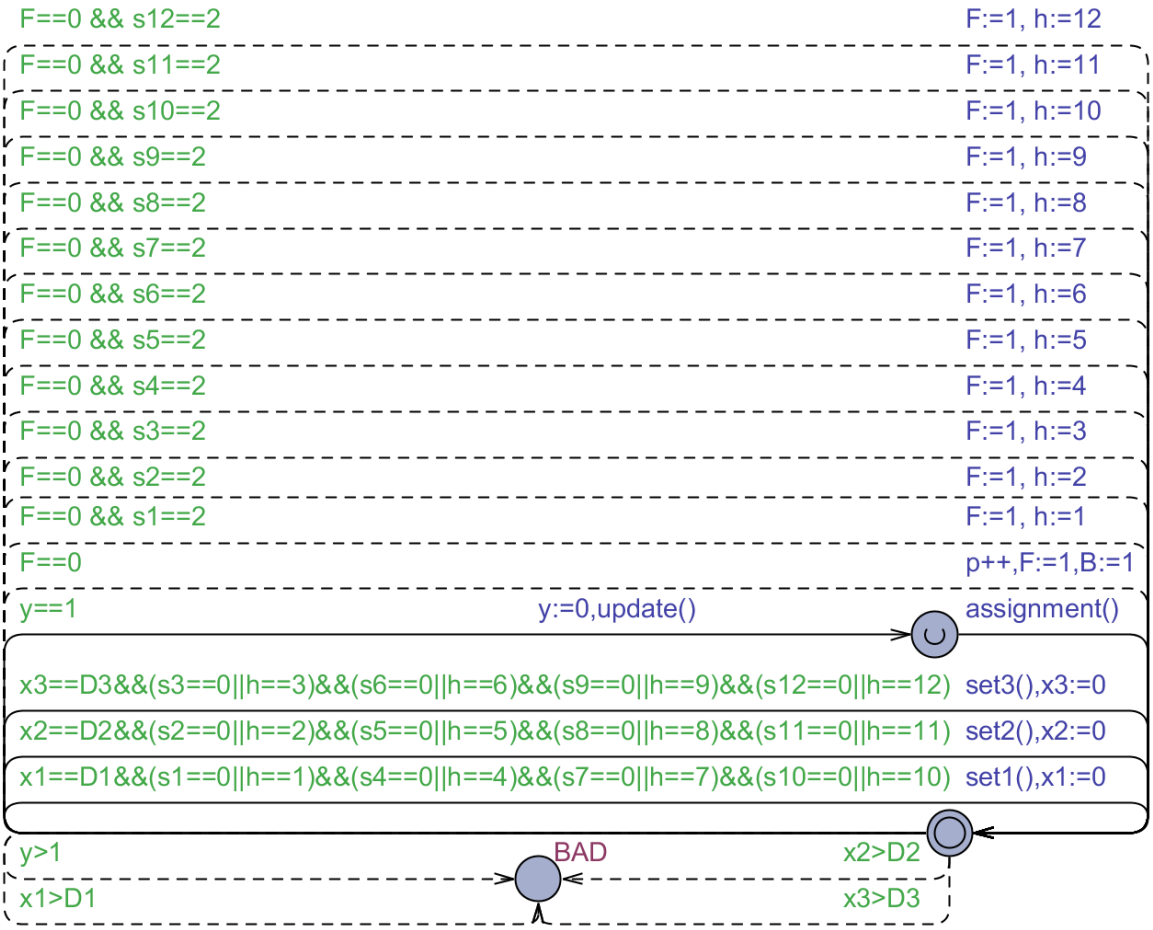


Figure 1.4: A system represented as a timed game automaton in Uppaal Tiga to perform schedulability analysis of the system

We intended to use timed game-based analysis to check schedulability of fault-tolerant

SMP systems. A real-time control problem can be viewed as a two-player timed game [190, 29, 100] between the controller and the environment, where the controller aims to find a strategy to guarantee that the system will satisfy a given property, no matter what the environment does [94]. The purpose of such formulation is to find a strategy for the scheduler to prevent every task to reach its deadline before its worst-case execution time. For illustration, we pick an SMP system having the following configuration:

- Twelve tasks T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, and T12
- Total priority T1> T4> T7>T10>T2>T5>T8>T11>T3>T6>T9>T12
- Different WCETs' W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11, and W12 for tasks T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, and T12, respectively
- Three release periods or deadlines D1 (for tasks T1, T4, T7, T10), D2 (for tasks T2, T5, T8, T11), and D3 (for tasks T3, T6, T9, T12)
- Four symmetric processing cores

We eliminated safety violations of tasks because this type of error does not have any negative impact on the schedulability; instead it makes the system more schedulable for the controller because the scheduler kills and blacklists such tasks. Figure 1.4 presents a model of the system as a timed game automaton in Uppaal Tiga. The model is constructed in a way that the controller cannot make location BAD in the model unreachable when the system is not schedulable. In more specific words, we check the existence of a strategy for the controller in the timed game such that no task reaches its deadline before its worst-case execution time, where the existence means the system is schedulable. Uppaal Tiga, unfortunately, could not perform the analysis because of state-space explosion. To avoid the

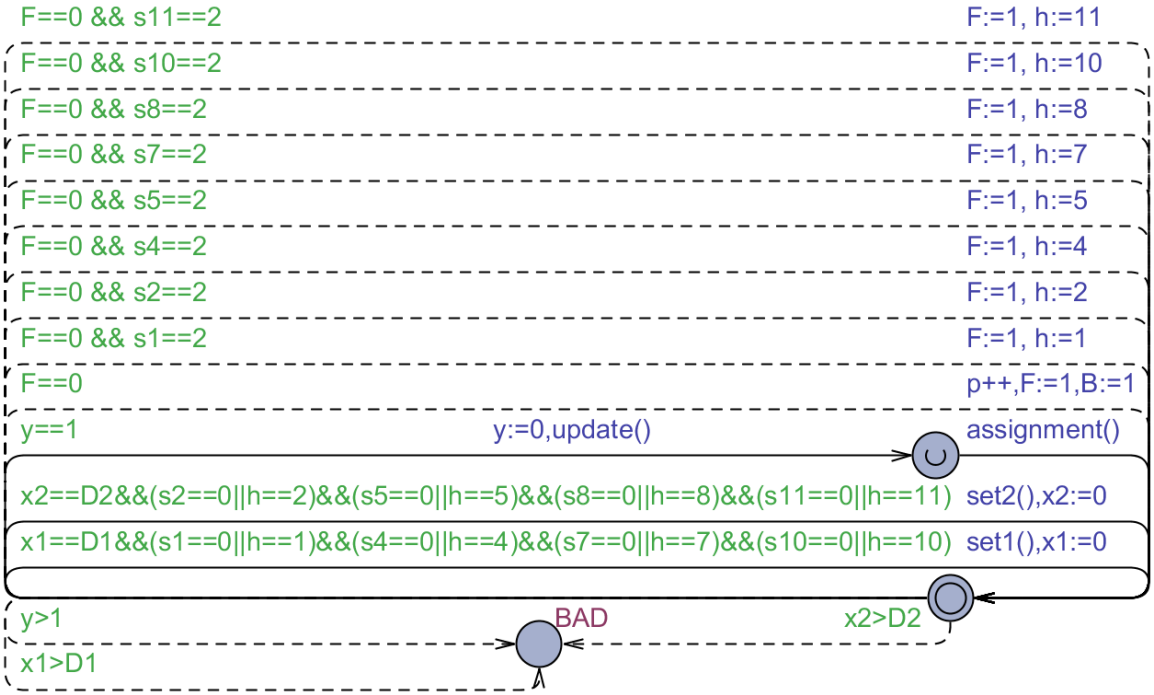


Figure 1.5: The same system of Figure 1.4 with eight tasks and two deadlines

explosion, we remove different resources one by one from the system to find the maximal resources with which the schedulability analysis can be performed. The system can be analyzed with eight tasks (T1, T2, T4, T5, T7, T8, T10, T11) and two deadlines (D1 and D2). Figure 1.5 presents the model of the system with that reduced configuration.

1.2 Problem Statement

The problem we address is to develop an automatable synthesis technique for reconfiguration frameworks using timed automata, and develop a theoretical foundation for timed automata 1) to allow compositional modeling with reuse for dynamic hierarchical open systems and 2) to allow timed games-based automatable analysis for large dynamic hierarchical open systems.

1.3 Challenges

The challenges for the work are the following:

- For automatable synthesis of reconfiguration frameworks using timed (game) automata:
  - Timed games-based synthesis has poor scalability, which has to be taken care of.
- For automatable reuse in compositional modeling:
  - A structured model for timed games is required in which the representation of an independent system can systematically be changed to a component of a larger system.
  - A structured model for timed games is required where an automatable renaming technique can be developed to construct n copies of component C of system system_1 (such as $C_1, C_2, \cdots C_n$) in a way that these new copies can communicate with the other components of system_1 and the environment.
- For automatable analysis:
  - The main challenge for automated analysis based on timed games is scalability. For hierarchical compositional systems, the size of the composition in the monolithic analysis is exponential in the depth of the hierarchy of the system due to the product construction of the state space. Therefore, an automatable state-space reduction technique is needed which can keep the number of control hierarchies constant in the analysis while maintaining enough precision to

obtain useful analysis results, irrespective of depth of the control hierarchies in the actual system.

- The state space in the analysis is also linear in the product of the sizes of all included components of the system. The components of industrial hierarchical systems, unfortunately, typically are very detailed. Therefore, an state-space reduction technique is needed which can keep the size of components constant and small during the analysis while maintaining enough precision to obtain useful analysis results—irrespective of the size of the components.
- Moreover, a well-defined modeling structure for timed games is required to apply an automated state-space reduction technique to analyze time-critical dynamic hierarchical open systems.

1.4 Scope

This dissertation develops a service-based task-level reconfiguration techniques for mixed-criticality multi-core systems within the following scope:

- Engineers of our industrial collaborator aimed to develop service-based solution for task-level reconfigurations to achieve fault tolerance for mixed-criticality multi-core systems. However, they were struggling to provide formal guarantee that the proposed services ensure fault tolerance. This dissertation synthesizes these services with formal assurance to solve their problem. Moreover, this sysntheis process can be automated.

This dissertation also proposes a variant of timed automata for large dynamic hierarchical open systems within the following scope:

- Theoretically, the new variant is not more expressive than the class of timed game automata. For instance, on the semantic level it uses timed games for the analysis. However, the variant allows automatable analysis of larger dynamic hierarchical open systems.

- The thesis also identifies the modeling concepts required for expressing hierarchy and dynamism. In traditional models, these elements grow with the number of layers and make the model complicated and error-prone. The new variant is more suitable for expressing these kinds of systems through the use of new constructs.

- By using existing timed game solvers, the new model allows automated controlled safety and reachability analysis of arbitrary number of dynamic processes; but there is an implicit bound on the maximal number of active processes at a time.

- The thesis develops an efficient automatable state-space reduction technique for the proposed model.

1.5 Contributions

The proposed research aims to improve on the current state-of-the-art in model-based dense-time controllability analysis by developing a semantic model based on timed automata that addresses the limitations stated in Section 1.1. More specifically, the contributions of the proposed research are seven-fold:

Chapter 2 A comprehensive survey of timed automata.

Chapter 3 A novel automatable synthesis technique for reconfiguration services that assures fault tolerance of mixed-criticality multi-core systems.

Section 3.5 Results of experiments provide evidence of the usefulness of aggressive abstractions for state-space reduction.

Section 4.3 A timed automata variant called timed process automata that provides compositionality with reuse feature to model dynamic hierarchical open systems.

Section 4.4 An automated dense-time controllability analysis technique for the developed model.

Section 4.5 An automatable state-space reduction technique for the developed automated controllability analysis, which will allow the analyses of larger dynamic hierarchical open systems.

Section 4.6 Results of experiments to determine effectiveness of the developed state-space reduction technique. The result provides evidence of the usefulness of the technique.

1.6 Organization

The structure of the rest of the dissertation is the following:

- Chapter 2 presents a survey on semantics, closure properties, decision problems, variants, and tools of timed automata. The chapter shows that modeling techniques, automated analyses, and other key issues of timed automata are mostly addressed for *static closed systems*. The survey also reveals that the uses and application domains of timed automata are growing.

- Chapter 3 describes one of our industrial projects. In this case study, we use timed

game automata to automatically synthesize task-level reconfiguration services to reduce the number of processing cores and decrease the cost without weakening fault-tolerance. We apply aggressive abstractions to develop a manual and ad-hoc state-space reduction technique. The technique shows that aggressive abstractions can dramatically improve the scalability of timed games-based tools.

- Chapter 4 proposes timed process automata for modeling and analysis of time-critical systems which can be open, hierarchical, and dynamic. The model offers: (i) compositional modeling with reusable designs for different contexts, and (ii) automatable state-space reduction technique. The use of timed process automata and the reduction technique are described using the case study of the previous chapter.

- Chapter 5 concludes the dissertation.

Chapter 2

State-of-the-Art in Timed Automata: A Survey

Timed automata [14, 15] were introduced by Alur and Dill in the early 1990s. Since then, timed automata have become one of the most dominant formal models to support model driven development (MDD) research of real-time systems. A real-time transition system of a timed automaton can be arbitrarily large due to its ability to express dense time. A real-time transition system can be converted into an equivalent finitely large symbolic transition system called a *region graph*, where reachability is decidable. Later on, *zone graphs* were developed and evolved to provide better scalability in practice compared to region graphs. Decidability of reachability is a core requirement for automated formal verification, and this property of timed automata plays the foremost role to establish timed automata as a major real-time formal model. Rich closure properties and decidability of many important decision problems have contributed to the adaptation of timed automata in many research problems of real-time systems.

During the first two decades of timed automata, many kinds of generalizations and variants of timed automata were proposed and studied to address practically all aspects and features of real-time systems. The strong foundation of timed automata has inspired the emergence of a huge number of tools for analysis, controller synthesis, and code synthesis

for timed automata. This chapter is an attempt to provide an organized description of the development of timed automata and their variants from theory to practice during the first two decades after the birth of timed automata.

This chapter presents a compact discussion on syntax, operational semantics, and two major symbolic semantics, named region and zone, of timed automata. The chapter describes different kinds of analysis techniques such as region-based, zone-based, and flattening-based techniques. Main decision problems and closure properties for timed automata are also listed in this chapter. This chapter classifies around eighty variants of timed automata in an effort to determine current developments. It uses analysis techniques, formal properties, and decision problems to draw distinctions between different versions. Finally, the chapter identifies and classifies forty tools, which are based on timed automata.

This chapter includes only those theorems which are important to describe the works surveyed. The chapter does not provide any proofs or proof sketches. The remainder of the chapter is organized as follows: Section 2.1 discusses the syntax of timed automata, while Section 2.2 explains the operational and symbolic semantics of timed automata. Section 2.3 presents formal linguistic aspects of timed automata. Section 2.4 enumerates variants of timed automata and then classifies them into twelve classes. Section 2.5 presents several academic tools which are based on timed automata.

2.1 Syntax

A timed automaton is a finite state automaton with a set of asynchronous nonnegative real valued clocks and a set of clock constraints. If a timed automaton is considered as a directed graph, locations represent the vertices of the graph, and locations are connected by edges. Locations of a timed automaton are graphically represented as circles. A *clock*

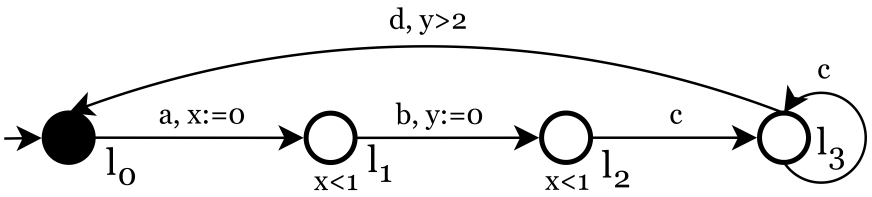


Figure 2.1: A timed automaton with 2 clocks [8]

valuation over the set of clocks is a mapping which assigns to each clock a nonnegative real value. An *initial clock valuation* maps each clock of a timed automaton to zero. The clock constraint which is associated with a *location* is called the *local invariant*[1] of that location. Control can stay in a location only if the clock valuation satisfies the local invariant of that location. Local invariants are used to ensure the progress of the model [140], that is, control can stay in a location until its local invariant permits. Instead of local invariants, *Büchi* or *Muller* acceptance conditions can be used to enforce progress [14, 15]. An edge in a timed automaton is associated with a clock constraint, a subset of the clocks, and a *label*. The clock constraint which is associated with an edge is called the *guard* of that edge. An edge can be traversed only if the clock valuation satisfies the guard of that edge. Clock constraints are used to restrict the timing behaviors of the automaton. Each associated clock of an edge is reset to 0 when the edge traverses. At any instant, the value of a clock equals the time elapsed since the last time it was reset. While edges are instantaneous, time can elapse in a location. Consider the timed automaton [8] in Figure 2.1 with two clocks x and y. The clock x is set to 0 each time the system traverses from l_0 to l_1 on symbol a. The local invariant $x < 1$ associated with the locations l_1 and l_2 ensures that the c-labeled edge from l_2 to l_3 happens within one time unit of the occurrence of a. Resetting clock y together with the b-labeled edge from l_1 to l_2 and the guard of the d-labeled edge from l_3

[1] A timed automaton with local invariants is called a *safety timed automaton* [140].

to l_0 ensures that the delay between b and the following d is always greater than two time units.

Definition 1. *A timed automaton A is a tuple* $\langle L, L_0, L_F, \Sigma, C, E, I\rangle$*, where L is a finite set of locations,* $L_0 \subseteq L$ *is the set of initial locations,* $L_F \subseteq L$ *is the set of final locations,* Σ *is a finite alphabet, C is a finite set of nonnegative real valued clocks,* $E \in L \times \Phi(C) \times (\Sigma \cup \{\epsilon\}) \times 2^C \times L$ *is the set of edges, and* $I : L \longrightarrow \Phi(C)$ *is a mapping that assigns local invariants to locations.*

The set $\Phi(C)$ of clock constraints δ is defined inductively by $\delta := x \sim q \mid x - y \sim q \mid \neg\delta \mid \delta_1 \wedge \delta_2 \mid true$ and $q \in \mathbb{Q}$, $\sim \in \{=, <, >, \leq, \geq\}$, elements of the alphabet Σ are observable actions, ϵ represents unobservable actions, and the set of clocks C is ranged over by x, y etc. The above stated clock constraints only allow one to compare a clock or the difference of two clocks with a rational constant. Clock constraints of the form of $x - y \sim q$ are called *diagonal clock constraints* or *difference clock constraints*. A timed automaton without diagonal clock constraints is called a *diagonal-free timed automata* [50]. A *k-bounded clock constraint* is a clock constraint which involves only constants between $-k$ and k.

An edge $e = \langle l, a, \phi, \gamma, l'\rangle \in E$ from location l to l' can occur and reset the set of clocks $\gamma \in 2^C$ on symbol a if the current clock valuation ν satisfies the guard ϕ, which is noted as $\nu \vDash \phi$. Only the clock constraints which are downwards closed[2] are used as local invariants because a local invariant merely asserts maximum how long control can stay in the associated location.

[2]A clock constraint in the form $x \preceq n$ or $x - y \preceq n$ is downwards closed, where $\preceq \in \{<, \leq\}$ and n is a nonnegative integer.

2.2 Semantics

This section first describes the operational semantics, which could be arbitrarily large for a timed automaton. After that the section describes finite symbolic representations of the semantics of timed automata.

2.2.1 Operational Semantics

A *timed transition system* is a tuple $\langle S, S_0, S_F, \Sigma, \rightarrow \rangle$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $S_F \subseteq S$ is a set of final states, Σ is an alphabet, and $\rightarrow \subseteq S \times (\Sigma \cup \{\epsilon\} \cup \mathbb{R}_{\geq 0}) \times S$ is a *transition relation*.

Definition 2. *The* semantics of a timed automaton $A = \langle L, L_0, L_F, \Sigma, C, E, I \rangle$ *is defined by associating a timed transition system* $\mathcal{TS}(A)$ *of the same alphabet with A [14, 15]: a state in* $\mathcal{TS}(A)$ *is expressed as a pair* $\langle l, \nu \rangle$ *such that location* $l \in L$ *and* ν *is a clock valuation that satisfies the local invariant* $I(l)$*. A state* $\langle l, \nu \rangle$ *is the initial state* S_0 *if and only if* l *is an initial location* ($l \in L_0$) *and* ν *is the initial clock valuation* ν_0*. Similarly, a state* $\langle l, \nu \rangle$ *is a final state if and only if* l *is a final location* ($l \in L_F$)*.* $\mathcal{TS}(A)$ *can have two types of transitions:*

Action transition: $\langle l, \nu \rangle \xrightarrow{a} \langle l', \nu[\gamma := 0] \rangle$ *for an edge* $\langle l, a, \phi, \gamma, l' \rangle$ *if* $\nu \vDash \phi$*, where* $a \in \{\Sigma \cup \{\epsilon\}\}$*,* $\gamma \in 2^C$ *and* $\nu[\gamma := 0]$ *denotes a clock valuation that differs from* ν *only in that clocks in set* γ*, which have been reset to* 0*.*

Time transition: $\langle l, \nu \rangle \xrightarrow{\tau} (l, \nu + \tau)$ *if* $(\nu + \tau') \vDash I(l)$ *for* $\forall \tau' : 0 \leq \tau' \leq \tau$*, where* $\tau \in \mathbb{R}_+$*.*

Due to the real-value time transitions, the state-space of the timed transition system of a timed automaton could be arbitrarily large.

A *timed action* is a pair (t, a), where action $a \in (\Sigma \cup \{\epsilon\})$ is taken by a timed automaton A after $t \in \mathbb{R}_+$ time units since A has been started. The absolute time t is called a *time-stamp* of the action a. A *timed word* is a sequence of timed actions $\xi = (t_1, a_1)(t_2, a_2)...(t_i, a_i)$ where $t_i \leq t_{i+1}$ for $\forall i : i \geq 1$. A *run* of A in $\mathcal{TS}(A)$ with initial state $\langle l_0, v_0 \rangle$ over the timed word $\xi = (t_1, a_1)(t_2, a_2)...(t_i, a_i)$ is a sequence of transitions:

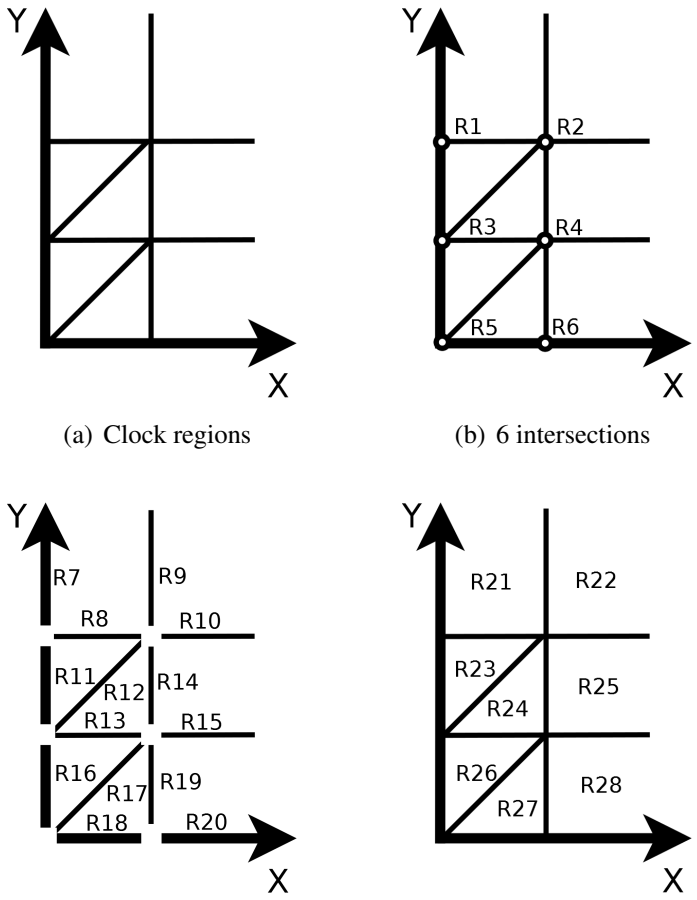
$$\langle l_0, v_0 \rangle \xrightarrow{t_1} \langle l_0, v'_0 \rangle \xrightarrow{a_1} \langle l_1, v_1 \rangle \xrightarrow{t_2 - t_1} \langle l_1, v'_1 \rangle \xrightarrow{a_2} \langle l_2, v_2 \rangle ... \xrightarrow{a_i} \langle l_i, v_i \rangle$$

A run is *accepting* if and only if $\langle l_i, v_i \rangle$ is a final state. The *timed language* Σ_t^* over Σ is the set of all timed words over Σ. The *generated timed language* $L_{gt}(A) \subseteq \Sigma_t^*$ is the set of all timed words for which there exists a run of timed automata A. The set of all timed words with an accepting run of a timed automaton A is the *accepted timed language* $L_t(A) \subseteq L_{gt}(A)$ by A. The *untimed language* L_u is the set of all words in the form $a_1 a_2 a_3 ...$ for which there exists a timed word $\xi = (t_1, a_1)(t_2, a_2)...(t_i, a_i) \in \Sigma_t^*$.

2.2.2 Symbolic Semantics

Exhaustive verification via state-space exploration is not possible on an arbitrarily large state-space. In the last two decades, researchers have made many attempts to convert this arbitrarily largestate space into an abstract state space with a finite, tractable number of states such that the new coarser state space preserves all the important verification properties of the original state-space.

An arbitrarily large state space of a timed transition system $\mathcal{TS}(A)$ can be converted into an equivalent finite state-space of a symbolic transition system called a *region graph* $\mathcal{R}(A)$ [10, 14, 15]. The decidability results e.g., reachability analysis, untimed language inclusion, language emptiness, etc. in timed automata are based on symbolic state-spaces. A *region*, a state of a region graph $\mathcal{R}(A)$, is a pair $\langle l, r \rangle$; where l is a location and r is a set



(a) Clock regions (b) 6 intersections

(c) 14 lines (d) 8 spaces

Figure 2.2: All the 28 clock regions in Figure 2.2(a) for the timed automaton of Figure 2.1: 6 intersections, 14 lines, and 8 spaces

of clock valuations known as a *clock region*. The integral part of a clock value is important to decide whether or not a specific clock constraint is satisfied, while the ordering of the fractional parts is needed to decide which clock will change its integral part first. Two clock valuations ν *and* μ *are in the same clock region*, denoted $\nu \approx^{\mathcal{R}} \mu$, if for any clock x_1 these clock valuations have equal integral part, and for all clocks these clock valuations preserve the order of the fractional parts. If the number of clocks $|C|$ is fixed and each clock

$x \in C$ has a maximal constant m_x, the number of clock regions is finite: the number of clock regions can be at most $|C|!4^{|C|}\cdot\prod_{x\in C}(m_x+1)$ [15]. All clock regions for the timed automaton of Figure 2.1 are shown in Figure 2.2. If $\nu \approx^{\mathcal{R}} \mu$ then $\langle l, \nu\rangle$ and $\langle l, \mu\rangle$ are *untimed bisimilar* (or bisimilar w.r.t. $L_u(A)$) for $\forall l : l \in L$ [15]. As a consequence, untimed bisimulation is used to construct region graphs.

The first attempt to construct region graphs was made on diagonal-free timed automata. Diagonal clock constraints are necessary to model many applications such as scheduling problems [122]. It was shown that a timed automaton A with difference clock constraints can be converted into an equivalent timed automaton A' which has no difference clock constraints [50]. This conversion is based on a region construction. The size of the transformed model is exponential in the number of diagonal clock constraints. The number of clock regions in $\mathcal{R}(A)$ grows exponentially with the number of clocks and the size of maximal constants in the clock constraints. Many techniques for the minimization of region automata have been proposed [11, 140, 226]. None of these proposed techniques has been successful in practice.

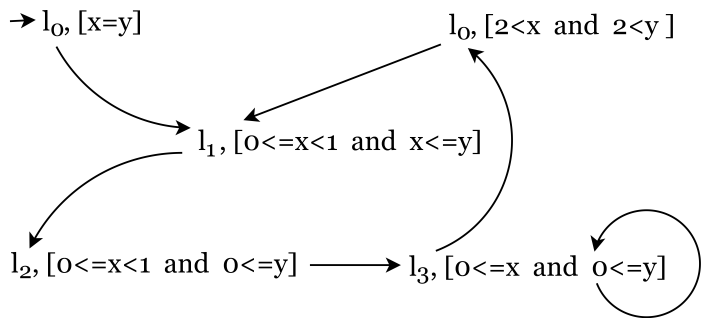


Figure 2.3: Zone graph for the automaton of Figure 2.1 with only 5 zones

A practical efficient abstract state space of a timed automaton A is given by its *zone graph* $\mathcal{Z}(A)$ [106]. A *zone* $\langle l, [\delta]\rangle$ is a pair of a location l and a *clock zone* $[\delta]$, which is the

maximal set of clock valuations satisfying $\delta \in \Phi(C)$. If a timed automaton has n clocks, then its clock zones are convex sets in n-dimensional euclidean space. Every clock region is a clock zone [205]. If the addition of two clock regions (or clock zones) is a convex set then the addition is a clock zone [205]. The number of clock zones is the number of convex unions of clock regions [215]. In the worst case, this number is exponential in the number of clock regions. In practice, clock zones are coarser and more compact than clock regions (e.g., the timed automaton of Figure 2.1 has 28 clock regions as shown in Figure 2.2, while it has only five clock zones as shown in Figure 2.3). Zones have been used to implement all the major timed automata-based tools (e.g., Uppaal [36], Kronos [98]).

For a timed automaton $A = \langle L, L_0, L_F, \Sigma, C, E, I \rangle$, its zone graph $\mathcal{Z}(A)$ is a transition system: states of $\mathcal{Z}(A)$ are zones of A, the zone $\langle l_0, [C = 0] \rangle$ is the initial state of $\mathcal{Z}(A)$ (where $l_0 \in L_0$ and $C = 0$ means that the value of any clock in C is 0), and for every edge $e = \langle l, a, \phi, \gamma, l' \rangle \in E$ and every zone $\langle l, [\delta] \rangle$ there is a transition $\langle \langle l, [\delta] \rangle, a, succ_e(\langle l, [\delta] \rangle) \rangle$; where $succ_e$ is a successor function which returns all the zones which can be reached from the zone $\langle l, [\delta] \rangle$ by first performing the edge e, then letting time pass in the new location, while continuously satisfying the local invariant. The successor function $succ_e$ and reachability analysis in a zone graph are possible because clock zones are closed under the three operations $[\delta_1] \wedge [\delta_2]$, $[\delta]^{\Uparrow,\tau}$, and $[\delta][\gamma := 0]$ where $[\delta_1] \wedge [\delta_2]$ denotes the intersection of $[\delta_1]$ and $[\delta_2]$, $[\delta]^{\Uparrow,\tau}$ denotes the set of interpretations for $\nu + \tau$ for $\nu \in [\delta]$ and $\tau \in \mathbb{R}_+$, and $[\delta][\gamma := 0]$ denotes the set of clock valuations $\nu[\gamma := 0]$ for $\nu \in [\delta]$ and $\gamma \in C$.

A clock zone $[\delta]$ is *closed under entailment* when δ cannot be strengthened[3] without reducing the solution set. A *canonical zone graph* $\mathcal{Z}(A)$ means that for every $[\delta] \in \mathcal{Z}(A)$, there is a unique clock zone $[\delta']$ (where $\delta' \in \Phi(\mathbb{C})$) such that $[\delta]$ and $[\delta']$ have exactly

[3]Let $\delta_1 = \delta \wedge x \preceq n_1$ and $\delta_2 = \delta \wedge x \preceq n_2$ are two clock constraints such that δ_1 and δ_2 have the same solution set and $n_1 > n_2$, then δ_1 can be strengthened by replacing n_1 by n_2 in δ_1.

the same solution set and $[\delta']$ is closed under entailment. Clock zones of a canonical zone graph are represented and manipulated in a data structure called *difference bounded matrices* (DBM) [41, 44, 106], which is the major structure for the efficient implementation of real-time state-space exploration using symbolic semantics.

2.3 Timed Regular Languages and the Decision Problems

A language $L \subseteq \Sigma_t^*$ is a *timed regular language* when there exists a timed automaton A such that $L = L_t(A)$. An untimed language of a timed regular language is a *regular language* [15]. *Timed regular expressions* [27] can be used to represent timed regular languages and operations on them. Some variants [68, 108] of timed regular expressions exist in the literature. In MDD, closure properties and decision problems are crucial for modeling, operations on models, and formal analyses. For example, the underlying languages need to be closed under *intersection* and *shuffle* to model a concurrent system using a synchronous and interleaving semantics, while *emptiness checking* is used to detect the violation of *safety properties* ("nothing bad will happen") in a model. Timed regular languages are closed under *union* [14, 15], *intersection* [14, 15], *concatenation* [27], *projection* [14], *renaming* [14], and *Kleene-star* [27]. Timed regular languages are not closed under *complementation* [14, 15] and *shuffle* [109, 124].

The *emptiness checking* problem for timed automata is PSPACE-complete and can be solved in time $O(|E|\cdot|C|!\cdot 4^{|C|}\cdot(m\cdot m'+1)^{|C|})$, where m is the largest numerator in the constants in the clock constraints and m' is the least-common-multiple of the denominators of all the constants in the clock constraints [15, 20]. *Minimum-time reachability*[4] for timed automata is PSPACE-hard [28, 90, 195]. *Timed bisimulation* [12, 83, 181] and *timed simulation* [218]

[4]Given a timed automaton A, is there a run of A from some initial location $l_0 \in L_0$ to some final location $l_f \in L_f$? If so, find such a run which consumes minimum-time.

are decidable in EXPTIME. *Universality* [15], *language equivalence* [14, 15], *language inclusion* [14, 15], *determinizability*[5] [125, 222], *computing the clock degree* [222, 239], *minimization of the number of clocks*[6] [125, 222], and *reducing the size of constants*[7] [222] for timed automata are *undecidable*.

A *flat timed automaton* is a timed automaton that does not have any nested loops: for every location l there is at most one non-empty path from l to itself. Any timed automaton can be emulated by a flat timed automaton [89]. Comon and Jurski have shown that the *binary reachability* between any two sets of states of a timed transition system of a timed automaton is decidable and they left the complexity issue as an open problem [89]. Instead of conventional region-based or zone-based technique, they first convert a timed automaton to an equivalent flat timed automaton and then use the *additive theory of real numbers* to prove the decidability of binary reachability in a timed automaton. We will call their technique the *flattening technique*. The flattening technique allows one to express and verify some important properties that cannot be expressed or verified by region-based (or zone-based) techniques such as "the delay between event a_1 and event b_1 is never larger than twice the delay between event a_2 and event b_2". On the other hand, their technique is unable to express all the region-based (or zone-based) timing properties.

A *deterministic timed automaton* has at most one initial state, no ϵ-transitions, and no pair of edges which have the same action from the same source location with a common clock valuation which can satisfy the guards of both edges. A deterministic timed

[5]Given an automaton A, does there exists a deterministic automaton B such that $L(A) = L(B)$? If so, construct B.

[6]Given a timed automaton A with n clocks, does there exists a timed automaton B with $n-1$ clocks, such that $L_t(A) = L_t(B)$? If so, construct B.

[7]Given a timed automaton A where constants are not greater than k, does there exist a timed automaton B where constants are not greater than $k-1$, such that $L_t(B) = L_t(A)$? If so, construct B.

automaton has only one run. Deterministic timed automata are *strictly contained* in nondeterministic timed automata [14, 15]. Deterministic timed automata are closed under *union* [14, 15], *intersection* [14, 15], and *complement* [14, 15]. Deterministic timed automata are not closed under *projection* [14, 15] and *renaming* [16]. *Emptiness checking*, *universality*, *language inclusion*, *languages equivalence* problems for deterministic timed automata are PSPACE-complete [14, 15].

2.4 Variants

Many variants of timed automata have been proposed in the literature. There are three primary motivations behind this flourish of variants: the first and most significant one is to improve existing analysis capabilities of timed automata (e.g., optimal path finding [22], schedulability checking [39, 121], memory consumption checking [22, 38, 121], and so forth), the second one is to increase expressiveness by adding modeling features (such as probability [34, 166] or recursion [92]), and the last reason is to increase conciseness of the model [64]. There are also some variants of the semantics [30, 101] to make timed automata a more robust and accurate real-time model. This chapter identifies almost eighty variants of timed automata, and there may be many more. The number is surprising if one considers that the first variant was proposed only two decades ago. The chapter classifies all these variants into eleven classes: *classical timed automata* (Section 2.4.1), *timed automata with other clock constraints* (Section 2.4.2), *timed automata with other clock updates* (Section 2.4.3), *timed automata with other clock rates* (Section 2.4.4), *timed automata with resources* (Section 2.4.5), *timed automata with probability* (Section 2.4.6), *timed automata with communication* (Section 2.4.7), *timed automata with determinizability* (Section 2.4.8),

timed automata with self-embedded recursion (Section 2.4.9), *timed automata with succinctness* (Section 2.4.10), and *timed automata with games* (Section 2.4.11). This classification (Tables 2.1–2.2) is intended to help a reader to understand the major objectives, similarities, and dissimilarities of a huge number of variants of a complicated theoretical research area. According to Tables 2.1–2.2, the class of timed automata with resources has the highest number of variants and the class of timed automata with determinizability has the second highest number of variants. The main motivation behind the flourish of the class of timed automata with resources is to improve expressiveness and analysis capabilities. On the other hand, the goal of the research on timed automata with determinizability is to improve the complexity of key decision problems and to achieve more closure properties. Typically, an increase in expressive power and analysis capabilities comes at the expense of increased complexity and fewer closure properties. Both of these conflicting goals are being extensively researched. This section attempts to provide a glimpse into all these eleven classes by discussing the fewest possible variants with their major decidability results and tools.

2.4.1 Classical Timed Automata

We classify timed automata variants with the same major theoretical properties and expressive power of the *standard timed automata* of Definition 1 as *classical timed automata*. Either a construction rule or an accepting condition of a variant of classical timed automata is dissimilar to other variants of this class. Büchi timed automata [14], Muller timed automata [14], diagonal-free timed automata [14], timed automata with diagonal constraints [14], timed automata with ϵ-transitions [14], timed automata without ϵ-transitions [14], safety timed automata [140], flat timed automata [89], and timed I/O automata [156, 4, 95]

Class	Variants
Classical Timed Automata	Büchi Timed Automata [14], Muller Timed Automata [14], Diagonal-Free Timed Automata [14], Timed Automata with Diagonal Constraints [14], Timed Automata with ϵ-Transitions [14], Timed Automata without ϵ-Transitions [14], Safety Timed Automata [140], Flat Timed Automata [89], Timed I/O Automata [156, 4, 95]
Timed Automata with Other Clock Constraints	Timed Automata with Multiplication Clock Constraints [14], Timed Automata with Periodic Clock Constraints [87], Timed Automata with Additive Clock Constraints [47], Timed Automata with Irrational Clock Constraints [194], Parametric Timed Automata [17], L/U Automata [145] , Timed Automata with ASAP Semantics [101]
Timed Automata with Other Clock Updates	Updatable Timed Automata [64], Suspension Automata [193], Integer Reset Timed Automata [217], Weighted Integer Reset Timed Automata [191], Task Automata [121], Fixed Task Automata [121], Flexible Task Automata [121], Feedback Task Automata [121], Non-Feedback Task Automata [121],
Timed Automata with Other Clock Rates	Hybrid Automata [13], Rectangular Automata [138], Controlled Timed Automata [102], Stopwatch Automata [80], Distributed Time-Asynchronous Automata [110], Distributed Timed Automata with Independently Evolving Clocks [3], Interrupt Timed Automata [49], Robust Timed Automata [129], Perturbed Timed Automata [19]
Timed Automata with Resources	Weighted Timed Automata [22], Priced Timed Automata [38], Uniformly-Priced Timed Automata [37], Dual-Priced Timed Automata [180], Multi-Priced Timed Automata [180], Priced Probabilistic Timed Automata [51], Extended Timed Automata with Tasks [196], Extended Timed Automata with Asynchronous Processes [122], Task Automata [121], Fixed Task Automata [121], Flexible Task Automata [121], Feedback Task Automata [121], Non-Feedback Task Automata [121], Concavely-Priced Timed Automata [155] , Concavely-Priced Probabilistic Timed Automata [152], Timed P Automata [32],Weighted Integer Reset Timed Automata [191], Priced Timed Game Automata [61]

Table 2.1: Classification of the variants of timed automata (part 1)

Class	Variants
Timed Automata with Probability	Discrete Probabilistic Timed Automata [9], Continuous Probabilistic Timed Automata [167], Concavely-Priced Probabilistic Timed Automata [152], First-Order Probabilistic Timed Automata [123]
Timed Automata with Communication	Communicating Timed Automata [158], Communicating Hierarchical Timed Automata [171], Multi-Queue Discrete Timed Automata [209], Omega Deterministic Timed Alternating Finite Automata [120], Synchronized Concurrent Timed Automata [234], Queue-Connected Discrete Timed Automata [147], Phase Event Automata [143] , Timed Cooperating Automata [173], Cottbus Timed Automata [54]
Timed Automata with Determinizability	Event-Clock Automata [16], Event-Recording Automata [16], Event-Predicting Automata [16], Eventual Timed Automata [115], Recursive Event-Clock Automata [141], Product Interval Timed Automata [117], Timed Automata with Input-Determined Guards [116], Continuous Timed Automata with Input-Determined Guards [85], Counter-Free Input-Determined Timed Automata [86], Event-Clock Visibly Pushdown Automata [229]
Timed Automata with Self-Embedded Recursion	Recursive Event-Clock Automata [141], Discrete Pushdown Timed Automata [92], Pushdown Timed Automata [92], Past Pushdown Timed Automata [93], Timed Visibly Pushdown Automata [119], Event-Clock Visibly Pushdown Automata [229], Recursive Timed Automata [228], Timed Recursive State Machines [43]
Timed Automata with Succinctness	Timed Automata with Deadlines [56], Prioritized Timed Automata [185], Variable Driven Timed Automata [219], Timed Automata with Urgent Transitions [33], Alternating Timed Automata [182], Weak Alternating Timed Automata [201]
Timed Automata with Games	Timed Game Automata [190], Priced Timed Game Automata [61], Timed I/O Automata [156, 4, 95]

Table 2.2: Classification of the variants of timed automata (part 2)

are the major variants of this class.

2.4.2 Timed Automata with Other Clock Constraints

This subsection presents variants of timed automata having more expressive clock constraints than classical timed automata. Usually these more expressive variants lose many important theoretical properties to facilitate extra expressiveness. In terms of practical applications, *parametric timed automata* [17] is the most influential and important group of variants in this subsection.

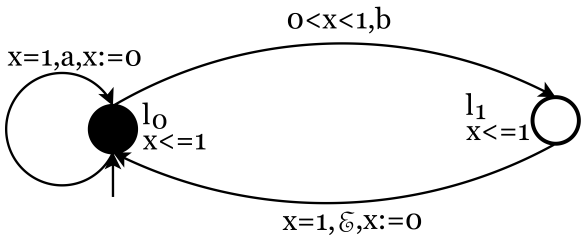


Figure 2.4: A timed automaton with an ϵ-transition having no equivalent ϵ-transition-free timed automata [50]

Timed Automata with Periodic Clock Constraints The class of ϵ-transition-free timed automata is *strictly less expressive* than the class of timed automata with ϵ-transitions [50]. The timed automaton in Figure 2.4 accepts a timed language L_ϵ which can be described as follows: in each open time interval $(i, i+1), i \geq 0$ there occurs at most one b; moreover, there is an a at time $i+1$ if and only if there is no b in $(i, i+1)$. This L_ϵ cannot be accepted by a timed automaton which has no ϵ-transitions. ϵ-transitions *without resets* can be removed from a timed automaton [48] and an ϵ-transition which *does not lie in a loop* can be eliminated [105]. *Periodic clock constraints* are clock constraints of the form $d + n \cdot \theta \leq x \leq e + n \cdot \theta$ or $d + n \cdot \theta \leq x - y \leq e + n \cdot \theta$, where $n \in \mathbb{N}$, $e \in \mathbb{R}$, and $\theta \in \mathbb{R}_+$. Periodic clock constraints can express properties such as "the value of clock x is odd" or "the value of clock x is of the form $0.7 + 4 \cdot n$, where n is some integer". *Timed*

automata with periodic clock constraints in the guards and classical timed automata have the same expressive power [87]. The ϵ-transition-free (deterministic) timed automata with periodic clock constraints in the guards are *strictly more expressive* than the ϵ-transition-free classical (deterministic) timed automata [87]. All ϵ-transitions can be removed from timed automata by using periodic clock constraints and *periodic clock updates*[8] [111].

Additive, Multiplication, and Irrational Clock Constraints Clock constraints of the form of $x + y \sim q$ are called *additive clock constraints*. The emptiness checking problem is undecidable for *timed automata with additive clock constraints* which have four clocks [47]. Timed automata with additive clock constraints having two clocks are strictly more expressive than classical timed automata with two clocks. The emptiness checking problem is decidable for timed automata with additive clock constraints having two clocks [47]. While the emptiness checking problem is still open for timed automata with additive clock constraints which have three clocks. Introducing clock constraints such as $x = q \cdot y$ in the guards makes the emptiness checking problem for timed automata undecidable [15]. Allowing irrational constants in the clock constraints causes the emptiness checking problem to be undecidable [194].

Parametric Timed Automata Timing properties of almost all the real-time protocols are typically not concrete but parametric such as "message delivery within the time it takes to execute two assignment statements" [17]. Concrete timing properties, such as "message has to be delivered within 2 time units and an assignment statement has to be executed within 1 time unit", are applicable only for a specific environment. In MDD of real-time systems,

[8] During a periodic update of a clock that clock is reset to a periodic value instead of 0.

parametric timing properties are very appealing for a real-time model of a reusable software module (which is important in MDD of software) or an off-the-shelf real-time hardware (which is gaining popularity in the automotive industry to cope with different *original equipment manufacturers* (OEM) and brands). Moreover, frequently real-time systems are embedded in diverse environments which forces a designer to model the system according to certain parameters. In the early design phases, parametric models are usually more convenient for a designer compared to concrete models. Parametric timed automata [17], a generalized form of timed automata, can model parametric timing properties by introducing parametric clock constraints. A parametric timed automaton is a timed automaton having an accepting run for a parameter valuation of its parametric clock constraints. The emptiness problem for a parametric timed automaton is described as "is there a parameter valuation for which the automaton has an accepting run?". The emptiness checking for parametric timed automata with three or more clocks is undecidable, while it is decidable with only one clock and is an open problem with two clocks [17]. Parametric timed automata can be divided into *linear parametric timed automata* (where all parametric expressions are linear) and *non-linear parametric timed automata*. An important subclass of parametric timed automata is *lower bound automata* [145], in which parameters are only used to calculate the lower bounds in clock constraints. Similarly, the class of *upper bound automata* [145] is a specialization of parametric automata and parameters in upper bound automata are only used to determine the upper bounds in clock constraints. These two classes of automata are together called *lower bound/upper bound automata* or *L/U automata* [145]. Although L/U automata are a restricted form of parametric timed automata, they can be used to model many noteworthy algorithms and protocols such as *Fisher's mutual exclusion algorithm* [170], and *the root contention protocol* [1]. The emptiness

checking problem for L/U automata is PSPACE-complete [70, 145]. IMITATOR [24] can extract the largest safe[9] subset of parameter values for a parametric timed automaton from a given set of parameter values. HYTECH [135] is also used for the analysis of parametric timed automata such as reachability analysis and operations on states set. Using the open source library REDLIB [237], RED [235] also performs parametric safety analysis, simulation checking, and model checking form parametric timed automata. VerICS [157] and TREX [25] are two other model checkers and analyzers for parametric timed automata.

2.4.3 Timed Automata with Other Clock Updates

Variants of timed automata which add more expressive clock updates to the existing clock reset of classical timed automata are discussed in this subsection. Like the variants of Subsection 2.4.2, variants of this subsection also fail to retain some important theoretical properties of classical timed automata.

Updatable Timed Automata Timed automata with diagonal constraints are *exponentially more concise*[10] than diagonal-free timed automata [62]. Timed automata with diagonal constraints are not more expressive than diagonal-free timed automata [15, 50]. Diagonal constraints may yield different behavior in an extension of timed automata called *updatable timed automata* [64]. Unlike a classical timed automaton, when a edge is taken, an updatable timed automaton can update a specified subset of clocks to values other than 0. An update u of a clock x is deterministic if u has at most one possible value to assign

[9] Safe in a sense that the model is guaranteed not to violate a set of specified safety properties.

[10] The *size of an automaton* A, denoted $|A|$, is the length of its (binary) encoding (states and transitions) on the tape of a *Turing Machine*. Automaton A_1 is exponentially more concise than automaton A_2 if these two automata are language equivalent and $|A_1|$ is polynomial in n, where $|A_2|$ is at least exponential in n.

as $\nu'(x)$ for any clock valuation ν, where $\nu'(x)$ is the value of x after the update u. An example of deterministic update is $x := c$, whereas $x :> c$ is an nondeterministic update which can assign any value as $\nu'(x)$ which is greater than c. The emptiness checking problem for updatable timed automata with updates of the form $x := x - 1$ or $y + c <: x :< z + d$ is undecidable, where $c, d \in \mathbb{Q}_+$ [64]. Only allowing updates of the form $x := c$ or $x := y$ or $x :< c$ keeps the emptiness checking problem PSPACE-complete [64]. Updatable timed automata behave surprisingly for updates of form $x := x + 1$ or $x := y + c$ or $x :> c$ or $x :\sim y + c$ or $y + c <: x :< y + d$; because these updates make the emptiness checking problem for updatable timed automata with diagonal constraints undecidable, while the emptiness checking problem for diagonal-free updatable timed automata with these updates is PSPACE-complete [64]. Updatable timed automata for which the emptiness problem is decidable can be converted into equivalent classical timed automata [64]. These decidable updatable timed automata are more concise than classical timed automata [64].

Suspension Automata A *bounded subtraction clock update* [121] is a clock update of the form $x := x - n$ if $n \leq \nu(x) \leq k(x)$, where $n \in \mathbb{N}_0$ and $k(x)$ is the ceiling for clock x. *Suspension automata* [193], a variant of timed automata, use *stopwatch*-like clocks and bounded subtraction clock updates along with $x := 0$. The language emptiness checking problem and the language inclusion problem for suspension automata are decidable [193]. However, the language emptiness checking problem for timed automata with *unbounded subtraction clock update* in the form $x := x - n$ is undecidable [64].

Integer Reset Timed Automata The class of *integer reset timed automata* [216, 217] is a subclass of classical timed automata since it can reset a clock to zero only when it has an integer value. Edges without reset can occur at any time including at fractional times.

Integer reset timed automata are less expressive than classical timed automata, e.g., integer reset timed automata cannot distinguish between the time stamps of actions occurring within a unit open interval $(i, i + 1)$. Although language inclusion for classical timed automata is undecidable, it is *decidable* to check whether a timed regular language contains an integer reset timed regular language [217]. Contrary to classical timed automata, integer reset timed automata are closed under complementation [217].

2.4.4 Timed Automata with Other Clock Rates

The evolving rate of change for continuous time clock is called *clock rate* of that clock. All clocks of a classical timed automata have the same monotone clock rate. Adding different kinds of clock rates with classical timed automata gives birth to a very expressive, challenging, and popular arena of formal methods called formal methods for hybrid systems. Many researchers consider these variants a completely separate class from timed automata called *hybrid automata* [13].

Rectangular Automata and Controlled Timed Automata Each clock may have a different clock rate in *rectangular automata* [136, 138], which is an interesting extension of timed automata. Each clock rate is bounded by upper and lower bound constants. In a rectangular timed automata each clock can have its own and bounded variable clock rate. A clock can change its clock rate only after performing reset operation in *initialized rectangular timed automata*, where the initialization property states that whenever the rate of a clock changes it must be reset. For each initialized rectangular automaton there is an equivalent timed automaton [138]. Thus the reachability problem is decidable for this variant. Relaxing either the clock rate boundedness or the initialization assumption leads to undecidability of the reachability problem. Like rectangular automata, clocks in *controlled*

timed automata [102] have variable clock rates. Controlled timed automata also allow periodic clock constraints and stopwatch-like clocks. *Stopwatches automata* [80], *interrupt timed automata* [49], and *distributed time-asynchronous automata* [110] are other two general variants of timed automata which use stopwatch to increase the expressive power of timed automata. *Distributed timed automata with independently evolving clocks* [3] are inspired by distributed time-asynchronous automata and execute in a network of timed automata each of which may have different clock rates.

Hybrid Automata *Hybrid systems*—e.g., *biological cell networks* [127]—are described by the combination of analog and digital inputs and outputs. Hybrid automata [13], probably the most famous and most expressive generalization of timed automata, can model hybrid systems. Hybrid automata thus model discrete controllers embedded within an analog environment e.g., a digitally controlled drone flies in a continuously changing environment. A hybrid automaton is a finite automaton associated with real-valued variables whose trajectories obey general dynamic laws described by differential equations. Under specified conditions a hybrid automaton can change to different dynamic laws. There are many subclasses of hybrid automata which are not timed automata such as *non-initialized rectangular automata* [136], *affine hybrid automata* [127], *polynomial hybrid automata* [126]. The area of hybrid automata is exceedingly large, for example timed automata can be seen as a subclass of hybrid automata, and out of the scope of this chapter. Interested readers can read surveys on hybrid automata [78, 97, 134, 169, 220, 225].

2.4.5 Timed Automata with Resources

This group of variants has been introduced almost a decade after the introduction of classical timed automata. This group of variants has quickly received a lot of attention because of

the significance of resources in real-time systems. Now there are at least 18 variants which can be classified as timed automata with resources. No other class of timed automata has so many variants.

Weighted Timed Automata or Priced Timed Automata In MDD, a timed automaton serves as a superior model for a real-time system over a finite state automaton because timed automata can explicitly assert time constraints. A classical timed automaton, however, is unable to inform the designer how many resources—such as, bandwidth, power, development time, money, and so forth—its implementation will consume. This resource consumption information (especially optimal resource consumption) may play a crucial role in MDD. A designer can extract the total resource consumption information of the implementation from a model if the designer attaches a resource consumption function to each state and to each transition of that model. A timed automaton with resource consumption functions is more desirable than a finite state automaton with resource consumption functions when the resource consumption is proportional to the units of time the implementation stays in a state. After recognizing the absence of timed automata with resource consumption functions, Alur et al. and Larsen et al. independently introduced timed automata with resource consumption functions in 2001, and called them *weighted timed automata* [22] and *priced timed automata* [38], respectively. A *weighted/priced timed automata* consists of a timed automaton A and a price/cost function $\mathcal{P}$ that maps every location $l \in L$ and every edge $e \in E$ to a nonnegative rational number: $\mathcal{P}(l)$ is the cost for staying in l per unit of time and $\mathcal{P}(e)$ is the cost for performing the edge e. Thus, every run in a weighted/priced automaton has its own accumulated cost and an automaton may have many runs. As a result, to reach a location from a source location with optimal cost (minimum or maximum cost) is an important decision problem for weighted/priced automata.

This problem is called *optimal reachability* and is decidable [22, 38]. Optimal reachability is a general form of minimum-time reachability. The optimal-reachability problem for weighted/priced timed automata is PSPACE-complete [22, 38, 60]. Optimal reachability and *optimal scheduling* using weighted/priced automata are well studied and supported in Uppaal CORA, a variant of Uppaal, which is a specialized tool for optimal reachability and optimal scheduling [39, 40, 176]. REMES-IDE can transform REMES [210] (REsource Model for Embeded Systems) models into behaviorally equivalent weighted/priced timed automata [148]. REMES-IDE provides a graphical editor for the resulting priced automata, as a tool to visually inspect transformation results. Model files for both Uppaal (timed automata) and Uppaal CORA (weighted/priced timed automata) can be exported to REMS-IDE for verification and analysis. Because of the good analytical power of weighted/priced automata, they have been studied extensively and many variants have been proposed such as *uniformly-priced timed automata* [37], *dual-priced timed automata* [180], *multi-priced timed automata* [180], *concavely-priced timed automata* [155], *priced timed game automata* [61], *concavely-priced probabilistic timed automata* [152], *weighted integer reset timed automata* [191], and *priced probabilistic timed automata* [51, 52]. More interesting information about weighted/priced automata may be found in an article [65] by Bouyer et al..

Task Automata or Timed Automata Extended With Real-Time Tasks Finite automata can only describe the *arrival sequence* among the actions, while classical timed automata can describe both the arrival sequence among the actions and the *arrival time* of an action. Like finite automata, classical timed automata also describe every action as an instantaneous instance. Norström et al. [196] have extended timed automata by adding real-time tasks with actions. Later on their work evolved into *task automata* or timed automata

extended with real time tasks with locations [121, 122, 162]. A task is an executable program. A task can be described by its *task type* (or *task name*), *best case computational time*, *worst case computational time*, *relative deadline*, *priority for scheduling*, and (occasionally) resource consumption information. Timed automata in the form of task automata have been studied for several important analyses such as schedulability, boundedness checking, non-Zenoness checking, resource consumption computation, and so forth. Compare to the other variants of timed automata, task automata are a natural model for code synthesis if the target platform ensures the *synchrony hypothesis*, that is, the run-times of related system functions are negligible compared to the different execution times of the associated tasks of the model. TIMES [23], based on task automata, is a popular tool in the research community for real-time code synthesis and scheduling. Schedulability analysis problems of task automata for multi-processor platforms have been studied in [161].

Timed P Automata A biologically inspired—specifically, the structure and the functioning of living cells—model called *P systems*[11] [204] has received huge attention[12] in the area of theoretical computer science for its impressive computational and modeling power. Membrane computing naturally models mobility, distributed parallel computing, biomolecular systems, and ecological systems. A P system comprises a hierarchy of membranes: each of these membranes contains a multiset of reactant objects and possibly other membranes. An evaluation rule describes reactants and the resulting product. An evaluation rule can be applied only to objects of that membrane. In *timed P systems* [82], a variant of P systems, each evolution rule is associated with an integer which represents the number of

[11] P Systems was introduced in 1998 by Gheorghe Păun, whose last name is the source of the letter P in 'P Systems'. For more information on P systems please visit http://ppage.psystems.eu/.

[12] On 3rd October 2003, membrane computing has been selected by Thomson Institute for Scientific Information (ISI) as a "Fast Emerging Research Front in Computer Science".

time units needed by the rule to be entirely executed. *Timed P automaton* [32] is a timed automaton with a discrete time domain where every location is a timed P system. Timed P automata are useful to study a population which dynamically changes with time e.g., the population of a place whose dynamics changes with seasons.

2.4.6 Timed Automata with Probability

Any real-time property can be either a *hard real-time property* (e.g., "the car stops within 800 time units after the break is applied") or a *soft real-time property* (e.g., "at most 3% of all the messages will not be delivered within 5 unit of times"). While hard real-time properties are essential in many safety critical real-time systems (e.g., robotic surgery), soft real-time properties are required for many commonly used real-time systems (e.g., video streaming). Unfortunately, classical timed automata and all its variants discussed above cannot support soft real-time properties. To serve as a complete model for the MDD of real-time systems, timed automata have to have support for the specification and analysis of soft real-time properties along with hard real-time properties. Soft real-time properties are frequently used in fault tolerant real-time systems (e.g., communication protocols, multimedia protocols) where hard real-time properties are too restrictive: violating a deadline does not affect the functionality of a fault tolerant protocol. Every edge of a *probabilistic timed automaton encodes* its likelihood to occur. This likelihood is calculated from the execution of certain actions by the system. Hence, probabilistic timed automata can be used to evaluate *quality of service* which is the quantitative estimation of the probability of achieving some target (e.g., perform a certain task in a time bound). Soft real-time properties are supported by *discrete probabilistic timed automata* [9, 34, 150, 166]. An expressive generalization of discrete probabilistic timed automata has been proposed called *first-order*

probabilistic timed automata [123]. Clocks in a *continuous probabilistic timed automata* [167] can be reset according to *continuous probability distributions*. On top of soft deadline properties, continuous probabilistic timed automata also enable *stochastic timing*, that is, soft deadlines must be satisfied under the assumption that some set of events is influenced by a certain continuous time probability distribution. An example [168] of stochastic timing properties is "the arrival rate of video frames is normal with mean of 40ms and variance of 5ms, and service is exponential with rate 45ms". Thus stochastic timing properties can estimate some important *performance parameters* such as *throughput* and *mean service time*. Among tools, Uppaal PRO and Fortuna [52] can analyse *maximum probabilistic reachability properties* of probabilistic timed automata. PRISM 4.0 [165] provides more general support for the verification and analysis of both discrete and continuous probabilistic timed automata. mcpta [133] is another model checker for probabilistic timed automata.

2.4.7 Timed Automata with Communication

Concurrent and communicating models are ideal to model mobile systems, cloud computing, and concurrent embedded systems. Untimed concurrent and communicating models widely use FIFO channels (queues) to communicate among them. Channels are also common in real-time concurrent and communicating models such as *communicating real-time state machines* [212] and π_{klt}*-calculus* [203]. Krcál and Yi developed *communicating timed automata* in 2006 [158]. A communicating timed automata is a network of timed automata extended with unbounded channels. Untimed communicating finite state models are not more expressive than classical finite state automata. A communicating timed automaton with only one channel and no sharing states has the power of a one-counter machine. In contrast, a communicating timed automaton with only two channels and no

sharing states has the power of two-counter machines or Turing machines, thus channels make the verification of communicating timed automata more difficult [158]. Other timed automata variants which also use channels to communicate are *multi-queue discrete timed automata* [209], *omega deterministic timed alternating finite automata* [120], synchronized concurrent timed automata [234], and queue-connected discrete timed automata [147]. An interesting timed automata variant with communication is *phase event automata* [143], which combines both state-base (e.g., Kripke structure) and event-based (e.g., finite state automata) structures. The advantage is one can combine the benefits of both process algebra (which depends on event-base structure) and model-checking (which depends on state-base structure).

A state in a *hierarchical state machine* can be either a normal state or a super-state, which contains some other states. Although hierarchical state machines, for example, STATECHARTS [132], UML [55], are a widespread model in MDD, very little research has been done to understand their theoretical aspects such as expressiveness, decision problems, concurrency complexity, and formal (unambiguous) semantics. Alur et al. [18] published one of the first publications on the topic of the decision problems and succinctness of (untimed) *communicating hierarchical state machine*. Inspired by their work another group came up with *communicating hierarchical timed automata* [171, 172] to study theoretical aspects of real-time hierarchical state machines. Like (untimed) communicating hierarchical state machines, the reachability problem for communicating hierarchical timed automata is EXPSPACE-Complete.

Beyer and Rust developed a hierarchical variant of timed automata for modular specification. The name of their variant is *Cottbus timed automata*, which are developed in Cottbus, Germany [54]. Rabbit [53] is a reachability and refinement-checker for Cottbus

timed automata. Another hierarchical timed automata variant is *timed cooperating automata* [173, 174]—a real-time variant of *cooperating automata* [112].

2.4.8 Timed Automata with Determinizability

Alur, Fix, and Henzinger [16] proposed a determinizable subclass of timed automata named *event-clock automata* after determining that the major obstacle to achieving determinizability of classical timed automata is nondeterministic clock resets. All the clocks in an event-clock automaton are divided into two disjoint sets: one set contains only *event-recording clocks* and another set has only *event-predicting clocks*. Every action or event in event-clock automata has a one-to-one relation with an event-recording clock and with an event-predicting clock. All the clocks in an event-recording automaton are associated with actions and the number of actions are fixed, thus the number of clocks is fixed. An event-recording clock records when the associated action occurred the last time, and an event-predicting clock shows when the associated action will occur next time. Event-clock automata do not have any ϵ-transitions. Removing all the event-predicting clocks from an event-clock automaton will convert it into an *event-recording automaton*. Similarly, eliminating all the event-recording clocks from an event-clock automaton will transform it into an *event-predicting automaton*.

Event-clock (or event-recording or event-predicting) automata are determinizable thus they are closed under complement. Event-clock (or event-recording or event-predicting) timed automata are closed under all the Boolean operations. The language-inclusion problem for event-clock automata is PSPACE-complete [16]. Dima defined a class of regular expressions equivalent to event-clock automata [107]. D'Souza discussed the logical characterization of event-clock automata and event-recording automata [113, 114]. *Event-clock*

visibly pushdown automata [229] and *recursive event-clock automata* [141] have also been proposed for determinizable self-embedded recursive timed automata. *Product interval timed automata* [117] are a subclass of event-recording automata that can be used to model the timed behavior of asynchronous digital circuits. Other related timed automata variants are *timed automata with input-determined guards* [116], *eventual timed automata* [115], *counter-free input-determined timed automata* [86], and *continuous timed automata with input-determined guards* [85]. TEMPO [214] is a model checker for event-recording automata and was first released in 2001.

2.4.9 Timed Automata with Self-Embedded Recursion

Self-embedded recursion[13] can model naturally the control flow of sequential computation in typical programming languages with nested and recursive invocations of program modules. A *pushdown timed automata* [92] is a variant of classical timed automata which can express real-time self-embedded recursive properties by augmenting a timed automaton with a stack. Many real-time non-regular properties are required for real-time software verification. Unfortunately, introducing self-embedded recursion destroys many important closure properties (e.g., intersection) for modeling and verification. Therefore, these kind of properties are usually handled by less expressive but practically efficient *finite indexing* techniques such as bounded real-time model-checking [202].

The binary reachability of a pushdown timed automaton is decidable [92]. The binary reachability of *past pushdown timed automata* [93], a parametric variant of *discrete pushdown timed automata* where the past-formulas[14] can be used as clock constraints, is also decidable. The universality problem and language inclusion problem for *timed visibly*

[13] Balanced parentheses languages are well known examples for self-embedded recursion.
[14] A past formula is a formula which includes the past parametric values.

pushdown automata [119], nondeterministic timed version of *visibly pushdown automata* [21], even with a single clock is undecidable.

A deterministic timed automata version of visibly pushdown automata called event-clock visibly pushdown automata [229] is closed under Boolean operations. It is decidable to check whether a timed visibly pushdown language is included in an event-clock visibly pushdown language [229]. *Recursive timed automata* [228] and *timed recursive state machines* [43] were proposed in 2010.

2.4.10 Timed Automata with Succinctness

The main motivation behind the creation of this group of timed automata variants is to improve modeling rather than to achieve better analyses.

Alternating Timed Automata An (untimed) *alternating finite automaton* [84] is a nondeterministic finite automaton whose transitions are divided into existential and universal transitions. Let A be, for example, an alternating automaton; for an existential transition $(s_1, a, s_2 \vee s_3)$, A nondeterministically chooses an edge from state s_1 to either s_2 or s_3 after reading a—like a nondeterministic finite automaton; for a universal transition $(s_1, a, s_2 \wedge s_3)$, A moves to s_2 and s_3 after reading a, where the transition simulates the behavior of a parallel machine. An alternating finite automaton accepts a word when there exists a run tree on that word such that every path ends in an accepting state. A run is represented by a run tree due to the universal quantification. Any alternating finite automaton is equivalent to a nondeterministic finite automaton. Alternating models are useful to express clauses which are combined by Booleans. *Alternating (tree) timed automata* [103], a real-time extension of alternating automata, are closed under all Boolean operations [182, 198]. Emptiness checking for alternating timed automata is decidable only for one clock over finite timed

words; any extension, such as, infinite timed words, more than one clock, ϵ-transitions, leads to undecidability [182, 198]. Undecidability proofs of the emptiness checking problem for alternating timed automata with one clock over infinite words rely on the ability to express "infinitely often" properties. *Weak alternating timed automata* [201] do not permit one to express "infinitely often" properties, thus the emptiness checking problem for weak alternating timed automata over infinite words is decidable. Interestingly, bounded time model checking of alternating timed automata over finite or infinite words is decidable as in bounded time the emptiness checking is decidable [149]. TCTL model checking for alternating timed automata has also been discussed [103].

Timed Automata with Deadlines *Urgency—urgent transitions* and *urgent locations*—is a common and important concept in real-time models, such as, in timed Petri nets [151], because they allow more succinct representation and resolution of non-determinism in real-time concurrent models. When an urgent transition is enabled the control of the automaton has to perform the transition instantaneously without spending any time at that location. All the transitions originating from an urgent location are urgent transitions. Urgency has been first introduced by Bornot et al. with timed automata as *timed automata with deadlines* [56]. Later on many others generalized timed automata with deadlines. Among them Brabuti and Tesei proposed a model, which is called *timed automata with urgent transitions* [33]. In Brabuti's model, an urgent transition must be performed within a fixed time interval from its enabling time and an urgent transition has higher priority than other non-urgent transitions enabled in the same state. Although from a language point of view timed automata with urgent transitions are not more expressive than classical timed automata, from a specification point of view the use of urgent transitions allows shorter and clearer specifications of urgent and periodic behaviors. *Variable-driven timed automata* [219] and

prioritized timed automata [185] are two additional timed automata variants mainly focusing on urgency issues. Uppaal [36], Uppaal Tiga [35] and many other tools use urgency for the specification of their models.

2.4.11 Timed Automata with Games

A classical timed automaton models only closed real-time systems where every thing is controlled while there exist many open real-time systems which interact with uncontrolled environments (or other systems) and these uncontrolled environments influence the behavior of those systems. A good example of real-time open systems is a pacemaker (an open system) which continuously interacts with a heart (an uncontrolled environment). Pacemaker's performance crucially depends on the exact timing of an action performed either by the system or by the environment. *Timed game automata* [190] along with their controller synthesis strategies have been introduced to develop such open real-time systems. The *game reachability* problem is whether the system has a strategy to reach a target state regardless of how the environment behaves. The *game minimum-time reachability* problem in timed game automata is finding the minimum time required by the system to reach a target state regardless of how the environment behaves. Both the game reachability and the game minimum-time reachability problems for timed game automata are EXPTIME-complete [74, 137, 154]. Bouyer et al. have discussed optimal strategies in priced timed game automata, which is a combination of timed game automata and priced timed automata [61].

A timed I/O automaton [156, 4, 95] is a timed automaton which has an input alphabet along with a regular output alphabet. The controller plays controllable output transitions and the environment plays uncontrollable input transitions; thus timed I/O automata are

a natural model for timed games. Uppaal Tiga [35], a timed I/O automata-based tool, is a well-known tool for solving games based on timed game automata with respect to reachability and safety properties. Synthia [118], a tool developed in Saarland University, Germany, performs verification and controller synthesis for timed game automata.

Variants of this class, timed automata with games, are a direct generalization of classical timed automata. This class itself is emerging as a big research area mainly because of its controller synthesis application. As for the case of hybrid automata, this survey does not elaborately discuss the timed automata with games class because of its vastness. An interested reader of timed games may find the article [81] to be an easy and comprehensive reading.

2.5 Tools

The rich and strong theoretical foundation of timed automata makes them a good candidate to use as the underlying formal model for real-time MDD. Region-based approaches are not suitable for practical purposes because of the exponential size of region automata. Most of the tools use zone-based approaches as these approaches are practically more efficient and scalable. Zone-based techniques have changed a lot over time to overcome many deficiencies. This large number of changes has made it challenging to keep these tools up to date. A few tools, such as, Uppaal [36], RED [235], and VerICS [157], have been actively maintained and have evolved for a long period of time. A bright future waiting for timed automata-based tools as zone-based techniques are now well-established. A complete and detailed survey paper on timed automata-based tools could be an appealing work in the research community. A survey on timed automata-based tools is a small part of this chapter and it provides only a skeleton view of timed automata-based tool research. Nonetheless,

this chapter lists forty timed automata-based tools. We did not personally perform any experiments on these tools. Rather, our survey mostly relied on related publications, websites, and developers of each tool to collect information.

Tables 2.3–2.6 list some of the timed automata-based or closely related tools available on the web and in the literature. All these listed tools were developed and are maintained mainly for research and academic purposes. The first column of these tables displays the names of the listed tools; the second column presents the description of the functionality of these tools; at the end, the third and fourth columns show the first release year and the latest release year[15] of these tools. All these release years have been confirmed by the respective developers other than TASM [200], TEMPO [214], HyTech [135], RED [235], TART [131], Fortuna [52], PRISM 4:0 [165], XAL [76], and McAiT [186].

Uppaal [36], probably, is the most successful and influential timed automata-based analysis and verification tool. Uppaal is now also available for commercial use. This tool was developed and is maintained by the Uppaal research group. Uppaal group, a timed automata focused research group, is formally a collaboration between two research groups of Uppsala University, Sweden and Aalborg University, Denmark. Uppaal PRO, Uppaal PORT [144], Uppaal CoVer [142], Uppaal Tiga [35], Uppaal Cora [39], Uppaal TRON [178], TIMES [23], CATS [160], SAVE IDE [211], McAiT [186], and TASM [200] are some other timed automata-based tools that were developed and are maintained by the Uppaal research group. Tools such as McAiT [186], SAVE IDE [211], TASM [200] have been put into the Uppaal group because in addition to having been strongly influenced by the Uppaal tool, some of the major developers (e.g., Wang Yi and Paul Pettersson) of these tools have strong past or present ties with the Uppaal group. Uppaal Tiga [35], TIMES [23], and Uppaal TRON [178] are popular and highly-cited timed automata-based research

[15] We did not consider releases after year 2011.

Name	Description	First Rele.	Latest Rele.
Uppaal [36]	An integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types.	1995	2011
Uppaal PRO	A reachability analysis tool for probabilistic timed automata.	2008	2009
Uppaal PORT [144]	A tool for component-based modeling, simulation, and verification of embedded systems modelled as timed automata.	2006	2008
Uppaal CoVer [142]	A tool for creating test suites from timed automata with coverage specified by coverage observers.	2005	2009
Uppaal Tiga [35]	A tool for solving games based on timed game automata with respect to reachability and safety properties.	2005	2011
Uppaal Cora [39]	A tool for cost optimal reachability analyses for priced timed automata.	2002	2006
Uppaal TRON [178]	A black-box conformance testing tool, based on timed automata, for embedded real-time software.	2004	2009
TIMES [23]	A tool set for modeling, schedulability analysis, synthesis of (optimal) schedules and executable code.	2002	2005
CATS [160]	A tool for compositional timing and performance analysis of systems modeled using timed automata and the real-time calculus.	2007	2007
SAVE IDE [211]	A tool for design, analysis and implementation of component-based embedded real-time systems using timed automata.	2008	2009
McAiT [186]	A timed automata-based analyzer for multicore real-time software.	2010	2010

Table 2.3: Timed automata-based or related tools (part 1)

Name	Description	First Rele.	Latest Rele.
TASM [200]	A tool for specification, simulation, and verification of real-time systems using timed automata.	2007	2008
Kronos [98]	A model checker for timed automata against TCTL formulas.	1992	2002
SynthKro [7]	A tool for controller synthesis of timed automata.	2002	2002
Open-Kronos [227]	A model-checker for timed Büchi automata.	1997	2005
TAXYS [88]	A timed automata-based tool for the development and verification of real-time embedded systems.	2000	2001
minim [226]	A tool for minimization of timed automata with respect to time-abstracting bisimulation.	1996	2001
RTSpin [224]	A verification tool which extends Spin with quantitative dense time features using timed Büchi automata.	1993	2004
IF [69]	A validation platform for a timed automata-based specification language which is expressive enough to represent major modeling and programming languages for distributed systems such as real-time SDL and UML.	1998	2004
TReX [25]	A reachability analyzer for parametric timed automata.	2000	2006
IMITATOR [24]	A tool for extracting the largest safe subset of parameter values for a parametric timed automaton from a given set of values.	2009	2011
CMC [175]	A timed automata-based compositional model-checker.	1995	2004
Synthia [118]	A tool for verification and controller synthesis for timed automata.	2011	2011

Table 2.4: Timed automata-based or related tools (part 2)

Name	Description	First Rele.	Latest Rele.
mcpta [133]	A model checker for probabilistic timed automata.	2009	2011
MCTA [163]	A timed automata-based model checker to provide shortest error trace.	2008	2009
Rabbit [53]	A model checker for Cottbus timed automata.	1999	2002
MIRELA Framework [104]	MIRELA's compiler generates timed automata for simulation and verification of mixed reality applications.	2007	2008
XAL [76]	A web oriented programming language based on timed automata.	2008	2008
WST [75]	A tool for design, validation and verification of composite Web Services with timed restrictions using timed automata.	2007	2011
VerICS [157]	A (bounded, unbounded, parametric, and non-parametric) model checker for networks of communicating automata (such as timed automata, time Petri nets) and for high level languages (such as Promela, UML, Java, and Estelle).	2003	2010
PRISM 4:0 [165]	A verification tool for probabilistic models including probabilistic timed automata.	2010	2011
AITARTOS [164]	A tool for automatic implementation of timed automata model in a real-time operating system.	2010	2011
Fortuna [52]	A model checker for priced probabilistic timed automata.	2010	2010
Priced-Timed Maude [42]	An analyzer for priced timed automata.	2008	2008
RED [235]	A model checker and simulation checker for timed automata and parametric analyzer for parametric timed automata.	2000	2011

Table 2.5: Timed automata-based or related tools (part 3)

Name	Description	First Rele.	Latest Rele.
HyTech [135]	A model checking and analyses tool for linear hybrid automata including parametric timed automata.	1995	2003
TEMPO [214]	A model-checker for event recording automata.	2001	2001
DREAM [188]	A timed automata-based analyzer for distributed embedded systems.	2005	2007
TART [131]	A prototype to generate Java code from timed automata using RTSJ.	2010	2010
VInTiMe [5]	VInTiMe is a suite of timed automata-based tools (Lapsus [73], VTS [6], ObsSlice [71], and Zeus [72]) that combines high-level expressive power, unassisted property-preserving model-reduction and low-level distributed model checking power to describe and verify complex real-time systems.	2003	2009

Table 2.6: Timed automata-based or related tools (part 4)

tools for controller synthesis, code synthesis, and black-box testing, respectively.

Verimag, France is a leading research center in embedded systems that was officially established in 1993. Until 2006, Verimag was active in timed automata-based tool research and development. Many timed automata-based tools such as Kronos [98], SynthKro [7], Open-Kronos [227], TAXYS [88], minim [226], RTSpin [224], IF [69], and TReX [25] are developed and maintained by this research center. Kronos [98] is a very well-known timed automata-based Verimag research tool. However, there has been no official release of Kronos in the last 10 years. Laboratory Specification and Verification (LSV), ENS Cachan, France developed a timed automata-based compositional model checking tool named CMC [175] in 1994, and there is no new version of this tool after 2004. Instead of updating CMC [175], LSV is actively developing a new timed automata-based tool

IMITATOR [24] for parametric analysis. Saarland University of Germany has developed two timed automata-based tools: mcpta [133] for model checking and Synthia [118] for verification and controller synthesis. No other group has developed more than one tool of Table 2.3 and Table 2.4. Almost all the timed automata-based research tools are developed in Europe. Uppaal and Verimag are the main two driving forces of timed automata-based tool research. Release dates in Tables 2.3–2.6 indicate Verimag has not been very active in this research arena recently. A large number of successful tools, the diversity in tools functionality, and the long maintenance period suggest that the Uppaal group is the most established group in this arena.

Purposes	**Tools**
Black-Box Testing and Related	Uppaal TRON [178], Uppaal CoVer [142]
Code Synthesis and Scheduling	TIMES [23], SAVE IDE [211], AITARTOS [164], TART [131]
Controller Synthesis	Uppaal Tiga [35] [35], SynthKro [7], Synthia [118]
Component-Based Development	Uppaal PORT [144], SAVE IDE [211]
Model Minimization	minim [226], VInTiMe [5]
Mixed Reality Language Development	MIRELA Framework [104]
Web Related Development	XAL [76], WST [75]
Parametric Analysis and Verification	TReX [25], IMITATOR [24], VerICS, HyTech [135], RED [235]
Probabilistic Analysis and Verification	Uppaal PRO, mcpta [133], PRISM 4:0 [165], Fortuna [52]
Resource Analysis and Verification	Uppaal Cora [39], TIMES [23], CATS [160], Fortuna [52], Priced-Timed Maude [42]
Other Analyses and Verification	Uppaal [36], TASM [200], McAiT [186], Kronos [98], Open-Kronos [227], TAXYS [88], RTSpin [224], IF [69], CMC [175], MCTA [163], Rabbit [53], VerICS, RED [235], TEMPO [214], DREAM [188], VInTiMe [5]

Table 2.7: Major purposes of timed automata-based or related tools

From the tool descriptions of Tables 2.3–2.6, we can categorize these forty tools into eleven classes depending on their major functionality: *black-box testing and related*, *code synthesis and scheduling*, *controller synthesis*, *component-based development*, *model minimization*, *mixed reality language development*, *web related development*, *parametric analysis and verification*, *probabilistic analysis and verification*, *resource analysis and verification*, and *other analyses and verification*. Table 2.7 presents all these timed automata-based (or related) tools and their classification. Table 2.7 clearly indicate that the majority of timed automata-based tools is used for real-time analysis and verification purposes and that tools are also beginning to be used for other purposes. The Uppaal group, Verimag, LSV, and Saarland University usually develop separate tools for each major functionality. On the other hand, developers of RED [235] and VerICS [157] add major functionality—such as parametric analysis, higher level model analysis—with these tools in each new release. It would be worthwhile to do a survey which would present detailed comparisons depending on functionality and performance of all the timed automata-based tools of the same class.

2.6 Discussion

This survey took more than a year to finish. It helped us to pick the right methods, variants, and tools that arose during the thesis research. We hope this work brings similar awareness to new researchers in timed automata-based research. This chapter, additionally, presents the background knowledge required for the subsequent chapters.

We propose *timed process automata* in Chapter 4. Resource constraints may not permit a hierarchical system to activate all of its components at the same time. Such resource constraints of can be modeled using timed process automata thus they are a variant of timed automata with resources. Task automata can model only two layers of hierarchy

and only closed systems. Our proposed variant can model any numbers of hierarchies and can model both closed and open systems. Moreover, timed process automata can model private communication among components. Timed process automata are a member of the class of timed automata with succinctness because they hide many design details from the designers to achieve succinctness. Timed process automata are also timed game automata [190, 100, 4, 95] because the new variant uses timed games for analysis.

This survey did not discuss real-time temporal logics, real-time formal verification, and real-time controller synthesis because these topics are mostly related to real-time formal models in general instead of being specific to timed automata. There are many surveys [2, 81, 67, 189, 192, 199, 225, 236, 240] which may be attractive for readers who are interested in these real-time formal methods.

Chapter 3

Synthesis of a Reconfiguration Service for Mixed-Criticality Multi-Core Systems

We synthesize task-level reconfiguration services to ensure fault-tolerance of a mixed-criticality automotive system that consists of an asymmetric multi-core processor (AMP). The system has a fault-intolerant AMP scheduler. We augment the existing scheduler with supplementary reconfiguration services, which we synthesize. The services assure the periodic executions of all the critical tasks in the presence of faults from a fault model.

We use timed games at synthesis-time and lookup tables at runtime to provide task-level reconfiguration, a cost-effective fault-tolerance technique, for mixed-criticality multi-core systems. System-level requirements for embedded, real-time software in many domains (such as automotive) have enough flexibility to reallocate tasks from one processing core to another. A task-level reconfiguration technique reduces the number of redundant cores that are used only to provide fault-tolerance by reallocating the loads of the failed cores to the non-redundant operational cores. Reduction in the amount of expensive hardware gives task-level reconfiguration a hope to be a dominant fault-tolerance technique in the automotive industry, where cost-efficiency and fault-tolerance are both crucial issues. Since

this economical technique can handle tasks with different levels of criticality, one of its prospective application sectors is next-generation automotive systems, most of which are expected to be mixed-criticality multi-core systems.

Formal methods have been used for the development of fault-tolerant real-time systems. However, in the industry, fault-tolerance problems are typically designed, analyzed, and solved using classical control theory [241, 146]. Timed game theory [190, 29, 100], a dense-time automata-based game theory, is almost unexplored in the industry. The use of timed game theory to solve industrial problems is attractive because of automated controller synthesis, visual modeling, and dense-time formal analysis. Nevertheless, applying timed game theory to solve industrial problems is challenging because of its high computational complexity.

We use timed games to synthesize the embedded controllers of the reconfiguration services. Our approach guarantees fault-tolerance up to a certain maximal number of concurrent faults after inserting the services into the system. Such reliable and accurate information is very helpful to build mixed-criticality systems cost effectively. Intellectual property regulations do not allow us to present the case study on the systems of our industry partner. Instead we demonstrate the synthesis process using a small system, which is complex enough to show the essence of the problem and our approach, yet simple enough to allow a compact and comprehensible presentation.

Methodology In Section 3.2 we propose a service-based task-level reconfiguration technique to guarantee fault-tolerance of mixed-criticality multi-core systems. Timed games are used to synthesize controllers that select safe operational cores to reallocate the periodic executions of critical tasks from failed cores. Lookup tables are used at runtime to suspend and reinstate the periodic executions of non-critical tasks to ensure that operational cores

have enough free processing capacity for the executions of the newly reallocated critical tasks. We synthesize the reconfiguration services in six steps:

Section 3.3 Identification of modeling abstractions and required system parameters to construct a scalable timed game model of task-level reconfiguration services for fault tolerance to synthesize them.

Sections 3.3.1–3.3.3 Construction of a timed game model where unsafe locations are reachable if and only if a core exceeds its load capacity.

Section 3.4.1 Analysis of the model for the existence of a central controller that ensures no unsafe location is reachable; binary search for the maximal value of the concurrent-failures–limit for which such a controller exists; and automated synthesis of the controller of the maximal concurrent-failures–limit.

Section 3.4.2 Synthesis of the reconfiguration services by distributing the synthesized central controller and combining the abstracted elements of Section 3.3.

Section 3.5 Leverage scalability of the whole process for industrial systems using aggressive abstractions.

Section 3.6 Generalization of the synthesis process to apply in other multi-core systems, such as for symmetric multi-core processing (SMP) systems.

We use Uppaal Tiga [35]—a solver for timed games—to model, analyze, and synthesize. The methodology, however, can be applied using any solvers for timed games, such as Synthia [118].

3.1 Related Work

A real-time control problem can be viewed as a two-player timed game [190, 29, 100] between the controller and the environment, where the controller aims to find a strategy to guarantee that the system will satisfy a given property, no matter what the environment does [94]. An example of such reformulation is to find a strategy for the controller (or a reconfiguration service) to prevent the system from becoming unstable in the presence of the faults of the fault model. No prior work considered timed games to synthesize controllers for mixed-criticality fault-tolerant multi-core systems.

We use dense-time model-based approach to synthesize the services because dense-time models can capture fault occurrences and other uncontrollable behaviors at dense-time, not only uncontrollable behaviors at discrete time. Timed automata [15, 230]—finite automata with dense-time clocks, clock constraints, and clock resets—are a prominent class of formal models to analyze safety and reachability properties of real-time systems. Clocks, clock constraints, and clock resets are used to express timing behaviors in timed automata, which have been used for many purposes [230] including for fault diagnosis [221, 63, 238], analyzing multi-core systems [187, 91], task models [121], and analyzing mixed-criticality systems [213].

A timed I/O automaton [156, 95] is a timed automaton having an input alphabet and a set of uncontrollable transitions along with a regular (output) alphabet and a set of regular (controllable) transitions. The controller plays controllable transitions and the environment plays uncontrollable transitions; thus timed I/O automata are a natural model for real-time games. Uppaal Tiga [35] is a well-known timed games-based tool that uses timed I/O automata for modeling.

3.2 Task-Level Reconfiguration Technique

We introduce a service-based task-level reconfiguration technique to assure fault-tolerance of mixed-criticality multi-core systems.

3.2.1 Systems

We consider a class of multi-core systems having asymmetric processing cores. Different asymmetric cores may exhibit different performance for the same task. The systems under consideration are mixed-criticality systems, because they execute both critical tasks and non-critical tasks with two different priorities.

Definition 3. *A mixed-criticality system, of our consideration, consists of*

- *N asymmetric processing cores:* core_1, core_2, $\cdots$, core_N
- *M tasks:* task_1, task_2, $\cdots$, task_M
- *P critical tasks, where* $P < M$
- *A fault-intolerant criticality-unaware AMP scheduler with a static allocation of tasks*
- $\texttt{load}(\mathsf{task}_i, \mathsf{core}_j)$ *is a function mapping each task-core pair to the worst-case load that the task generates on the core, represented as a number* $\{0, 1, \cdots, 100\} \cup \{+\infty\}$*, where* $+\infty$ *represents incompatibility between the core and the task.*
- *Function* $\texttt{primary}(\mathsf{task}_i)$ *maps* task_i *to the core on which the task runs in the initial system-state*
- *Predicate* $\texttt{critical}(\mathsf{task}_i)$ *holds only for critical tasks*

- *Each task is executed periodically. Tasks always terminate within the prescribed periods. Each task is described as a timed I/O automaton. These automata do not communicate*[1]*. Every task can be killed (and resumed) in any of its states by a reconfiguration technique.*

- *Fault Model: The system is fault-free in its initial system-state. In the other system-states, the system might suffer three types of faults: safety violations by tasks, permanent core failures, and temporary core failures. Critical tasks are assumed not to breach any safety constraints. Non-critical tasks may violate safety constraints. Every core of the system may fail. However, all cores of a system cannot simultaneously be in their failed states. The maximal number of cores that can fail concurrently is restricted by CFL, concurrent-failures–limit. No limit is imposed on the total number of fault occurrences in a run.*

Given a mixed-criticality system of Definition 3, we want to obtain a task allocation policy that is able to cope with the failures admitted by the fault model. We will synthesize distributed reconfiguration services that assure uninterrupted executions of all the critical tasks. Section 3.2.2 explains how the reconfiguration technique is expected to work using an example.

3.2.2 Task-Level Reconfiguration Service

We propose a service-based reconfiguration technique for the fault-tolerance of mixed-criticality systems, where the system has a task-level reconfiguration service for each core.

[1]More generally, the communication can be abstracted by suitable understanding of worst and best case execution times, and terminations are independent of communication

The services manage critical tasks differently from non-critical tasks. Consider, for instance, a simple mixed-criticality AMP system system_1, one of the systems that are described in Section 3.2.1. System system_1 executes six periodic tasks S, W, D, N_1, N_2, and N_3. Only three tasks S, W, and D are the critical tasks, where in an execution S records exactly one update of a speedometer, and W (respectively, D) records at most one update of a wiper (resp., door). The system has three cores core_1, core_2, and core_3, which are asymmetric but each core is able to execute all six tasks.

	core_1	core_2	core_3
s_1	core_1: operational S : primary N_1: safe	core_2: operational W: primary N_2: safe	core_3: operational D: primary N_3: safe
s_2	core_1: operational S: primary N_1: safe	core_2: failed W: primary N_2: safe	core_3: operational D: primary N_3: safe
s_3	core_1: operational S: primary N_1: safe	core_2: failed	core_3: operational D: primary W: backup
s_4	core_1: operational S: primary N_1: unsafe	core_2: failed	core_3: operational D: primary W: backup
s_5	core_1: operational S: primary	core_2: failed	core_3: operational D: primary W: backup
s_6	core_1: operational S: primary	core_2: operational	core_3: operational D: primary W: backup
s_7	core_1: operational S: primary	core_2: operational W: primary N_2: safe	core_3: operational D: primary N_3: safe

Figure 3.1: Sample trace of system_1 with reconfiguration

Figure 3.1 presents a trace of a desirable behavior of system_1 in the presence of different

faults after inserting the reconfiguration services; the figure omits suspended non-critical tasks to avoid clutter. At any given time, the periodic execution of a task can be assigned to at most one operational core. A task is assigned to its primary core in the initial system-state, where a core is responsible to execute only its primary tasks. For instance, core_1 is the primary core of task S, and S is a primary task of core_1 in Figure 3.1. We call a non-primary core a backup core of a critical task when that core can execute that task; similarly, a task is a backup task of its backup core. Whenever a core fails, the services assign the critical tasks that were previously assigned to that failed core to the operational cores. The services may kill and suspend temporarily one or more non-critical tasks on the operational cores during a reallocation process to ensure enough processing capacity for the reallocated critical tasks. In Figure 3.1, core core_2 fails in system-state s_2; in the next system-state, the periodic execution of critical task W is assigned to a backup core core_3 and the periodic execution of non-critical task $\mathtt{N}_3$ is suspended temporarily on core_3 to have enough processing capacity for W. A critical task is allowed to execute further on a backup core only if the primary core is in a failed state. The services kill a critical task on a backup core (if that task is initialized or released) and cancels the assignment of that task on that backup core, whenever the primary core recovers from a temporary failure. As an example, core core_2 recovers from a temporary failure in system-state s_6, and after that only core_2 is assigned to perform critical task W. The services reinstate a suspended non-critical task as soon as enough processing capacity for that task is regained due to the recovery of a core from a temporary failure; for example, the periodic execution of non-critical task $\mathtt{N}_3$ is reinstated in system-state s_7. The services permanently suspend a non-critical task when it performs some harmful activities, such as illegal memory access. For instance, non-critical task $\mathtt{N}_1$ performs some harmful activities in system-state s_4, and the task is permanently

suspended in system-state s_5.

Problem Statement Given a mixed-criticality system as specified in Definition 3, the problem is to synthesize a reconfiguration service $\mathsf{service}_i$ for each core core_i such that $\mathsf{service}_i$: (i) reacts whenever any other core fails or a core recovers (including core_i), or a non-critical task violates a safety constraint on core_i; (ii) at that time $\mathsf{service}_i$ may kill, resume, and suspend any task running on core_i; and (iii) as long as core_i is in a failure state, none of its tasks nor $\mathsf{service}_i$ executes. All reconfiguration services of a system together satisfy a property that at all times critical tasks are allocated to operating cores as long as the CFL limit is observed, and any non-critical task that has violated a safety constraint is suspended from execution indefinitely.

3.3 Modeling

We construct a timed game model of the system in a way that an unsafe location becomes reachable when a core exceeds its processing capacity. The model explicitly or implicitly captures the behaviors of the scheduler, the reconfiguration services, the cores, and the tasks.

To reduce complexity: (i) we model only a single (central) reconfiguration service for the whole system, instead of one service per core; (ii) we assume that every non-idle state of a task requires the worst-case core load of the task on the current core; and (iii) we abstract away the non-critical tasks. These three assumptions do not prevent synthesis of a distributed reconfiguration service per core, which will be shown in Section 3.4. Our model depends on four system parameters: (i) the release period of each task (constants `pS`, `pW`, `pD`); (ii) the worst-case load of each task on each core, in percent of the processing

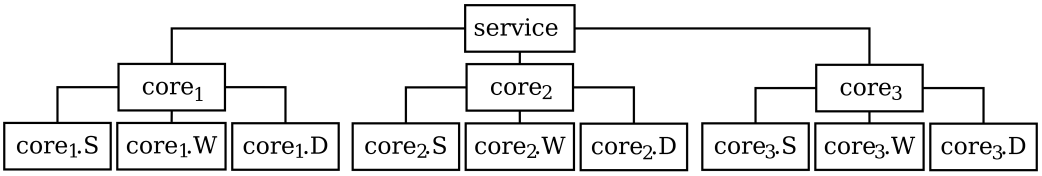


Figure 3.2: Architecture of system_1 after adapting abstractions of Section 3.3

capacity of that core (constants `lS1`, `lW1`, `lD1`, `lS2`, `lW2`, `lD2`, `lS3`, `lW3`, `lD3`); (iii) the worst-case execution time (WCET) of each task on all cores (constants `wS`, `wW`, `wD`); and (iv) the best-case execution time (BCET) of each task on all cores (constants `bS`, `bW`, `bD`).

Now we illustrate the design process by constructing a concrete model of mixed-criticality AMP system system_1. The main design principle behind this model is to describe each component of the system in detail as a timed I/O automaton then obtain an intuitive concrete model by composing all the components using parallel composition [95]. The concrete model has 13 timed I/O automata, which follow five different templates. In general, the concrete model has at most $(N \times K) + N + 1$ timed I/O automata and $K + 2$ templates, where N is the number of total cores, K is the number of total critical tasks, constant 1 automaton for the central service, constant 1 template for cores, and constant 1 template for the central service.

Each automaton of the concrete model represents exactly one rectangle of Figure 3.2. The automata synchronize using both actions and global variables. The model does not have any local variables and constants. A task automaton models initialization, killing, resumption, termination, and state information of a task on a specific core; for example, task automaton core_1.`S` in the bottom of Figure 3.4 represents the activities of task `S` on core_1. A core may fail only if the fault model allows it to fail. A core automaton models initializations and terminations of tasks on a core along with failures of the core and safety

violations; for instance, core automaton core_1 in the middle of Figure 3.4 represents the activities of core core_1. The `service` automaton in the top of Figure 3.4 models reallocations of the critical tasks when a core fails or recovers. In the model a failed core may recover at any time. All automata of the model are presented in the appendix and in a technical report [232].

The automata modeling cores follow the same template. For instance, automaton core_1 uses action `kS1` to model the killing of task `S` on core_1 (edge 16 in Figure 3.4), `kW1` to model the killing of task `W` on core_1 (edge 17), `kD1` to model the killing of task `D` on core_1 (edge 18), and global variable `L1` to record the current worst-case load on core_1 (edges 9–14,16–18); similarly, automaton core_2 uses action `kS2` to model the killing of `S` on core_2 and global variable `L2` to record the current worst-case load on core_2. The automata modeling the same task—but on different cores—follow the same template.

3.3.1 Task Automata

A task automaton represents two types of activities of a task on a core:

Task Life-Cycle Activities (edges 1–5) Every task can be initialized, killed, and resumed by performing controllable actions. Task terminations are modeled using uncontrollable actions because neither the scheduler nor the reconfiguration services can control the exact termination period of a task. The models are built in Uppaal Tiga [35], which displays controllable transitions as solid arrows (edges 1–4), and uncontrollable transitions as dashed arrows (edges 5–8). The duration between an initialization and the immediate termination of a task encompasses one complete execution of that task. A task can be killed and then resumed arbitrarily many times in a single execution. Initialization, killing, resumption, and termination of task `S` on core_1 is modeled by performing actions `iS1` (edge 1), `kS1`

(edges 3–4), rS1 (edge 2), and tS1 (edge 5), respectively. Every task automaton has at least two locations: Idle and Active. The task is either killed or yet to be initialized in location Idle. Every non-idle location has an invariant to force the task to terminate within the WCET; for instance, an automaton modeling task S has invariant x≤wS to force termination, where global clock x records the amount of time passed since the last initialization of S and global constant wS stores the WCET of S. Similarly, global constant bS stores the BCET of task S. Hence, clock guard x≥bS prevents task S to terminate before the BCET (edge 5).

Task Specific Activities (edges 6–8) Task S records exactly one update of a digital speedometer (modeled as global variable vS) in an execution: vS represents the speed in multiples of five varying from zero to hundred. Boolean variable uS is 1 if and only if the speedometer is updated in the current execution of S.

Task automata core_1.W and core_1.D in the concrete model is presented in the appendix and in the technical report [232]. The automata model task life-cycle activities and task specific activities of tasks W and D.

In general, a timed I/O automaton representing a task (in Definition 3) is transformed (like Figure 3.3) to a corresponding task automaton by adding controllable transitions to capture periodic (or cyclic) initialization of the task (by the scheduler), killing by reconfiguration services at all internal states (of interest), and resumption by reconfiguration services at all internal states (of interest).

3.3.2 Core Automata

A core automaton in the concrete model models two types of activities:

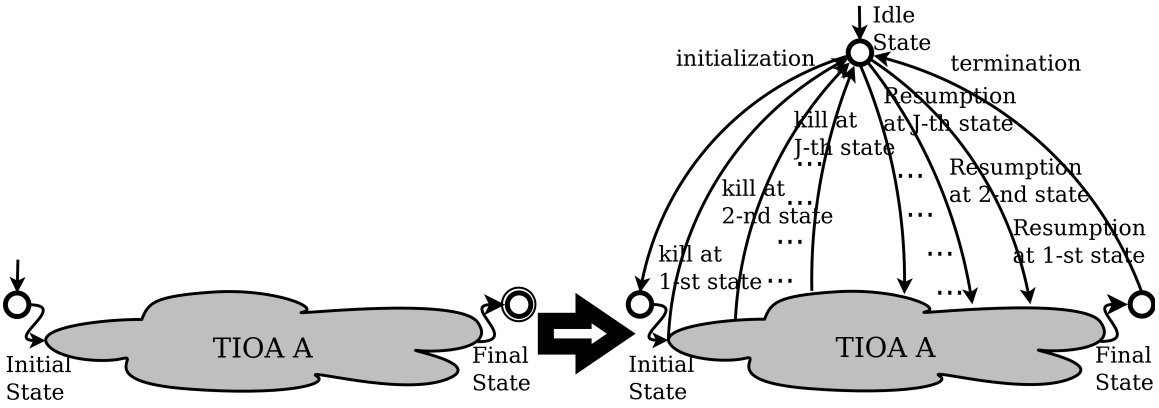


Figure 3.3: Transformation of a timed I/O automaton representing a task (in Definition 3) to a task automaton: from single to periodic executions with kill and resumption at all internal states

Operation-Time Activities (edges 9–14) A core automaton periodically initializes a task at its release period if the corresponding core is assigned to execute that task. Task terminates voluntarily after completing its assigned functions. A task between its initialization and termination occupies a portion of the resources. When a task terminates (resp., is initialized) on a core, the corresponding core automaton decreases (resp., increases) a variable modeling the current worst-case load. In location `Main`, task `S` is initialized by performing action `iS1` (edges 9) if `S` is assigned to core_1 (`aS==1`), and `S` is not initialized yet (`iS==0`), and clock `x` hits the value of the release period of `S` (`x==pS`). Automaton core_1 (edge 14) receives action `tS1` from task automaton core_1.`S` (edge 5) whenever `S` terminates its execution on this core. Function `terminate(S,1)` decreases (resp., `initialize(S,1)` increases) variable `L1`, modeling the worst-case load on core_1, by constant `lS1`, the worst-case load of task `S` on core core_1, and resets Boolean variable `iS` to 0 (resp., 1), that means task `S` terminates (resp., is initialized).

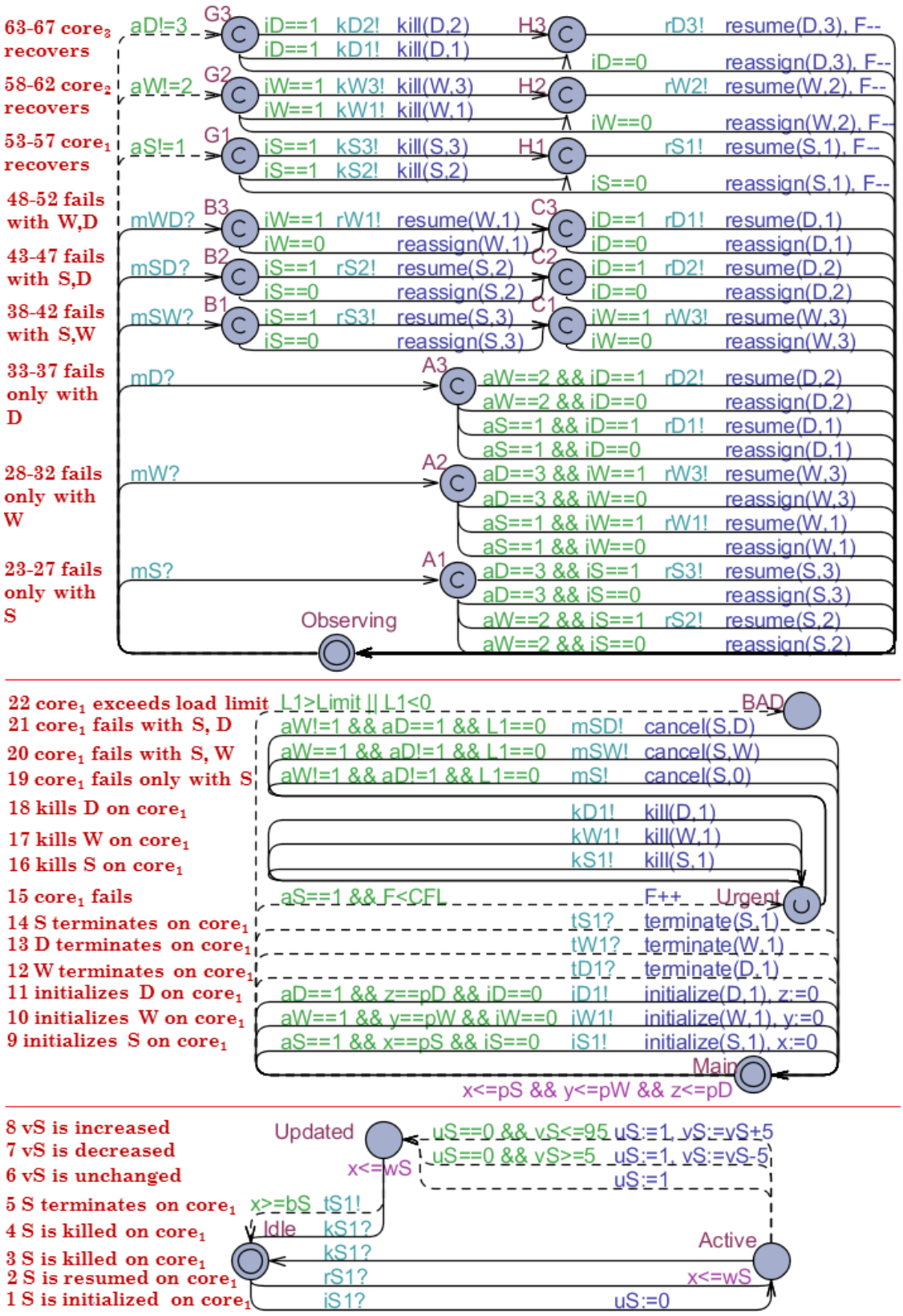


Figure 3.4: Automata $\mathsf{core}_1.\mathsf{S}$ (in the bottom), core_1 (in the middle), `service` (in the top)

Failure-Time Activities (edges 15–22) Core automaton core_1 models failures of the core by traversing an uncontrollable edge. In order to reflect our assumption on the concurrent failure limit, this edge is only admitted if the number of currently failed cores (`F`) is less than CFL (`CFL`). Location `Urgent` is reached from `Main` whenever core_1 fails. `Urgent` is one of the urgent locations[2], denoted as Ⓤ in Uppaal Tiga syntax, that means the automaton cannot spend time in this location (edges 15–21). When the core fails, the automaton instantaneously kills all tasks currently released by it—to simulate that no task can continue to run on a failed core (edges 16–18). Then the automaton instantaneously performs specific actions to broadcast a message containing the list of currently assigned tasks to that failed core: performs action `mS` if `S` is the only assigned task; action `mSW` if `S` and `W` are the only assigned tasks; and action `mSD` if `S` and `D` are the only assigned tasks (edges 19–21). To note that only task `W` or task `D` or both cannot be assigned to core core_1 without task `S` because a task (`S`) must be assigned to its operational primary core (core_1). At runtime, the reconfiguration services use a distributed monitoring system to identify these (task) assignments because no failed core can broadcast a message. An unsafe location `BAD` becomes reachable when the current worst-case load on core_1 exceeds the load limit of core_1 because of the failure of some other core(s) (edge 22). This prevents the synthesis algorithm from producing a strategy that would require illegal loads.

In general, a core automaton in Figure 3.5 is constructed for a core, which is compatible with K number of critical tasks of the system. The automaton has three locations: `Main`, `Urgent`, and `BAD`. The initial location `Main` has K controllable (resp., K uncontrollable) self-loops to simulate initializations (resp., terminations) of K critical tasks on the core. Location `Urgent` is reached from `Main` when the core fails. At `Urgent` all the active tasks

[2]Semantically, urgent locations are equivalent to: adding an extra clock, x, that is reset $x := 0$ on every incoming edge, and adding an invariant $x <= 0$ to the location.

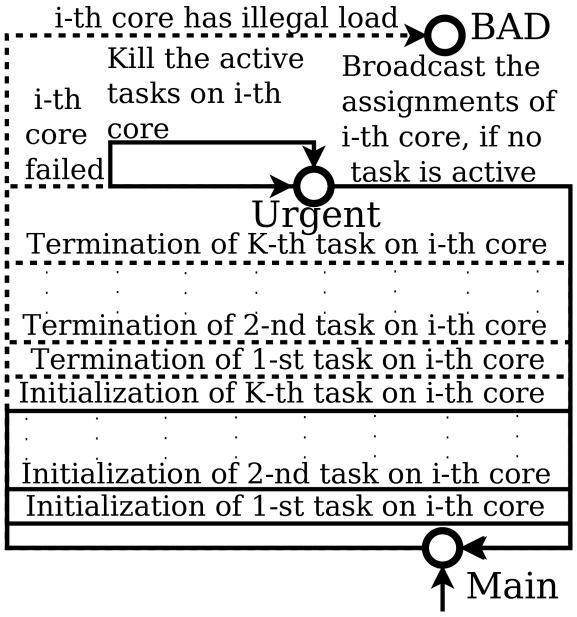


Figure 3.5: A core automaton in general

are killed instantaneously by traversing maximum K self-loops. After that `Main` is reached instantaneously by broadcasting the assigned critical tasks to the core. There are 2^{K-J} edges to broadcast all combinations of assigned tasks to the core, where J is the number of primary critical tasks of the core. However, the number of compatible cores for a task (resp., K) in an AMP system is typically low. The environment may take the game to location `BAD` when the respective core has more load than its load limit.

3.3.3 Service Automaton

A `service` automaton spends most of its time in observing states waiting for a fault to occur (or for a core recovery from a temporary failure). The automaton reallocates a task in two steps: (i) assigns the periodic execution of the task to a suitable operational core, and (ii) resumes the task on the assigned core if the task was initialized before the reallocation.

Other than `Observing` all locations are committed[3], denoted as Ⓒ in Uppaal Tiga syntax. They model states when reconfiguration decisions are taken, which are expected to be instantaneous and get precedence even over the urgent transitions of the other automata. Activities of the automaton can be divided into three groups described in the following.

Handling a Primary Core Failure (edges 23–37) Recall the invariant that an operating core is always assigned to execute its primary tasks, so in system_1 when a core (say core_1) is assigned to execute only one task then it must be a primary task (`S`). In the model a failure message is broadcast using an action (e.g., `mS`, `mSW`, and `mWD`) linked to the currently assigned tasks of the failed core, instead of the name of the core. Therefore, whenever a core failing with only assignment of the periodic execution of task `S` (or action `mS` is performed) then core_1, the primary core of `S`, must be that failed core. At that point, task `S` is reallocated to either core_2 or core_3. For example, location `A1` is reached from location `Observing` when core_1 fails (edges 23–27); in `A1` the target is reallocating `S`, the primary task of core_1, to core core_2 (bottom two outgoing edges) or to core core_3 (top two outgoing edges). Details of reallocation depending on whether the task was initialized (and needs to be reassigned then resumed) or was yet to be initialized (and just needs to be reassigned). For instance, to reallocate task `S` to core_2, location `Observing` is reentered from `A1` by: (i) assigning the periodic execution of `S` to core_2 (`aS:=2`) if core_2 is operational (`aW==2`) and `S` was yet to be initialized (`iS==0`), or (ii) assigning the periodic execution of `S` to core_2 and resuming `S` on the core (by performing `rS2`) if core_2 is operational and `S` was initialized (`iS==1`).

[3] A commited location is the same as a urgent location but after reaching a committed location the next transition must involve an outgoing edge of at least one of the committed locations of the network.

Handling a Backup Core Failure (edges 38–52) In our example, when a core is assigned to execute two critical tasks then one of them must be a backup task of that core; hence, after such a failure at least two cores concurrently are in their failed states. The fault model does not allow all cores to fail concurrently. For instance, core_1 must be operational when core_2 and core_3 are in their failed states; and the executions of tasks `W` and `D` have to be assigned to core_1. Location `B1` is reached from `Observing` when a core fails that is responsible to execute both `W` and `D` or when action `mWD` is received (edges 48–52). Location `C1` is reached from `B1` by assigning the periodic execution of `W` to the only operational core core_1 and resuming `W`, if necessary (`iW==1`). Then `Observing` is reached by assigning the periodic execution of `D` to core_1 and resuming `D`, if necessary (`iD==1`).

Handling a Primary Core Recovery (edges 53–67) The periodic execution of a task must be assigned to its primary core when it is operational. Therefore, a task must be reallocated from a backup core to the primary core whenever it recovers from a temporary failure. The periodic execution of task `S` can be assigned to a backup core (`aS!=1`) only if its primary core core_1 is in a failed state. Location `G1` is reached from location `Observing` when core_1 recovers from a failure (edges 53–57). In `G1` the controller has two main choices depending on the initialization condition of the task: `S` is yet to be initialized and needs to be only reassigned to its primary core (the bottom outgoing edge); and `S` is initialized on a backup core and needs to be killed (the top two outgoing edges) then to be resumed on the primary core (the outgoing edge from location `H1`).

In general, the `service` automaton of Figure 3.6 remains in observing states unless a core fails or recovers. Reconfiguration services need at least one operational core to run tasks. In the worst case when a failure occurs, CFL-1 cores are in their failed states (because if CFL cores had already failed, then no further failure can happen before some of the cores

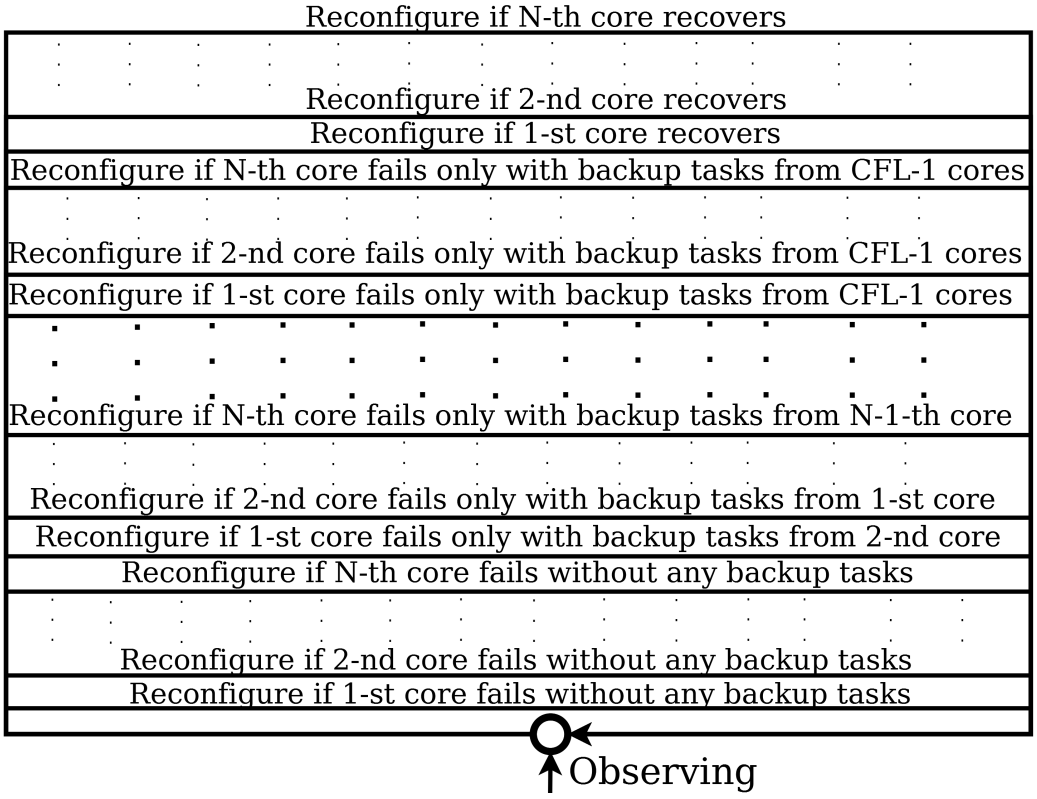


Figure 3.6: A service automaton in general

recover). When a task is failing as the last failure admitted by the fault model, there are $2^{\text{CFL}-1}$ possible subsets of cores from which its currently running task might have been migrated (so they are backup tasks). Therefore in total $N \times 2^{\text{CFL}-1}$ edges are used to match these situations, where the total number of cores is N. The construction is exponential in CFL, however usually CFL is much smaller than N. Similarly, the internal edges of core recoveries depend on CFL and task-core compatibility ($\texttt{load}(\mathsf{task}_i, \mathsf{core}_j)$) relationships.

3.4 Synthesis

We synthesize reconfiguration services in three sequential steps:

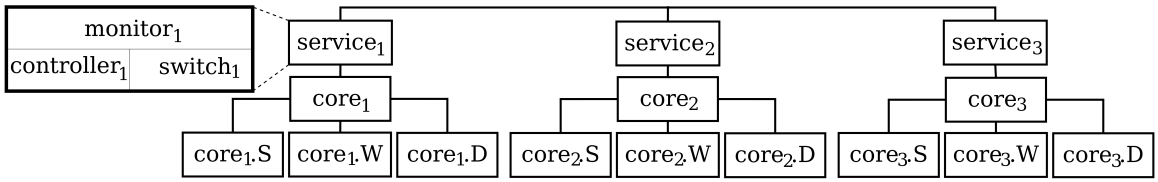


Figure 3.7: Architecture of system_1 at runtime

1. Generate a central controller for critical tasks,

2. Construct a distributed controller for each core by exclusively distributing the central controller, and

3. Synthesize a reconfiguration service for each core by adding its distributed controller with a constructed monitor to broadcast its health messages and a constructed switch to suspend and reinstate its non-critical tasks

A reconfiguration service runs on a core, which can fail. Hence, fault tolerance cannot be achieved using only one central reconfiguration service. We propose for each core to execute its own reconfiguration service that has three components: a distributed controller to reallocate critical tasks, a monitoring system to observe the system's conditions, and a edge to cancel and reinstate the periodic execution of non-critical tasks. All the distributed controllers of a system differ from each other—but complement each other in a way that they all together work similarly with a central controller, which is synthesized by analyzing the timed game model of Section 3.3. Figure 3.7 presents the architecture of system_1 with the reconfiguration services at runtime.

3.4.1 Central Controller Synthesis

We perform a controller synthesis for the monolithic model of Section 3.3 against a safety objective which states that there is a strategy to always avoid locations $\texttt{core}_1\texttt{.BAD}$, $\texttt{core}_2\texttt{.BAD}$, and $\texttt{core}_3\texttt{.BAD}$. If the property holds, the strategy—which is our central controller—is automatically synthesized by a timed game solver.

In order to obtain the most fault-tolerant controller possible, we synthesize it for the maximal concurrent-failures–limit (MCFL), the maximal value of CFL for which such a controller still exists. We use binary search to find MCFL. If MCFL is zero, no safe controller exists. The higher MCFL implies the better fault-tolerance by the reconfiguration services. The value of MCFL is strictly bounded by the total number of processing cores. Consider, for instance, configuration C1 in Table 3.1[4] where the release period, the WCET, the BCET of every task is 10, 5, and 4 time units, respectively; the worst-case load of tasks S, W, and D on core_1 (resp., core_2, core_3) are 60% (resp., 10%, 10%), 45% (resp., 80%, 5%), and 5% (resp., 5%, 85%), respectively. Configuration C1 does not have a controller for CFL 2. However, there is a controller for CFL 1. Maximal concurrent-failures–limit for system_1 for configuration C1 is 1 because 1 is the maximal value of CFL for which a controller exists.

3.4.2 Service Synthesis

We synthesize the distributed reconfiguration service of a core by combining its distributed controller with an embedded monitor and an embedded switch.

[4]To show clearer impacts of different modeling aspects on the analysis, we picked some imaginary system configurations instead of some actual system configurations.

Distributed Controller The functions of the central controller are completely and exclusively distributed into separate controllers for each core. A distributed controller is responsible for killing, reassignment, and resumption of critical tasks only on its core. A timed game represents all the possible transitions of the controller. As a result, a timed game may have non-deterministic choices for the controller. For example, in Figure 3.1 the controller has non-deterministic choices at system-state s_4 when only $core_2$ fails and the other two cores are operational (edges 28–32). A strategy removes non-determinism for the controller. By directing the controller to take the correct paths, a strategy plays a crucial role when in the model some paths guarantee satisfaction of a property (say reallocating task `W` to $core_3$ at system-state s_5 in Figure 3.1) and some paths do not (say reallocating `W` to $core_1$). For example, when $core_2$ fails (edges 28–32) a strategy (or the central controller) may say, "`if the system-state fulfills condition` X `then reallocate task W to` $core_3$`, otherwise to` $core_1$"; then the distributed controller of this portion (edges 28–32) for $core_3$ is "`if the system-state fulfills condition` X `then reallocate task W to` $core_3$"; and the distributed controller of this portion (edges 28–32) for $core_1$ is "`if the system-state does not fulfill condition` X `then reallocate task W to` $core_1$". Thus, deriving the distributed controllers from the central controller is a mechanical process and cannot fail.

Monitor The monitor of a reconfiguration service periodically broadcasts health messages of the corresponding core. A health message has three parts: (a) name of its core, (b) currently assigned critical tasks to its core, and (c) currently initialized critical tasks on its core. A monitor periodically also receives health messages—from the other reconfiguration services—and manipulates received messages. It marks a core as a failed core if two consecutive health messages of that core are not received. The monitor identifies a core

recovery when it receives a message from a previously failed core. In the same way, the monitor detects when the scheduler releases a task and when a task terminates on a core.

Switch A reconfiguration service has a static lookup table and a dynamic lookup table. The static lookup table lists the worst-case core load of every critical task (of the system) on this core and of every non-critical task assigned to this core. The dynamic lookup table keeps updated list of the assigned tasks, temporarily suspended non-critical tasks, and permanently suspended non-critical tasks. The controllers reallocate critical tasks from a failed or to a recovered core without considering the existence of non-critical tasks. The switch of a reconfiguration service (of the targeted core) suspends a set of non-critical tasks on its core using the lookup tables when the residual capacity on the core is insufficient to run the newly reallocated task safely. The distributed controllers first take necessary steps related to primary tasks of the recovered core whenever a core recovers. After that the switches reinstate the periodic executions of a set of suspended non-critical tasks on each source core where free processing capacity is revived due to the recovery. The switch permanently suspends a non-critical task when it breaches safety constraints.

3.5 Manual State-Space Reduction

The scalability of our service synthesis process mostly depends on the central controller synthesis as the remaining steps are mechanical and cannot fail. The concrete model has very large state space. For example, configuration C1 in Table 3.1 generates a strategy of size 290,663 KB in 94.20 seconds for this model when CFL is 1, presented in Table 3.2. Moreover, for many configurations the solver runs out of memory during analysis, such as, C3–C5 in Table 3.2. Detailed and monolithic models like the concrete model are easy to

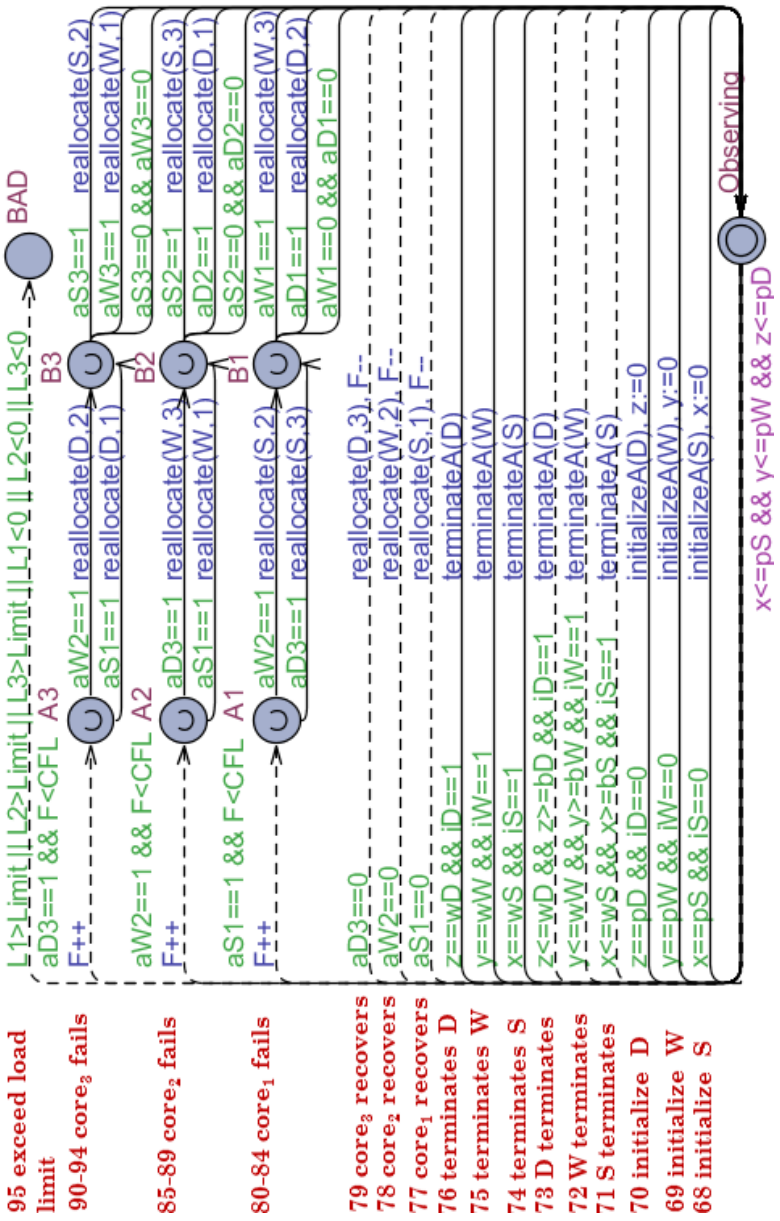


Figure 3.8: The abstract model(comments are on the left)

construct, understand, and present. However, large state spaces make them a poor choice for analysis.

The main purpose of the strategy is to resolve non-determinism among enabled controllable transitions in a way that guarantees satisfaction of the desired property. Hence, one can abstract away every detail from a timed game model that does not contribute to the non-determinism (or to the property). For instance, task specific activities and their non-deterministic updates of the tasks, which do not have any impact on our property, can be removed from a timed game model of system_1. Using such aggressive abstractions we construct the abstract model of system_1. Presented in Figure 3.8, the model has only one automaton.

The abstract model uses all the modeling abstractions and system parameters of Section 3.3. Explicitly it models only task initializations (edges 68–70), task terminations (edges 71–76), core failures (edges 77–79), core recovery (edges 80–94), and safety violations (edge 95). Like task killings and resumptions, task initializations and terminations change the load on a core; thus they play an important role in the required property (or the safety checking). The invariant is used to release or initialize the tasks periodically. While a task termination within the WCET is forced by allowing an additional controllable transition (edges 74–76). Reallocation is a function which combines task killings, reassignments, and resumptions (edges 77–94). The model uses nine Boolean variables `aS1`, `aW1`, `aD1`, `aS2`, `aW2`, `aD2`, `aS3`, `aW3`, and `aD3` to keep track the currently assigned tasks to cores: the value of `aS1` (resp., `aW1`, `aD1`) is 1 when the periodic execution of task `S` (resp., `W`, `D`) is assigned to core core_1, otherwise the value is 0; similarly, `aS2` (resp., `aW2`, `aD2`) is 1 if and only if the periodic execution of task `S` (resp., `W`, `D`) is assigned to core core_2. If both the concrete model and the abstract model use a variable or constant, it is used for the same purpose; for example, variable `iS` in both the models is used to identify when task `S` is initialized.

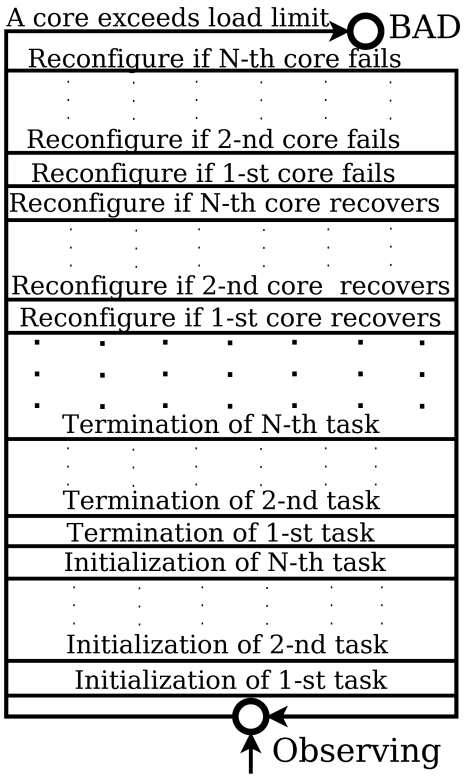


Figure 3.9: The abstract model in general

In general, the abstract model (in Figure 3.9) combines all automata of the concrete model into a single automaton. In both models, one clock per task is used for the execution times, and the worst-case loads have been used. A task can be killed and resumed at any internal states, and the internal control behaviors of a task cannot be effected by other tasks (see Definition 3). Therefore, internal executions of the tasks can be abstracted away by only tracking task assignments along with task initializations—which is actually done in the abstract model. The abstract model hides communications among automata of the concrete model but keeps their effects. For example, in a core automaton (in Figure 3.4 or in Figure 3.5) when the core fails, all the initialized tasks on the core are instantaneously killed by sending kill messages to the corresponding task automata and instantaneously

broadcast the assignments to the service automaton; the abstract model simulates core failures and hides the following two communication steps (but performs necessary changes in the variables). Similarly, task killing and resumption related communications in the concrete model between the service automaton and a task automaton abstracted away in the abstract model. The abstract model also hides obvious details of the concrete model. For example, in service automaton (in Figure 3.4) to reallocate, an initialized task is (reassigned and then) resumed and an uninitialized task is only reassigned; the abstract model (in Figure 3.8) models only task reallocations (and hides the other details of resumptions and reassignments). Therefore, a strategy extracted from the abstract model can be used for the concrete model by augmenting these communications and obvious details.

For the control problem described in this chapter, we constructed four different models: the concrete model as described in Section 3.3, the abstract model as described in this section, the monolithic model, and the compositional model. The last two models are presented in Chapter 4. We analyze these models with many configurations. This section discusses behaviors of the concrete and abstract models for 20 configurations of Table 3.1.

All the analyses and controller syntheses were performed by Uppaal Tiga-0.17 on a PC with an Intel Core i3 CPU at 2.4 GHz, 4 GB of RAM, and running 64-bit Windows 7. We compare the concrete and abstract models with respect to controller synthesis time and the strategy size. Uppaal Tiga(-0.17) generates the same (size of) strategy for the same configuration on the same machine. Controller synthesis time, on the contrary, varies a little for the same configuration on the same machine. Therefore, we synthesize a strategy for every configuration multiple times, and then take the average synthesis time for each configuration.

Experimental results of the concrete and abstract models are presented in Table 3.2.

Con-fig-ura-tion	Period of task			WCET of task			BCET of task			Load on $core_1$ of task			Load on $core_2$ of task			Load on $core_3$ of task		
	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D
C1	10	10	10	5	5	5	4	4	4	60	45	5	10	80	5	10	5	85
C2	10	10	10	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C3	10	15	20	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C4	10	15	20	5	5	5	0	0	0	60	35	5	10	80	5	10	5	85
C5	10	15	20	5	5	5	0	0	0	43	37	7	11	67	19	23	13	59
C6	10	15	20	5	5	5	0	0	0	43	37	59	11	67	39	23	13	59
C7	10	15	20	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C8	10	15	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C9	10	20	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C10	11	19	31	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C11	5	7	11	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C12	5	7	11	5	3	2	0	0	0	33	33	33	33	33	33	33	33	33
C13	5	7	11	5	3	2	5	3	2	33	33	33	33	33	33	33	33	33
C14	10	15	20	5	5	5	5	5	5	33	33	33	33	33	33	33	33	33
C15	10	15	20	5	7	11	5	7	11	33	33	33	33	33	33	33	33	33
C16	10	15	20	5	7	11	0	0	0	33	33	33	33	33	33	33	33	33
C17	10	15	20	7	7	7	7	7	7	33	33	33	33	33	33	33	33	33
C18	10	15	20	5	7	7	5	7	7	33	33	33	33	33	33	33	33	33
C19	10	15	20	7	7	11	7	7	11	33	33	33	33	33	33	33	33	33
C20	10	15	20	9	13	19	9	13	19	33	33	33	33	33	33	33	33	33

Table 3.1: Different configurations: combinations of release period, WCET, and BCET have abstract time units; and loads are in % of the respective core

We have the following six observations from this table:

1. The abstract model improves the scalability dramatically for every configuration of Table 3.1. Other than aggressive abstraction, it encodes the whole model into only one automaton to avoid parallel composition, because parallel composition typically increases the size of the state space rapidly.

2. The larger the difference between WCET and BECT the longer the analysis time, and the sparser the strategy. Consider, for example, configuration C1 versus configuration C2, C7 versus C14, C12 versus C13, and C15 versus C16.

Configurations of Table 3.2	CFL	Comparison			
		concrete model		abstract model	
		time	size	time	size
C1	2	No controller exists			
	1	94.20	290663	0.08	73
C2	2	No controller exists			
	1	115.71	296524	0.11	107
C3	2	No controller exists			
	1	Out of memory		0.14	242
C4	2	Out of memory		0.25	712
	1			0.14	266
C5	2	Out of memory		0.25	712
	1			0.14	266
C6	2	No controller exists			
	1				
C7	2	Out of memory		0.25	712
	1			0.14	266
C8	2	Out of memory		0.15	420
	1			0.11	159
C9	2	Out of memory		0.22	632
	1			0.14	234
C10	2	Out of memory		178.54	40668
	1			73.32	14647
C11	2	Out of memory		4.91	6274
	1			1.65	2277
C12	2	Out of memory		4.07	6272
	1			1.65	2275
C13	2	Out of memory		1.93	3639
	1			0.81	1332
C14	2	Out of memory		0.20	539
	1			0.14	204
C15	2	Out of memory		0.15	431
	1			0.11	164
C16	2	Out of memory		0.24	718
	1			0.14	270
C17	2	Out of memory		0.16	458
	1			0.12	173
C18	2	Out of memory		0.16	485
	1			0.10	184
C19	2	Out of memory		0.14	406
	1			0.10	154
C20	2	Out of memory		0.14	358
	1			0.09	135

Table 3.2: Comparisons of the concrete and abstract models with respect to controller synthesis average time (in seconds) and the strategy size (in kilobytes)

3. The smaller the least common multiples of release periods the smaller state space, the shorter analysis time, and the more compact strategy. Consider, for example, C2 versus C3, C8 versus C9, C9 versus C10, C10 versus C11, and so forth. For configurations C10 and C11, we use three different prime numbers as release times to get large least common multiples of the release times. As a result, for these configurations, we have sparse strategies along with long synthesis times even for the abstract model. One should check the least common multiples of the release times of a system before trying to (model and) synthesize controller for it using timed games. Unfortunately, timed games-based analytical tools are currently not mature enough to synthesize scheduler for practical systems having large least common multiples of the release times.

4. On the other hand, the least common multiples of the execution times have no visible impact on the analysis time or the size of the strategy; (for instance, C14 versus C15, C15 versus C17, C17 versus C18, C18 versus C19, C19 versus C20, and so forth).

5. Variations in the least common denominator of non-clock variables, such as different loads, do not have any significant impact on the analysis; (for example, C4 versus C5 and C5 versus C7).

6. Uppaal Tiga takes less time and generates a smaller strategy for a higher value for CFL; (for instance, configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20.)

Probably, the first observation is the most important one, which states that the scalability improves in the abstract model. Table 4.2 in Chapter 4 shows that the above observations are also true for the monolithic model and the compositional model.

The MCFL of system system_1 depends on its configuration and model:

- For the concrete model
  - The MCFL is unknown for configurations C3, C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20;
  - The MCFL is 1 for configurations C1 and C2; and
  - The MCFL is 0 for configurations C6.
- For the abstract model
  - The MCFL is 2 for configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20;
  - The MCFL is 1 for configurations C1, C2, and C3; and
  - The MCFL is 0 for configurations C6.

3.6 Discussion

A mixed-criticality system has two or more criticality-levels, where each level may have its own control objectives. Usually current industrial safety standards identify up to five criticality-levels in a system. For example, automotive functional safety standard ISO 26262 divides an automotive system into five criticality-levels: one non-safety criticality-level and four safety criticality-levels (ASIL A, ASIL B, ASIL C, and ASIL D). The success of a higher criticality unit cannot depend on its lower criticality units but the success of a lower criticality unit may depend on higher criticality units. For example, if the periodic execution of critical task S depends on the execution of non-critical task $\mathtt{N}_1$ then $\mathtt{N}_1$ must be a critical task.

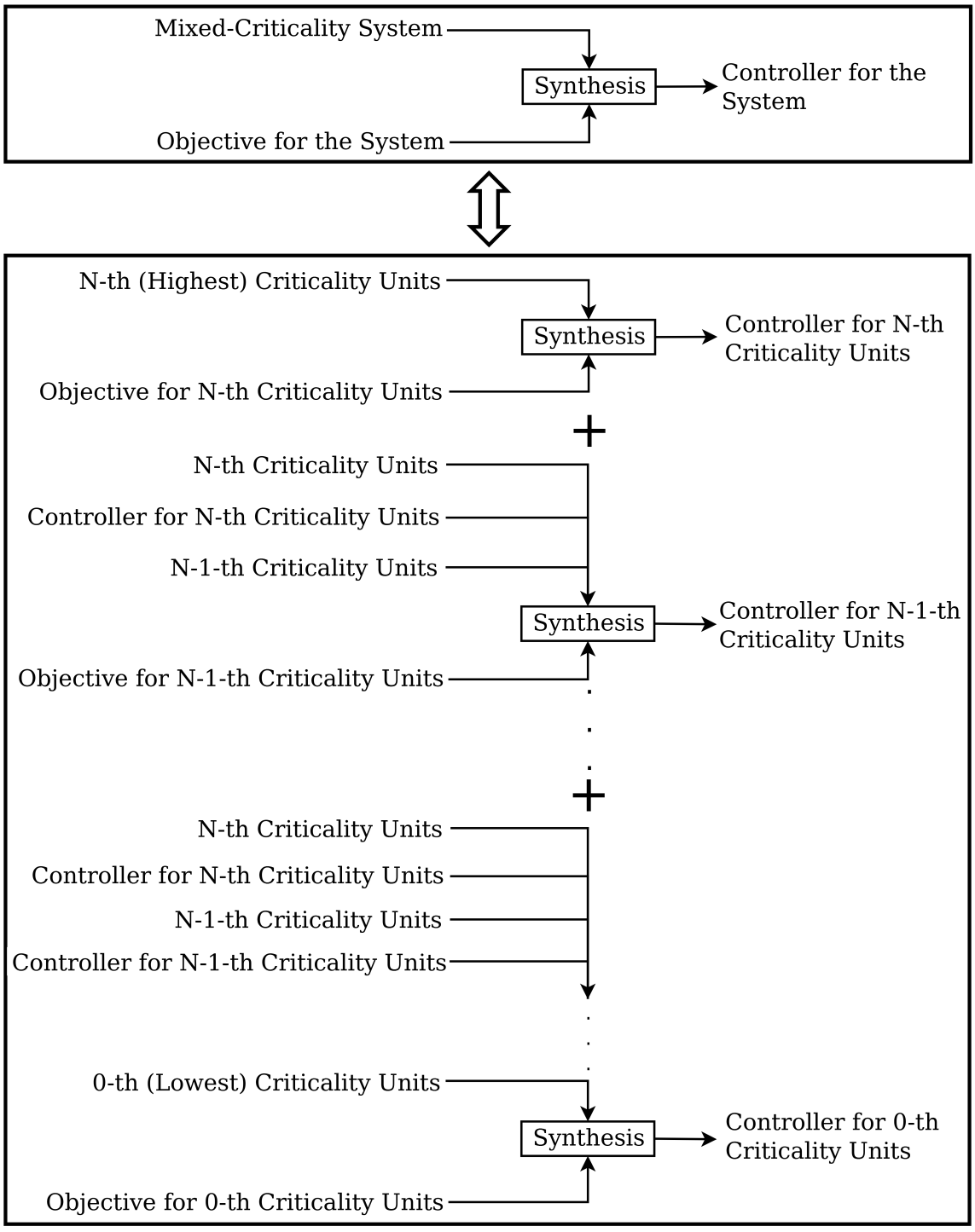


Figure 3.10: Divide and serially conquer strategy for the synthesis of mixed-criticality controllers

Until now this chapter only presented an application of our developed *divide and serially conquer strategy* for a specific and simple case-study, where the control problem for critical tasks is solved first and then the control problem for non-critical tasks is solved. Now we briefly discuss the strategy for the synthesis of general mixed-criticality controllers. Figure 3.10 shows our proposed and applied divide and serially conquer strategy. It divides a mixed-criticality controller synthesis problem into multiple controller synthesis problems. Moreover, the strategy suggests the synthesis of a controller of a higher criticality-level before solving control problems of a lower criticality-level.

In this chapter we solve "synthesize a controller to ensure that the critical tasks of system system_1 execute uninterruptedly and the non-critical tasks execute according to available resources in the presence of faults of a given fault model", which we divided into two control problems. First we solve "synthesize a controller to ensure that the critical tasks of system system_1 execute uninterruptedly in the presence of faults of a given fault model" then include the synthesized controller along with the critical tasks to solve "synthesize a controller to ensure that the non-critical tasks execute according to available resources in the presence of faults of a given fault model".

We briefly discuss the handling of systems with slightly different properties. For systems with asymmetric cores, which are unable to execute some tasks on some of the cores, we simply do not model the initialization, termination, killing, reassignment, and resumption for the illegal combinations of tasks and cores. For symmetric multi-core processing (SMP) one simply has to set the same load parameters on all the cores for each task. The synthesized reconfiguration services are oblivious to the tasks having substructure (subtasks), if they can be consistently abstracted by a single set of parameters (WCET, BCET and load).

We have assumed that an initialized task reallocated from a failed core should resume in the same state. If this is not required, i.e., a task can start from initial state on the new core at its next release period, then the model can be simplified, by removing the edges modeling resumption. We have not investigated the synthesis process for a scheduler with a dynamic allocation yet. In the next chapter, we present a theoretical framework for dynamic hierarchical open systems (such as system system_1) having any numbers of control hierarchies. The framework supports an automated state space reduction technique to allow timed games-based analysis for industrial dynamic hierarchical open systems.

Chapter 4

Timed Process Automata

This chapter develops a model for the *automated analysis* of *safety* and *reachability* properties in industrial *time-critical systems*. To fulfill industrial requirements, we consider time-critical systems that are open (communicate with external components), hierarchical (can be decomposed and recomposed into smaller control systems), and dynamic (the decomposition can change over time). In the chapter, we use *real-time systems*, meaning time-critical systems that fulfill all these features. The model also facilitates compositional modeling with reuse for different contexts.

An *open system* continuously interacts with an unpredictable environment. A good example of open time-critical systems is a pacemaker, which continuously interacts with a heart, an uncontrolled environment. The pacemaker's performance crucially depends on the exact timing of an action performed either by the system or by the environment. The *theory of timed games* [190, 4, 100, 95] is well-known in the research community for the analysis of open real-time systems.

A *hierarchical system* is a hierarchical composition of smaller systems. An automotive system, developed by an *original equipment manufacturer (OEM)*, may be used in different models of cars. In this case, the system has a *controller* which helps the system adapt to

different *environments* and cars. In other words, the system is an open system, which has two distinguished interacting segments: the controller and the environment. Typically, these systems consist of other smaller systems in a *hierarchical* structure. For instance, a system Actuator can be a component of a larger system Position, while Position can be a component of another system Brake-by-Wire, and so on. Every component of a system has a specific set of tasks; for example, system Brake-by-Wire may use its component Position to perform some desired tasks in interaction with the environment, and Brake-by-Wire may also indirectly—through using Position—use its sub-component Actuator to perform some desired tasks in interaction with the environment.

A *dynamic hierarchical system* is a hierarchical system whose components may change over time. Many hierarchical systems have dynamic characteristics, which are activating components only when needed. Dynamic behaviors are an important feature when resource constraints (such as limited memory) do not allow one to keep all the components active at the same time. Sometimes dynamic behaviors are inherent to the system. For example, we applied timed game theory in an industrial project to construct a fault-tolerant framework for a hierarchical open system that has a scheduler, a set of tasks, and a set of subtasks; only the scheduler is active in the initial system-state; subtasks are activated by their parent tasks, and the top level tasks are activated by their scheduler; thus the scheduler controls tasks, and a task controls its subtasks; due to the termination or the initialization of tasks (or subtasks) the structures of the processes may change; thus the system is a dynamic open system [233].

Modeling techniques, automated analyses, and other key issues of timed automata are typically addressed for *static closed systems*. The application domain of timed automata

is growing [230]. In our two projects with an automotive manufacturer, we used different timed automata-based analyses to investigate the fault-tolerance of real-time systems, which are part of many large-scale safety-critical systems. During our industrial projects, we observed that continuous-time formal methods of timed automata may provide the most accurate analysis; however, timed automata are not suited for industrial real-time systems mainly because of poor *scalability*. Moreover, we found that timed automata may introduce cumbersome design details in a large-scale real-time system having several control hierarchies. This chapter extends timed automata to achieve better modeling support and scalability for automated analysis of hierarchical open real-time systems.

We propose *timed process automata*, a variant of timed automata, for the development of industrial hierarchical open real-time systems. The proposed variant provides compositional modeling with reuse for three different contexts and automatable analysis—a system needs to be modeled and analyzed using timed process automata only once when copies of it are used as independent systems or multiple components of a larger system or components of different larger systems or a combination of all previous scenarios. The contributions of this chapter include:

1. Timed process automata, the first variant of timed automata that provides compositional modeling with reuse.

2. Definition of a formal semantics for timed process automata.

3. An automatable analysis for safety and reachability properties of timed process automata.

4. The first automatable state-space reduction technique for time-critical systems, which can be dynamic, hierarchical, and open.

The rest of the chapter can be divided into six sections:

Section 4.1 Describes the motivation for the work. The motivation is based on the experience achieved from a couple of automotive industrial projects, which are described in the previous chapters.

Section 4.2 Provides the required background to understand the chapter.

Section 4.3 Presents the syntax (Section 4.3.1) and the semantics (Section 4.3.2) of timed process automata, which use start actions, finish actions, final locations, and channels to facilitate compositional modeling to reuse designs without manual alterations.

Section 4.4 Presents an automatable analysis technique—based on timed games—for timed process automata. The analysis model of a timed process automaton T is constructed by composing a finite number of *timed I/O automata* [156, 4, 95], a variant of timed automata, to mimic the execution of T. The analysis model is constructed using an automatable technique that allows the designer to avoid manual alteration techniques for different compositions. Other than the automatable construction, the constructed analysis models essentially are timed I/O automata models, whose state spaces are too large to analyze industrial real-time systems.

Section 4.5 Develops an automatable state-space reduction technique for timed process automata that converts each callee process into a small automaton having only two locations and two edges, irrespective of the size of the callee. The technique uses structured construction of timed process automata, compositional reasoning, aggressive abstractions, and fewer synchronizations to ensure a smaller state space.

Section 4.7 Discusses related work. It also classifies timed process automata depending on the classification of timed automata variants presented in Chapter 2 and in [230].

4.1 Motivation

The first goal of the chapter is to develop a real-time model, where a designer will not need to readjust a design for different compositions. The second and main goal is to allow automated analysis of the model for industrial systems.

Figure 4.1 presents an abstract Brake-by-Wire system modeled using timed I/O automata, and the system is developed by an OEM. The model has seven automata representing different copies of only three elements: one copy of the *main thread* of Brake-by-Wire (the top automaton), two copies of the main thread of Position (the two automata in the middle), and four copies of Actuator system (the four automata in the bottom). Each Position system contains two *children* (Actuator systems) and its main thread that schedules the children, communicates with its parent (the main thread of Brake-by-Wire), and performs some other functions, which cannot be performed by the children. Similarly, this Brake-by-Wire system contains two children (Position systems) and its main thread that schedules the children and performs some other functions, which cannot be performed by the children. In this model, the main thread of Brake-by-Wire is the *root*, which does not have a parent. However, in the future a car manufacturer may include this Brake-by-Wire system in a car and then the main thread of Brake-by-Wire will no longer be the root. Then a central control system may be able to start the main thread of Brake-by-Wire. To analyze the new complex system, a designer will need to manually alter the model again by including *start* and *finish* actions (in the top automaton of Figure 4.1). Let us assume a complex system contains N Break-by-Wire systems; to analyze this complex system, a designer will need to manually construct at least $N \times 7$ automata with a proportionally growing alphabet! Existing timed automata-based modeling techniques do not support compositional modeling with reusable designs for different contexts; that is, a design may need to be altered

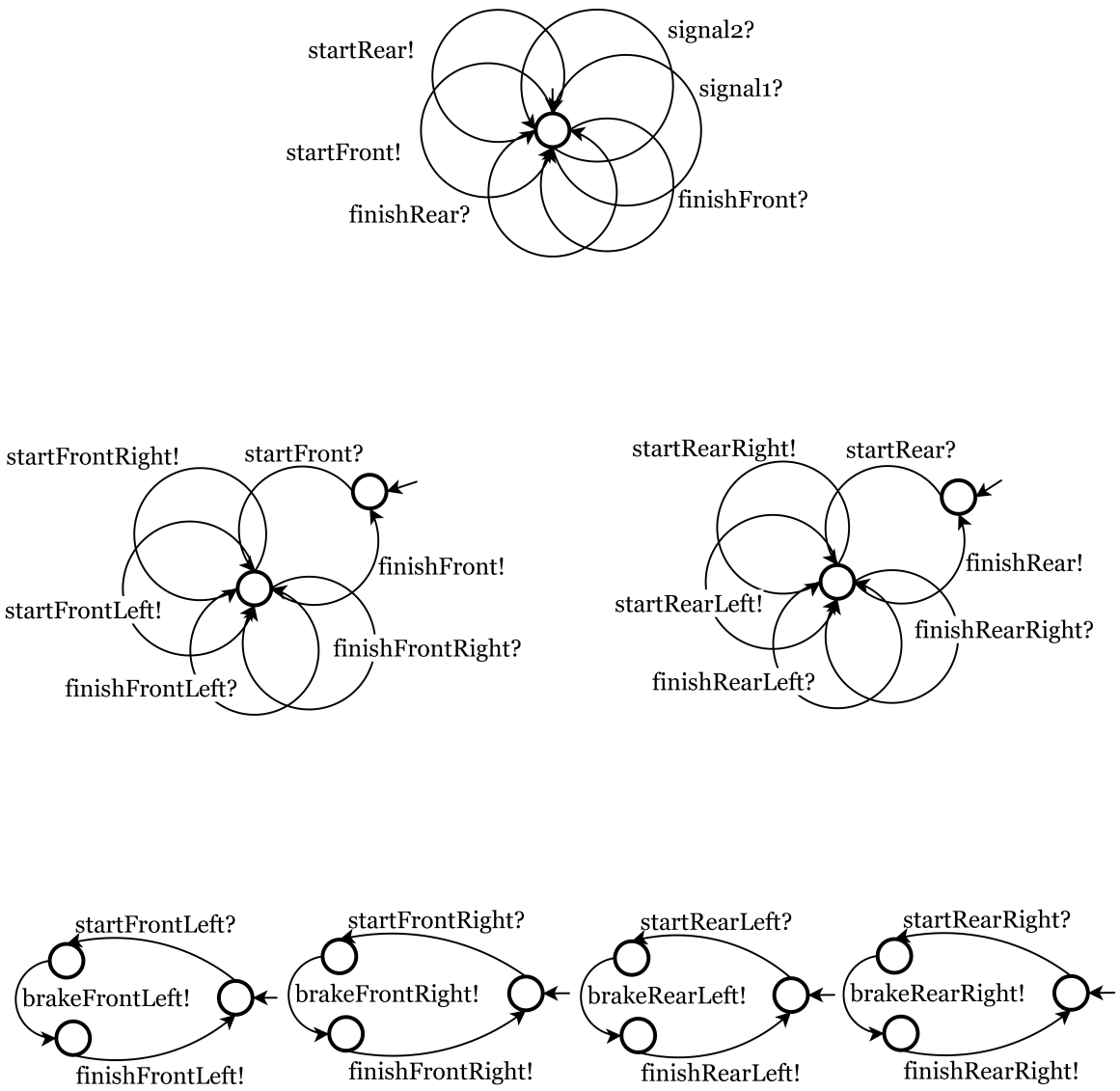


Figure 4.1: An abstract Brake-by-Wire system modeled using standard timed I/O automata, where one copy of the *main thread* of Brake-by-Wire (the top automaton), two copies of the main thread of Position (the two automata in the middle), and four copies of Actuator system (the four automata in the bottom)

manually in every composition. All these ad hoc alterations may make a large industrial design incomprehensible and error-prone.

Figure 4.2 contains the same Brake-by-Wire system of Figure 4.1 modeled by using

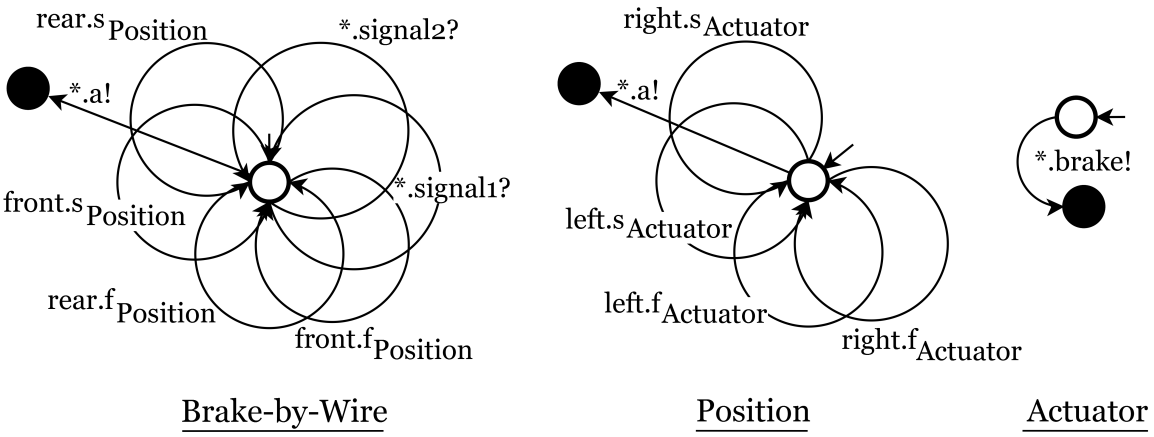


Figure 4.2: The same Brake-by-Wire system of Figure 4.1 is modeled using timed process automata

timed process automata. Timed process automata always model a system only once. For example, Figure 4.2 presents only three timed process automata, which are equivalent to the seven automata of Figure 4.1. Moreover, the number of copies and the root status of Break-by-Wire system has no impact on the new design.

No automated state-space reduction technique has been developed for the analysis of dynamic hierarchical open dense-time systems. During our two projects with an automotive manufacturer, we noticed that even a (practically) very small real-time system may have a state space too large for automated formal analysis because of hierarchy, dynamic behaviors, and time calculations. We overcame the scalability problem in one of the projects—construction of a fault-tolerance framework in Chapter 3 and in [233]—by developing a manual state-space reduction technique that applies aggressive abstractions and uses fewer synchronizations. Applying this manual technique to a design of an industrial system is a challenging task. A generalized automated reduction technique, therefore, is needed for analysis of dynamic hierarchical open time-critical systems, which is provided in this chapter by presenting an automatable reduction technique for timed process automata.

4.2 Background

The semantic construction of timed automata is expressed using semantics objects called *timed transition systems* [139, 95, 15]. A *timed I/O automaton* [156, 4, 95] is a timed automaton which has an input alphabet along with a regular output alphabet. The controller plays controllable output transitions and the environment plays uncontrollable input transitions; thus timed I/O automata are a natural model for timed games. Two timed I/O automata are *composable* with each other if they do not have a common output action. This section defines timed transition systems, timed I/O automata, composition of timed I/O automata, and all other terms required to understand the remaining chapter.

Definition 4. *[139, 95, 15] A* timed transition system *(with only one initial location but without final location and ϵ-transition) is a tuple $\mathcal{T} = (St, s_0, \Sigma, \dashrightarrow)$, where St is a set of* states*, $s_0 \in St$ is the* initial state*, Σ is an alphabet, and $\dashrightarrow : St \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times St$ is a* transition relation.

We use $d \in \mathbb{R}_{\geq 0}$ to denote delay. A timed transition system satisfies *time determinism* (i.e., whenever $s \overset{d}{\dashrightarrow} s'$ and $s \overset{d}{\dashrightarrow} s''$ then $s' = s''$ for all $s \in S$), *time reflexivity* (i.e., $s \overset{0}{\dashrightarrow} s$ for all $s \in S$), and *time additivity* (i.e., for all $s, s'' \in S$ and all $d_1, d_2 \in \mathbb{R}_{\geq 0}$ we have $s \overset{d_1+d_2}{\dashrightarrow} s''$ iff there exists an s' such that $s \overset{d_1}{\dashrightarrow} s'$ and $s' \overset{d_2}{\dashrightarrow} s''$). A *run* ρ of a timed transition system $\mathcal{T}$ from a state $s_1 \in St$ is a sequence $s_1 \overset{a_1}{\dashrightarrow} s_2 \overset{a_2}{\dashrightarrow} s_3 \cdots \overset{a_n}{\dashrightarrow} s_{n+1}$ such that for all $1 \leq m \leq n : s_m \overset{a_m}{\dashrightarrow} s_{m+1}$ with $a_m \in \Sigma \cup \mathbb{R}_{\geq 0}$. A state s is *reachable* in a transition system $\mathcal{T}$ if and only if there is a run $s_0 \overset{a_0}{\dashrightarrow} s_1 \overset{a_1}{\dashrightarrow} s_2 \cdots \overset{a_{n-1}}{\dashrightarrow} s_n$, where $s = s_n$. *Timed I/O transition systems* are timed transition system with input and output modalities on transitions. Timed I/O transition systems are used to define semantics of timed I/O automata.

A *clock* is a non-negative real variable. A *constraint* $\delta \in C(X, V)$ over a set of clocks X and over a set *counters*, non-negative finitely bounded integer variables, V is generated by

the grammar $\delta ::= x_m \prec q \mid k \prec \alpha \mid x_m - x_n \prec q \mid true \mid \Phi \wedge \Phi$, where $q \in \mathbb{Q}_{\geq 0}$, $\alpha \in \mathbb{Z}_{\geq 0}$, $\{x_m, x_n\} \subseteq X$, $k \in V$ and $\prec \in \{<, \leq, >, \geq\}$. Consequently, the set of *clock constraints* $C(X)$ is the set of constraints $C(X, V)$, where $V = \emptyset$. Let $\Psi(V)$ be the set of assignments over the set of variables V.

Definition 5. *[156, 4, 95, 15] A* timed I/O automaton *is a tuple* $\mathcal{A} = (L, l_0, X, V, A, E, I)$*, where* L *is a finite set of locations,* $l_0 \in L$ *is the initial location,* X *is a finite set of clocks,* V *is a finite set of counters,* $A = A_\mathrm{i} \oplus A_\mathrm{o}$ *is a finite set of actions, partitioned into input actions* A_i *and output actions* A_o*,* $E \subseteq L \times A \times \Phi(X, V) \times \Psi(V) \times 2^X \times L$ *is a set of edges, and* $I : L \to C(X)$ *is a total mapping from locations to invariants.*

A *clock valuation* over X is a mapping $\mathbb{R}^X_{\geq 0} : X \to \mathbb{R}_{\geq 0}$ and a *counter valuation* over V is a mapping $\mathbb{Z}^V_{\geq 0} : V \to \mathbb{Z}_{\geq 0}$. Given a clock valuation $v \in \mathbb{R}^X_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, we write $v + d$ for the clock valuation in which for each clock $x \in X$ we have $(v + d)(x) = v(x) + d$. For $\lambda \subseteq X$, we write $v[x \mapsto 0]_{x \in \lambda}$ for a clock valuation agreeing with v on clocks in $X \setminus \lambda$, and giving 0 for clocks in λ. For $\phi \in \Phi(X, V)$, $v \in \mathbb{R}^X_{\geq 0}$, and $n \in \mathbb{Z}^V_{\geq 0}$, we write $v, n \models \phi$ if v and n satisfy ϕ. Let $e = (l, a, \phi, \theta, \lambda, l')$ be an edge, then l is the source location, a is the action label, and l' is the target location of e; the constraint ϕ has to be satisfied during the traversal of e; the set of clocks $\lambda \in 2^X$ are reset to 0 and the set of counters are updated to θ whenever e is traversed.

Definition 6. *[4, 95] Two timed I/O automata* $\mathcal{A}^m = (L^m, l^m_0, X^m, V^m, A^m, E^m, I^m)$ *and* $\mathcal{A}^n = (L^n, l^n_0, X^n, V^n, A^n, E^n, I^n)$ *are* composable *with each other when* $A^m_o \cap A^n_o = \emptyset$*,* $X^m \cap X^n = \emptyset$*, and* $V^m \cap V^n = \emptyset$*; when composable, their* composition *is a timed I/O automaton* $\mathcal{A} = \mathcal{A}^m \| \mathcal{A}^n = (L^m \times L^n, (l^m_0, l^n_0), X^m \cup X^n, V^m \cup V^n, A, E, I)$*, where* $A = A_\mathrm{i} \cup A_\mathrm{o}$ *with* $A_\mathrm{o} = A^m_\mathrm{o} \cup A^n_\mathrm{o}$ *and* $A_\mathrm{i} = (A^m_\mathrm{i} \cup A^n_\mathrm{i}) \setminus A_\mathrm{o}$*. The set of edges* E *contains:*

- $((l^m, l^n), a, \phi^m \wedge \phi^n, \lambda^m \cup \lambda^n, \theta^m \cup \theta^n, (l'^m, l'^n)) \in E$ *for each* $(l^m, a, \phi^m, \theta^m, \lambda^m, l'^m) \in E^m$ *and* $(l^n, a, \phi^n, \theta^n, \lambda^n, l'^n) \in E^n$ *if* $a \in \{A_{\mathrm{i}}^m \cap A_{\mathrm{o}}^n\} \cup \{A_{\mathrm{o}}^m \cap A_{\mathrm{i}}^n\}$
- $((l^m, l^n), a, \phi^m, \lambda^m, \theta^m, (l'^m, l^n)) \in E$ *for each* $(l^m, a, \phi^m, \lambda^m, \theta^m, l'^m) \in E^m$ *if* $a \notin A^n$
- $((l^m, l^n), a, \phi^n, \lambda^n, \theta^n, (l^m, l'^n)) \in E$ *for each* $(l^n, a, \phi^n, \lambda^n, \theta^n, l'^n) \in E^n$ *if* $a \notin A^m$

and the set of invariants I *is constructed as follows:* $I(l^m, l^n) = I^m(l^m) \wedge I^n(l^n)$

4.3 Processes

Timed process automata model processes in a way that each process is a dynamic hierarchical open time-critical system, which we simply call real-time system in this chapter. Every process hierarchically contains its active callee processes. Thus the control of a process is hierarchically shared with its active callee processes. The main thread of a process can activate callee processes via communication channels. An active process can receive any input in any state. An active callee process can deactivate itself in any state of the main thread of its caller process. An activated callee process terminates within its worst-case execution time. This section presents the syntax and the semantics of timed process automata.

4.3.1 Timed Process Automata

Timed process automata are a variant of timed I/O automata. Unlike a timed I/O automaton, a timed process automaton has a finite set of *start actions* A_{s}, a finite set of *finish actions* A_{f}, a final location l_{f}, and a finite set of *channels* C.

The set of *actions* $A = A_{\mathrm{i}} \oplus A_{\mathrm{o}} \oplus A_{\mathrm{s}} \oplus A_{\mathrm{f}}$ of a timed process automaton is a disjoint union of finite sets of input actions A_{i}, output actions A_{o}, start actions A_{s}, and finish actions A_{f}. For every set of actions A, there exists a bijective mapping between its start actions A_{s}

and finish actions A_{f} in such a way that for each start action $\mathrm{s}_N \in A_{\mathrm{s}}$ there is exactly one finish action $\mathrm{f}_N \in A_{\mathrm{f}}$, and vice versa. These actions can be used for starting and finishing processes associated with N. We use s and f with the name N (of another timed process automaton) as a subscript index (e.g., s_N and f_N) to denote a start action and a finish action, respectively. We use the same subscript to indicate *paired* actions. We write a to denote an action in general. Processes synchronize via instantaneous channels. Each timed process automaton uses the same designated symbols for its *public channel* ($*$) and *caller channel* ($\triangle$). We use c to denote a channel in general.

Definition 7. *A* timed process automaton *is a tuple* $T = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$*, where* L *is a finite set of locations,* $l_0 \in L$ *is the initial location,* X *is a finite set of clocks,* $A = A_{\mathrm{i}} \oplus A_{\mathrm{o}} \oplus A_{\mathrm{s}} \oplus A_{\mathrm{f}}$ *is a finite set of actions as described above,* C *is a finite set of channels,* $E \subseteq (L \times A \times C \setminus \{\triangle, *\} \times \Phi(X) \times 2^X \times L) \cup (L \times (A_{\mathrm{i}} \cup A_{\mathrm{o}}) \times \{\triangle, *\} \times \Phi(X) \times 2^X \times L)$ *is a set of edges,* $I : L \to \Phi(X)$ *is a total mapping from locations to invariants, and* $l_{\mathrm{f}} \in L$ *is a designated final location which does not have any outgoing edges to other locations and has the invariant* $I(l_{\mathrm{f}}) = true$*.*

Figure 4.2 presents timed process automata Brake-by-Wire, Position, and Actuator. In the figure, each initial location has a dangling incoming edge, final locations are filled with black, and timed process automata names are underlined. The final location l_{f} of a timed process automaton may be unreachable from the initial location (and then l_{f} is not shown in the figure).

4.3.2 Process Executions

Every instance of a timed process automaton is a *process*. Two processes of the same timed process automaton represent two different copies of the same system. Every process has

a unique *process identifier*. A *process* is a tuple $P = (\mathsf{id}(P), \mathsf{tpa}(P), \mathsf{channel}(P))$, where $\mathsf{id}(P)$[1] is the process identifier, timed process automaton $\mathsf{tpa}(P)$ defines the execution logic, and *caller channel* $\mathsf{channel}(P)$ is the private channel to communicate with the caller and the other processes which are started via the same channel. A process Q is a *callee* of P if P is the caller of Q. We use $\perp$ to denote the caller channel of the root process. Every process P of $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_\mathrm{f})$ has its own copy $P.c$ of channel $c \in C$. We write $P.c.a$ meaning that action a is performed via channel $P.c$.

At the same time, no two processes of the same timed process automaton can have the same caller channel. A process P, therefore, may have at most $| C | \times | A_\mathrm{s} |$ active callee processes. For example, an instance of automaton Brake-by-Wire of Figure 4.2 can activate at most two instances of automaton Position of Figure 4.2 at the same time via two different channels front and rear, where the instance of Brake-by-Wire is the caller process of the two instances of Position, which are the callee processes of the instance of Brake-by-Wire. A *subprocess* is a callee or a callee of a subprocess, recursively. For example, every instance of Brake-by-Wire has six subprocesses: two instances of Position and four instances of automaton Actuator of Figure 4.2. Every process hierarchically contains all of its subprocesses. Two processes are *siblings* if they have the same caller channel. The caller can use separate channels to differentiate control over different callees, even if they are processes of the same automaton.

A process P starts a process Q of an automaton $\mathsf{tpa}(Q)$ via channel $P.c$ by traversing an edge $e_1 = (\_, \mathrm{s}_{\mathsf{tpa}(Q)}, c, \_, \_, \_)$ labeled by a start action $\mathrm{s}_{\mathsf{tpa}(Q)}$ if there exists no active process of tpa(Q) with caller channel $P.c$; dually, P traverses an edge $e_2 = (\_, \mathrm{f}_{\mathsf{tpa}(Q)}, c, \_, \_, \_)$ labeled by the paired finish action $\mathrm{f}_{\mathsf{tpa}(Q)}$ whenever Q reaches its final state. No edge labeled by $\mathrm{f}_{\mathsf{tpa}(Q)}$ will ever be traversed if $\mathsf{tpa}(Q)$ is a *non-terminating timed process automaton*.

[1] To avoid clutter, we abuse notation by writing P instead of $\mathsf{id}(P)$.

Correspondingly, note that existing processes may start different processes of $\mathsf{tpa}(Q)$—but always with different process identifiers. However, only P listens to finish action $\mathrm{f}_{\mathsf{tpa}(Q)}$ via channel $\mathsf{channel}(Q)$. Process P traverses an edge $e = (\_, a, c, \_, \_, \_)$ when P receives (respectively, sends) an input (resp., output) a in channel $P.c$. Process P communicates with its callee Q via $\mathsf{channel}(Q)$ and with the environment via channel $P.*$.

We formalize the above mechanics of execution by first giving the semantics of the main thread of the process, ignoring its subprocesses in Definition 8 and then giving the semantics of the entire process in Definition 15. The standalone semantics of a process are essentially the same semantics as a standard timed I/O automaton [15, 156, 4, 95]. The main difference is that states are decorated with process identifiers and edges with channel names to distinguish different instances of the same timed process automaton in Definition 15. Also the caller channel $\triangle$ is instantiated for an actual parent process. The technical reason for this will become apparent in Definition 15.

Definition 8. *The* standalone semantics $\mathcal{S}[\![P]\!]$ *of a process* $P = (P, \mathsf{tpa}(P), \mathsf{channel}(P))$ *is a timed I/O transition system* $\mathcal{S}[\![P]\!] = (L \times \mathbb{R}^X_{\geq 0} \times P, (l_0, \mathbf{0}, P), A^P, \twoheadrightarrow)$[2]*, where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_\mathrm{f})$, $\mathbf{0}$ *is a function mapping every clock to zero and* $\twoheadrightarrow \subseteq (L \times \mathbb{R}^X_{\geq 0} \times \{P\}) \times (A^P \cup \mathbb{R}_{\geq 0}) \times (L \times \mathbb{R}^X_{\geq 0} \times \{P\})$ *is the transition relation generated by the following rules:*

Action *For each clock valuation* $v \in \mathbb{R}^X_{\geq 0}$ *and each edge* $(l, a, c, \phi, \lambda, l') \in E$ *such that* $v \models \phi$, $v' = v[x \mapsto 0]_{x \in \lambda}$, *and* $v' \models I(l')$ *we have* $(l, v, P) \overset{P.c.a}{\twoheadrightarrow} (l', v', P)$ *if* $c \neq \triangle$, *otherwise* $(l, v, P) \xrightarrow{\mathsf{channel}(P).a} (l', v', P)$

Delay *For each clock valuation* $v \in \mathbb{R}^X_{\geq 0}$ *and for each delay* $d \in \mathbb{R}_{\geq 0}$ *such that* $(v + d) \models I(l)$ *we have* $(l, v, P) \overset{d}{\twoheadrightarrow} (l, v + d, P)$.

[2] A^P is the set of actions where action names are constructed using regular expression (P"."C | $\mathsf{channel}(P)$)"."A.

Theorem 1. *The transition system induced by the standalone semantics of a process is time deterministic, time reflexive, and time additive*

Proof. The Action transition rule does not allow hidden or internal transition. All non-action and delay related state changes, thus, occur according to the Delay transition rule. From this rule we can derive:

- The only state that can be reached from state (l, v, P) after delaying $d \in \mathbb{R}$ time units is $(l, v + d, P)$,
- The only state that can be reached from state (l, v, P) after delaying 0 time unit is (l, v, P), and
- For any two delays $d_1 \in \mathbb{R}$ and $d_2 \in \mathbb{R}$, the only state that can be reached from state (l, v, P) after delaying d_1 and d_2 time units is $(l, v + d, P)$ when $d_1 + d_2 = d$.

Therefore, the transition system induced by the standalone semantics of a process is time deterministic, time reflexive, and time additive. □

Definition 9. Ground timed process automata *are timed process automata that cannot perform a start or finish action ($A_s \cup A_f = \emptyset$).*

Automaton Actuator in Figure 4.2, for instance, is a ground timed process automaton.

Definition 10. Compound timed process automata *are timed process automata that can perform a start or finish action ($A_s \cup A_f \neq \emptyset$).*

For example, Brake-by-Wire and Position in Figure 4.2 are compound timed process automata.

Definition 11. *A* well-formed channel *cannot be used by two processes sharing an output action.*

Definition 12. *Processes of a* well-formed timed process automaton *have only well-formed channels.*

Definition 13. Non-recursive timed process automata *are defined inductively using the following rules:*

- *Every ground timed process automaton is a non-recursive timed process automaton, and*
- *A compound timed process automaton which performs only those start and finish actions whose subscripts are the names of some other existing non-recursive timed process automata is a non-recursive timed process automaton.*

All three automata in Figure 4.2, for example, are non-recursive timed process automata. A process of a non-recursive timed process automaton hierarchically contains only a finite number of subprocesses. The caller may activate an idle process, iteratively. Thus a process may activate a subprocess an arbitrary number of times. In this chapter, we are only concerned with non-recursive well-formed timed process automata.

A *standalone final state* of a process P is (l_{f}, v, P), where v is any clock valuation. We use st^P, st_0^P, c^P, and st_{f}^P to denote a standalone state, the standalone initial state, the set of channels, and a standalone final state of process P, respectively.

Definition 14. *We say that a process P is A'-enabled for a channel $P.c$ if for every reachable standalone state st^P we have $st^P \overset{P.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^P$ for some standalone state st'^P for each action $a \in A'$.*

We require that each process P is A_{i}-enabled (input enabled) for all channels of P, and A_{f}-enabled (finish enabled) for all channels of P other than channels $P.\triangle$ and $P.*$ to reflect the

phenomenon that inputs from the environment and the deaths of callees are independent events, beyond the control of a process. We present the semantics of a process in the following:

Definition 15. *The* global operational semantics $\mathcal{G}[\![P]\!]$ *(semantics* $[\![P]\!]$ *for short) of a process* $P = (P, \mathsf{tpa}(P), \perp)$ *are a timed I/O transition system* $\mathcal{G}[\![P]\!] = (2^s, s_0, \mathbb{P} \times \mathbb{C} \times \mathbb{A}, \rightarrow)$*, where* s *is the set of all the standalone states of all the processes in the universe,* $\mathsf{tpa}(P) = (L, l_0, X, A, E, I, l_\mathrm{f})$*,* $s_0 = \{st_0^P\}$ *is the initial state,* $\mathbb{P}$ *is the set of all the processes in the universe,* $\mathbb{C}$ *is the set of all the channels in the universe,* $\mathbb{A}$ *is the set of all the actions in the universe, and* $\rightarrow \subseteq 2^s \times (\mathbb{P} \times \mathbb{C} \times \mathbb{A} \cup \mathbb{R}_{\geq 0}) \times 2^s$ *is the transition relation generated by the following rules:*

$$\frac{st^Q \xrightarrow{Q.c.s_T} st'^Q \text{ and } c \notin \{\triangle, *\} \qquad \{st^W \in s \mid \mathsf{channel}(W) = Q.c \text{ and } \mathsf{tpa}(W) = T\} = \emptyset \qquad st^Q \in s \qquad (R, T, Q.c) \text{ is a freshly started process}}{s \xrightarrow{Q.c.s_T} \{s \setminus \{st^Q\}\} \cup \{st_0^R, st'^Q\}} \text{ Start}$$

$$\frac{st_\mathrm{f}^R, st^Q \in s \text{ and } \mathsf{channel}(R) = Q.c \qquad \{st^U \in s \mid \mathsf{channel}(U) \in C^R\} = \emptyset \qquad st^Q \xrightarrow{Q.c.\mathrm{f}_{\mathsf{tpa}(R)}} st'^Q}{s \xrightarrow{Q.c.\mathrm{f}_{\mathsf{tpa}(R)}} \{s \setminus \{st_\mathrm{f}^R, st^Q\}\} \cup \{st'^Q\}} \text{ Finish}$$

$$\frac{s' = \{st'^Q \mid st^Q \overset{d}{\twoheadrightarrow} st'^Q \text{ and } st^Q \in s \text{ and } (st^Q \neq st_\mathrm{f}^Q \text{ or } |s| = 1)\} \qquad |s| = |s'|}{s \xrightarrow{d} s'} \text{ Delay}$$

$$\frac{a \notin \bigcup_{st^Q \in s} A_\mathrm{o}^{\mathsf{tpa}(Q)} \qquad s' = \{st^Q \in s \mid st^Q \xrightarrow{Q.*.a} st'^Q\}}{s \xrightarrow{a} \{s \setminus s'\} \cup \{st'^Q \mid st^Q \xrightarrow{Q.*.a} st'^Q \text{ and } st^Q \in s\}} \text{ Input}$$

$$\frac{st^Q \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^Q \text{ and } a \in A_o^Q \text{ and } st^Q \in s \qquad s' = \{st^R \in s \mid st^R \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^R \text{ and } W.c \text{ is a channel}\}}{s \xrightarrow{Q.c.a} \{s \setminus s'\} \cup \{st'^R \mid st^R \overset{W.c.a}{\longrightarrow\!\!\!\!\!\rightarrow} st'^R \text{ and } st^R \in s\}} \text{ Output}$$

A *global state* is a set which holds standalone states of all active processes. The Start rule states that the initial standalone state of a freshly started callee is added to the global state whenever the corresponding start action is performed by its caller. The rule also states that no two active processes can have the same timed process automaton and the same caller channel. The Finish rule prescribes that the standalone-final state of a callee is removed from the global state and the caller executes the corresponding finish action whenever that callee is in the standalone-final state and no standalone state of its subprocesses is in global state. Thus the rule defines *global-final state* (*final state* for short) of a process: a process is in its the final state when the process is in its final location and the process has no active subprocess. The Delay rule declares that globally a process can delay if that process and all of its active subprocesses can delay in their respective standalone semantics. Every subprocess is a part of the root process and thus if a subprocess is performing an action (or not idle) then the root process is also not idle. The rule also says that a process cannot delay if that process or any of its subprocess is in its global final state. That means a process finishes as soon as it reaches its final state. The Input rule states that a process receives an input from the environment via channel $id.*$. Rule Output declares a process send an output via channel $id.c$ to others who share $id.c$.

Theorem 2. *The transition system induced by the global semantics is time deterministic, time reflexive, and time additive.*

Proof. The global semantics of a process of a ground timed process automaton is its standalone semantics. Therefore, the transition system induced by Definition 15 for that process

is time deterministic, time reflexive, and time additive.

Standalone states of a subprocess can be part of global states only after that subprocess is started. Whenever a subprocess reaches its terminal state, its standalone states can never be part of the global state because of the Delay rule and the Finish rule. Therefore, a nonactive subprocess has no impact on the transition system. None of the action transition rules allows hidden or internal transition. All non-action and delay related state changes, thus, occur according to the Delay transition rule. From this rule we can derive for a process P having n active subprocesses $P_1, P_2, \cdots, P_n$:

- The only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying $d \in \mathbb{R}$ time units is $\{(l, v + d, P), (l_1, v + d, P_1), (l_2, v + d, P_2), \cdots, (l_n, v + d, P_n)\}$,
- The only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying 0 time units is $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$, and
- For any two delays $d_1 \in \mathbb{R}$ and $d_2 \in \mathbb{R}$, the only state that can be reached from state $\{(l, v, P), (l_1, v, P_1), (l_2, v, P_2), \cdots, (l_n, v, P_n)\}$ after delaying d_1 and d_2 time units is is $\{(l, v + d, P), (l_1, v + d, P_1), (l_2, v + d, P_2), \cdots, (l_n, v + d, P_n)\}$.

Therefore, the transition system induced by Definition 15 is time deterministic, time reflexive, and time additive. □

The process semantics, hence, defines a well-formed timed I/O transition system. This allows us to use timed automata as a basis for analyzing timed process automata.

Definition 16. *A* local run *of the main thread of a process P is a standalone run of P for which there exists a global run of P such that every transition of that standalone run occurs in that global run.*

The *local behavior* of the main thread of P consists of all of its local runs.

4.4 Analysis

We are interested in *safety* and *reachability* properties of timed process automata. This section explains how such analyses can be performed using the theory of timed games. A standard timed I/O automaton can be viewed as a concurrent two-player timed game, in which the players decide both which action to play, and when to play it. The input player represents the environment, and the output player represents the system itself. Similarly, the main thread of a process acts as a concurrent two-player timed game: the environment plays input transitions and finish transitions, and the main thread of the process plays output transitions and start transitions. Let's consider interactions of a process defined in the previous section. A process controls its output and start transitions. After starting a callee, the main thread of the caller knows that the paired finish action will arrive within the worst-case execution time of the associated callee. However, the main thread does not have any control on the exact arrival time of a finish action. Finish transitions along with input transitions are uncontrollable. The environment of the main thread of a process consists of all the connected processes (such as caller, siblings, and subprocesses) and all unconnected entities.

A global state of a process is safe if and only if all of the standalone states which it holds contain no unsafe location. A *safety property* asserts that the system remains inside a set of global-safe states regardless of what the environment does. We are interested in *Safety Property I: Given a process P and a set of unsafe locations L_U of P, can the controller avoid L_U in P regardless of what the environment does?*

A global state of a process is a target state if and only if at least one of its standalone

states contains a target location. A *reachability property* asserts that the system reaches any of the global-target states regardless of what the environment does. We are interested in *Reachability Property I: Given a process P and a set of target locations L_T of P, can the controller reach a location of L_T in P regardless of what the environment does?*

The monolithic analysis constructs a static network of automata to represent all possible global executions by mimicking the hierarchical call tree of the analyzed process. It simulates a process execution by changing states of pre-allocated timed I/O automata which fall into two groups: a *root automaton* to simulate the local behaviors of the main thread of the root process and a finite set of *standalone automata* to simulate the local behaviors of the main threads of the subprocesses.

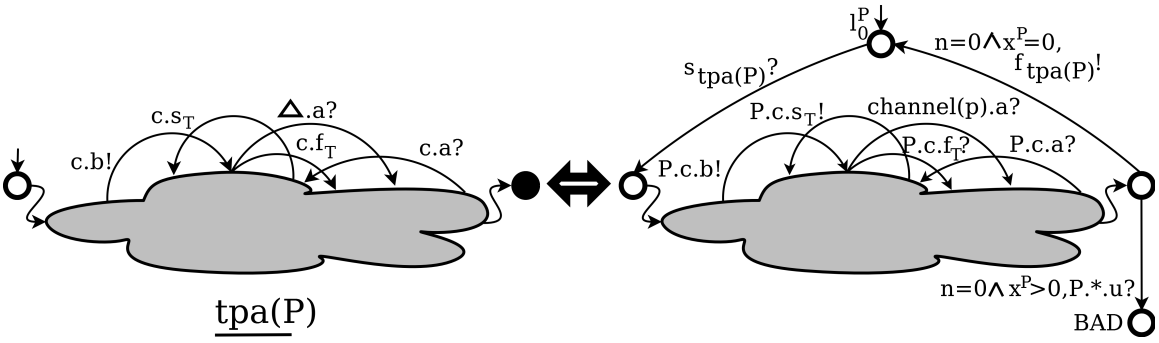


Figure 4.3: A generalized view of the standalone automata construction

Standalone Automata We construct a standalone automaton for each subprocess to simulate the main thread of that process. To construct a standalone automaton, we prefix the timed process automaton with a simulated start action and suffix it with a simulated finish action. We use non-negative finitely bounded integer variables[3] in standalone automata to count the number of active callees, in order to detect termination. We rename actions (e.g.,

[3]The use of non-negative finitely bounded integer variables can be avoided if a more cumbersome encoding is used.

a) of processes uniformly to encode channel names (e.g., $P.c$) in action names (e.g., $P.c.a$) of standalone automata; because standard timed I/O automata do not support private channels. A standalone automaton includes all the locations and slightly altered edges of the corresponding timed process automaton. Moreover, each standalone automaton has two additional locations: a new initial location l_0^{id} to receive (resp., send) a start (resp., finish) message from (resp., to) the caller, and a new unsafe location BAD to prevent the automaton from waiting in final states instead of finishing. Every start (resp., finish) increments (resp., decrements) a counter variable n. The automaton represents finishing of the process in the final location when $n = 0$.

Definition 17. *The standalone automaton of process P is* $\mathsf{standalone}(P) = (L \cup \{l_0^{\mathrm{P}}, BAD\}, l_0^{\mathrm{P}}, X \cup \{x^{\mathrm{P}}\}, \{n\}, A^{\mathrm{P}}, E^{\mathrm{P}}, I^{\mathrm{P}})$, *where* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$, l_0^{P} *and* BAD *are two newly added locations,* x^{P} *is a newly added clock,* n *is a non-negative finitely bounded integer variable with the initial value 0,* $A_{\mathrm{o}}^{\mathrm{P}} = A'_{\mathrm{o}} \cup A'_{\mathrm{s}} \cup \{\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}\}$ *and* $A_{\mathrm{i}}^{\mathrm{P}} = A'_{\mathrm{i}} \cup A'_{\mathrm{f}} \cup \{\mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}, P.*.u\}$ *such that* $A'_m = \{\mathsf{channel}(P).a \mid a \in A_m\} \cup \{P.c.a \mid a \in A_m \text{ and } c \in C \setminus \{\triangle\}\}$ *where* $m \in \{\mathrm{o}, \mathrm{s}, \mathrm{i}, \mathrm{f}\}$ *and newly added actions are* $\mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}$, $\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}$, *and* $P.*.u$. *The set of edges* E^{P} *contains*

- *Converted edges that do not communicate via caller channel* $\triangle$*:*
  - *An edge* $(l, P.c.a, \phi, \xi, \lambda \cup \lambda', l') \in E^{\mathrm{P}}$ *for each edge* $(l, a, c, \phi, \lambda, l') \in E$, *where* $c \in C \setminus \{\triangle\}$, *the integer assignment is empty* $\xi = \emptyset$ *when* $a \in A_{\mathrm{o}} \cup A_{\mathrm{i}}$, $\xi = \{n--\}$ *when* $a \in A_{\mathrm{f}}$, *and* $\xi = \{n++\}$ *when* $a \in A_{\mathrm{s}}$
- *Converted edges that communicate via caller channel* $\triangle$*:*
  - *An edge* $(l, \mathsf{channel}(P).a, \phi, \emptyset, \lambda \cup \lambda', l') \in E^{\mathrm{P}}$ *for each edge* $(l, a, \triangle, \phi, \lambda, l') \in E$

- *Additional new edges that simulate activation and deactivation:*
  - *Three more edges* $(l_0^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, X, l_0), (l_{\mathrm{f}}, \mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}, n = 0 \wedge x^P = 0, \emptyset, \emptyset, l_0^{\mathrm{P}}), (l_{\mathrm{f}}, P.*.u, n = 0 \wedge x^P > 0, \emptyset, \emptyset, BAD)$ *are in* E^{P}

$\lambda' = \emptyset$ *when* $l' \neq l_{\mathrm{f}}$, *otherwise* $\lambda' = \{x^{\mathrm{P}}\}$. *The invariant function* I^{P} *maps each location* $l \in L$ *to* $I(l)$ *and maps each location* $l \in \{l_0^{\mathrm{P}}, BAD\}$ *to true.*

The standalone semantics of automaton $\mathsf{tpa}(P)$ and the semantics of standalone automaton $\mathsf{standalone}(P)$ are essentially the same in a way that both have the same safety and reachability properties (that we consider) of the corresponding process.

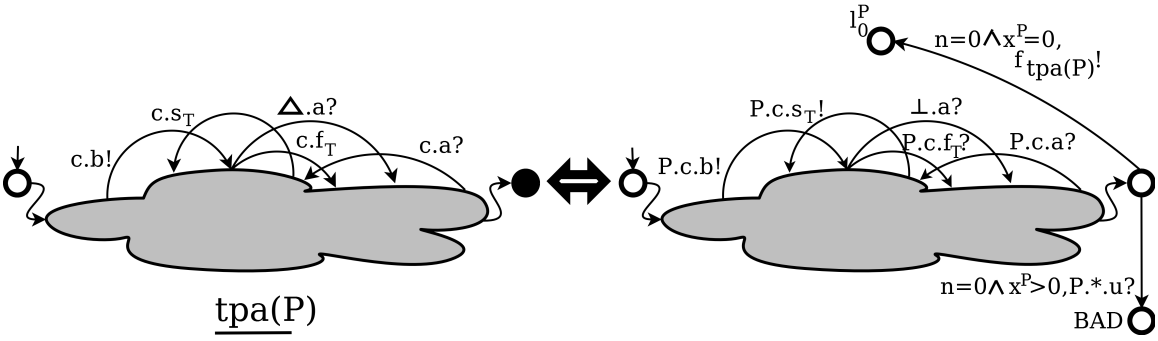


Figure 4.4: A generalized view of the root automata construction

Root Automata We construct a root automaton to simulate the main thread of the analyzed process.

Definition 18. *To analyze a timed process automaton* $\mathsf{tpa}(P) = (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$, *we construct the root automaton* $\mathsf{root}(\mathrm{P})$ *of process P. Standalone automaton* $\mathsf{standalone}(P)$ *is slightly different from* $\mathsf{root}(\mathrm{P})$. *The differences are:*

- *The caller channel is always* $\perp$,

- *The initial location of root automaton* root(P) *is the location* l_0*, which is also the initial location of* tpa(P)*, and*

- *Root automaton does not have edge* $(l_0^{\mathrm{P}}, \bot.\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, X, l_0)$*, which simulates activation of* P*.*

Monolithic Analysis Model Monolithic analysis models can be constructed in an automatable process.

Definition 19. *The monolithic analysis model of a ground timed processes automaton (such as* Actuator*) is its root automaton. We construct the monolithic analysis model of automaton* tpa(P) *in the following iterative manner:*

First Step: We construct the root automaton root(P)*.*

Iterative Step: We construct a standalone automaton for each triple (Q, s_T, c)*, where* Q *is process for which we have constructed a standalone automaton or the root automaton,* tpa(Q) $= (L, l_0, X, A, C, E, I, l_{\mathrm{f}})$*,* $c \in C \setminus \{\triangle, *\}$*,* $\mathrm{s}_T \in A_{\mathrm{s}}$*, and* $(\_, \mathrm{s}_T, c, \_, \_, \_) \in E$*.*

Figures 4.3–4.4 present a generalized view of the standalone and root automata constructions (a technical report [231] and the appendix present monolithic analysis models of processes of automata Actuator, Position, and Brake-by-Wire). The monolithic analysis model constructs a parallel composition of all the timed I/O automata constructed above. The construction is finite, and the composition is a timed I/O automaton, because we consider only non-recursive well-formed timed process automata.

We add avoiding *BAD* locations to our safety and reachability properties analyses. We convert Safety Property I to *Safety Property II: Given a process P and a set of unsafe locations* L_U *of P, can the controller avoid* L_U *and all the BAD locations in the analysis*

model regardless of what the environment does? We also convert Reachability Property I to *Reachability Property II: Given a process P and a set of target locations L_T of P, can the controller reach a location of L_T in the analysis model avoiding all the BAD locations regardless of what the environment does?* Special actions are added with timed process automata to construct corresponding root and standalone automata to simulate starts and finishes of processes. Avoiding all the newly added *BAD* locations in the analysis model ensures that each caller process performs the corresponding finish action as soon as the callee finishes—exactly as described in the global semantics. Executions (of the analysis model) that avoid all the newly added *BAD* locations, when projected on the original alphabet, are identical to the executions of the global semantics. Thus, if a Safety Property I (resp., Reachability Property I) holds for a process then its corresponding Safety Property II (resp., Reachability Property II) also holds in the analysis model, and vice versa.

Definitions 17, 18, and 19 provide automatable techniques to construct standalone automata, root automata, and monolithic analysis models, respectively. Thus one can remove manual alterations—such as manual renaming—by making these constructions automatic.

4.5 Automatable State-Space Reduction

We introduce an automatable state-space reduction technique for timed process automata to counteract state-space explosion. The technique relies on compositional reasoning, aggressive abstractions, and reducing process synchronizations. In the monolithic analysis of Section 4.4, a callee can be represented by an arbitrary number of standalone automata, and each of these automata can be arbitrarily large. The compositional reasoning technique described in this section replaces hierarchical trees of standalone automata representing subprocesses with simple abstractions (Figure 4.5)—so called *duration automata*.

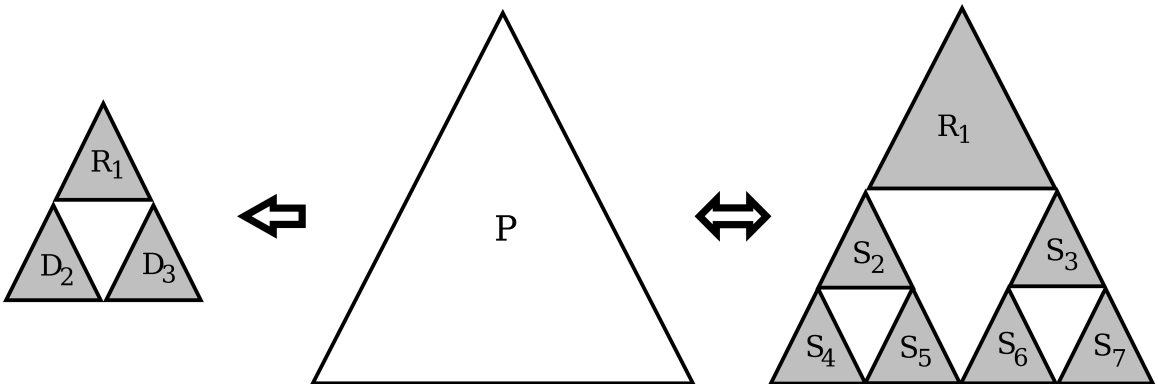


Figure 4.5: A compositional (sound) analysis model on the left and a monolithic (sound and complete) analysis model on the right of automaton Brake-by-Wire, where P is a process of the automaton, R_1 is the root automaton, S_2–S_7 are standalone automata, and D_2–D_3 are duration automata

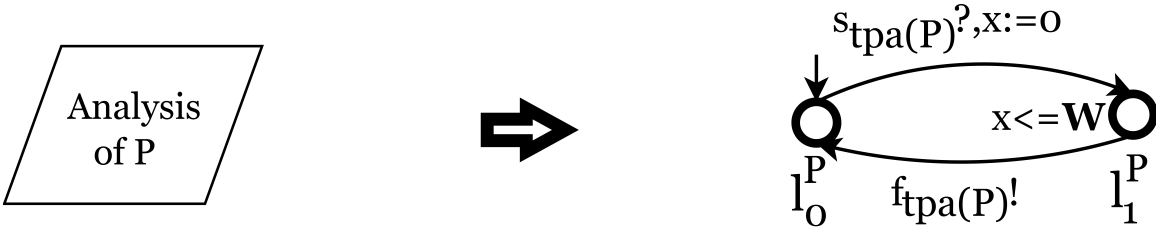


Figure 4.6: A generalized view of duration automata construction

Duration Automata A duration automaton (Figure 4.6) is timed I/O automaton with only two locations: the initial location (l_0^P) and the active location (l_1^P). A duration automaton of an analyzed process abstracts all the information of global executions of the process other than its *worst-case execution time (WCET)*. It can capture safety and reachability properties of interest. The *minimal-time safe reachability* of a target location is the *minimal-time reachability* [74, 153] for which the controller has a winning strategy to reach that target location by avoiding unsafe states. Like [79, 130], we assume that the WCET **W** of a process *P* is the minimal-time safe reachability time to reach location l_0^P of automaton

root(P) in the analysis model of P. The WCET of P is unknown ($\mathbf{W} = \infty$) when there is no winning strategy for the minimal-time safe reachability to reach location l_0^{P} of root(P).

Definition 20. *The duration automaton of process P is* duration(P) = $(\{l_0^{\mathrm{P}}, l_1^{\mathrm{P}}\}, l_0^{\mathrm{P}}, \{x^{\mathrm{P}}\}, \emptyset, A^{\mathrm{P}}, E^{\mathrm{P}}, I^{\mathrm{P}})$, *where* tpa($P$) = $(L, l_0, X, A, C, E, I, l_{\mathrm{f}})$, $A_{\mathrm{i}}^{\mathrm{P}} = \{\mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}\}$, $A_{\mathrm{o}}^{\mathrm{P}} = \{\mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}\}$, *the set of edges* $E^{\mathrm{P}} = \{(l_0^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{s}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, \{x^{\mathrm{P}}\}, l_1^{\mathrm{P}}), (l_1^{\mathrm{P}}, \mathsf{channel}(P).\mathrm{f}_{\mathsf{tpa}(P)}, \emptyset, \emptyset, \emptyset, l_0^{\mathrm{P}})\}$, *invariant* I^{P} *maps location* l_0^{P} *to true, and* I^{P} *maps location* l_1^{P} *to* $x^{\mathrm{P}} \leq \boldsymbol{W}$.

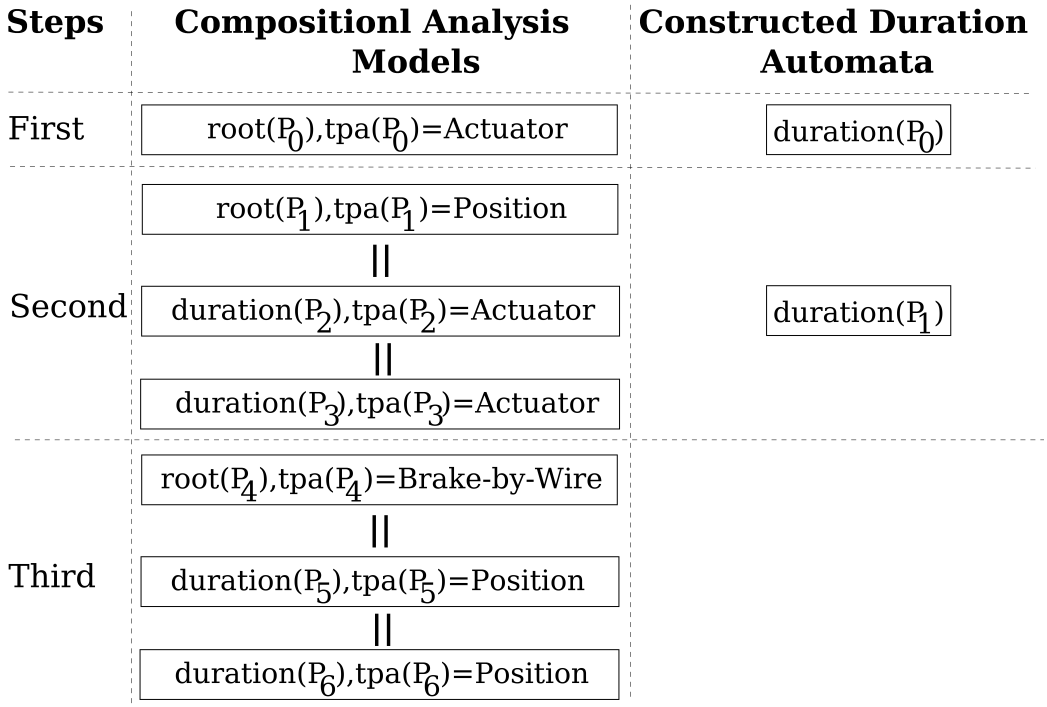


Figure 4.7: Steps of the compositional analysis of automaton Brake-by-Wire. In this figure, root(P_0), tpa(P_0) = Actuator means root automaton of process P_0, where P_0 is an instance of Actuator, and similar interpretations apply for root(P_1), tpa(P_1) = Position, duration(P_2), tpa(P_2) = Actuator, and so forth.

Compositional Analysis Model We construct the compositional analysis model in a bottom-up manner: analysis of a compound process is performed only after analyzing all

its callees. Like the monolithic analysis, the compositional analysis model of a ground timed process automaton tpa(Q) (such as Actuator) is a root automaton of process Q. That timed I/O automaton is analyzed to construct a duration automaton of Q. For a compound process P, we analyze automaton root(P) in the context of the duration automata of its callees (instead of the entire hierarchical structure of subprocesses).

Definition 21. *We construct the compositional analysis model of a timed process automaton* tpa(P) *in the following manner:*

First Step: We construct the root automaton root(P).

Second Step: We construct a duration automaton for each triple (P, s_T, c)*, where* tpa(P) = $(L, l_0, X, A, C, E, I, l_\mathrm{f})$*,* $c \in C \setminus \{\triangle, *\}$*,* $\mathrm{s}_T \in A_\mathrm{s}$*, and* $(\_, \mathrm{s}_T, c, \_, \_, \_) \in E$.

Figure 4.7 presents the compositional analysis procedure of Brake-by-Wire (the detailed models are presented in [231]). The compositional model construction procedure terminates, and the composition of all the above timed I/O automata is a timed I/O automaton, because we consider only non-recursive well-formed timed process automata.

The duration automaton of a process can capture safety properties: if a process has a winning strategy for a safety game, then both locations of its duration automaton are considered safe; otherwise, the active location (l_1^{id}) of the duration automaton is added to the set of unsafe locations L_U. Now this duration automaton can be used as a sound context to analyze the caller automaton for safety. A safety property holds for a compound process when the main thread of the process satisfies the property locally and allows the activation of a callee only if that callee also satisfies the property.

Duration automata can also capture reachability properties: if a process has a winning strategy for a reachability game then the active location (l_1^{id}) of the duration automaton

is added to the set of target locations L_T; otherwise, no target location is specified for this callee. This duration automaton can be used as a sound context to analyze the caller automaton for reachability. A reachability property holds for a compound process when the main thread of the process can reach the target locally or can activate a callee where the property holds. Like the monolithic analysis, the compositional analysis is performed for Safety Property II and Reachability Property II.

Theorem 3. *The compositional analysis is sound.*

Proof. A duration automaton does not contain any input and output actions of its process. Hence, the root automaton in a compositional model does not synchronize with the input and output actions of its callees—instead the automaton synchronizes for those actions with the environment. The duration automaton was created under the assumption that inputs are uncontrollable, so ignoring synchronization with inputs is sound. Similarly, it is sound to open the inputs of the root automaton from a callee, as they will be treated as uncontrollable and unpredictable actions, so will be analyzed in a more "hostile" environment than before the abstraction. Therefore, if a property holds in the compositional analysis then it also holds for the monolithic analysis. In other words, if a safety or reachability property holds in compositional analysis then it holds in the global semantics. □

Our compositional analysis is not complete because it is based on potentially quite coarse abstractions. In compositional analysis, abstracting from the input and output actions of callees and subprocesses causes the process to be analyzed in a more "hostile" environment (i.e., an environment in which no assumptions whatsoever are made about the timing and relative order of these actions. Therefore, the process might have a winning strategy in its actual operating environment, when our compositional analysis produces the opposite result. Definitions 18, 20, and 21 provide automatable techniques to construct root

automata, duration automata, and compositional analysis models, respectively. Thus one can automatically reduce state space by implementing our constructions.

4.6 Experimental Results

In all the steps of Figure 4.7, the largest composition contains only three automata, and except for the root automaton all are tiny duration automata. A monolithic analysis model of Brake-by-Wire is a composition of seven automata presented in Appendix D. A duration automaton always has a small constant size (modulo the size of the WCET constant), and so its state space is very simple (actually the discrete state space is independent of the input model).

We applied our approach to the case study presented in Chapter 3 in the following way:

- First, we model the central reconfiguration service (in Figure 4.8) and three tasks: `S` (in the top of Figure 4.10), `W` (in the top of Figure 4.11), and `D` (in the top of Figure 4.12) using timed process automata. The automaton in Figure 4.8 also models task releases (using start actions `sS`, `sW`, and `sD`) and terminations (using finish actions `fS`, `fW`, and `fD`). Like the concrete and abstract models of Chapter 3, an unsafe location `BAD` in this automaton is unreachable—a central reconfiguration service (or a controller) exists that makes the system fault tolerant—when the total load of no core can exceed its load limit. Similar to the concrete model, timed process automata of the tasks keep all the internal states of the corresponding tasks. Like the abstract model, the currently assigned core information is encoded into the automaton of Figure 4.8. These timed process automata together model system system_1 of Chapter 3 in a more abstract way than the concrete model but in a less abstract way than the abstract model.

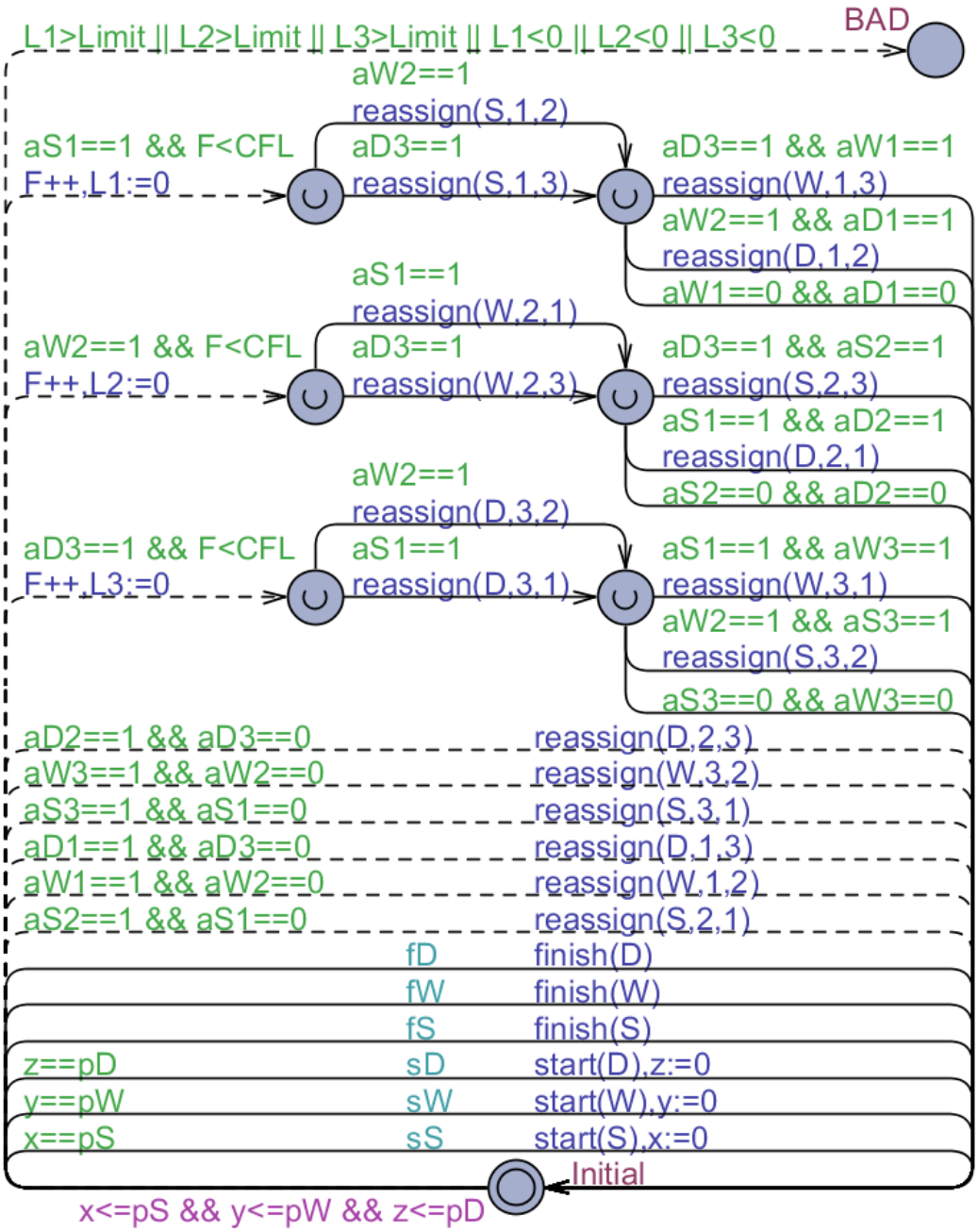


Figure 4.8: A timed process automaton representing the central reconfiguration service

- After that, according to the construction technique of Section 4.4, we construct the

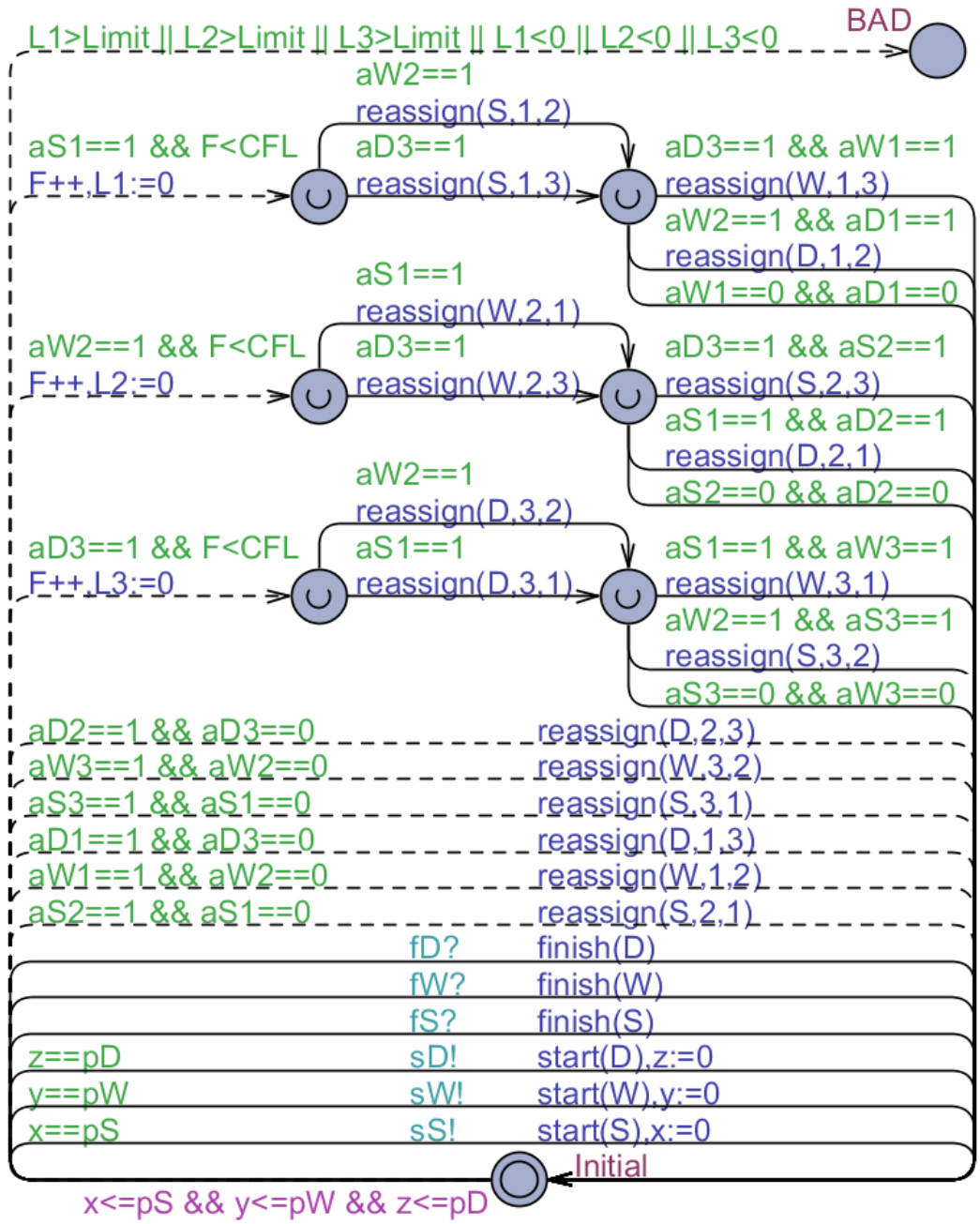


Figure 4.9: Root automaton of the central reconfiguration service

standalone automata (presented in the middle of Figures 4.10–4.12) of the timed process automata representing tasks (presented in the top of Figures 4.10–4.12) and the

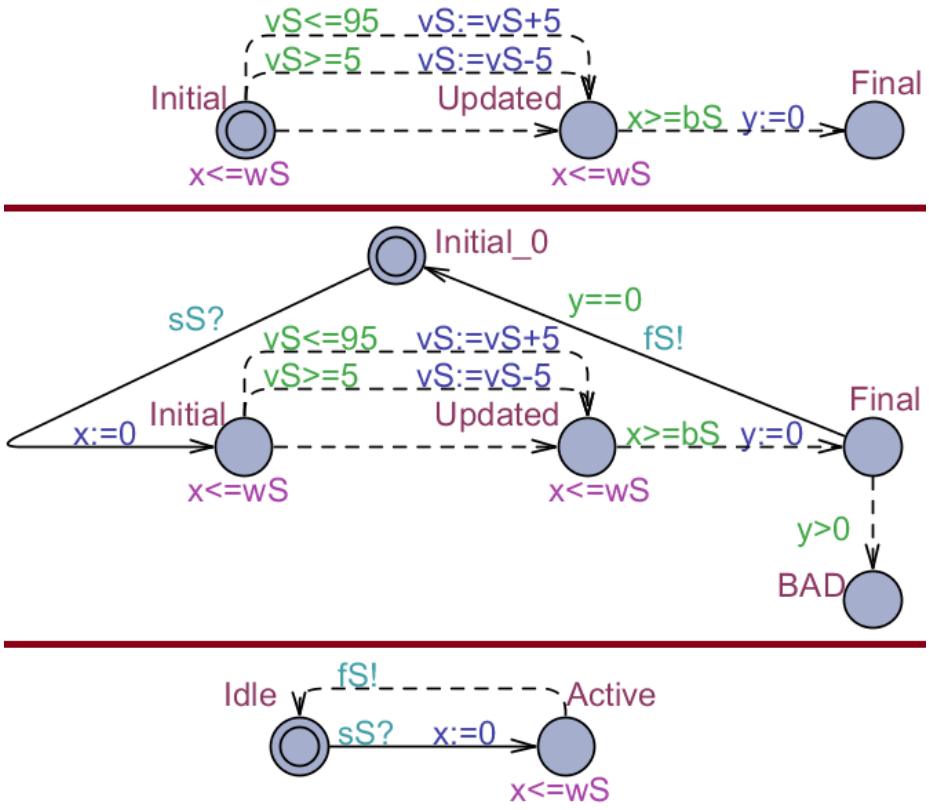


Figure 4.10: Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task S of Chapter 3

root automaton (presented in Figure 4.8) of the timed process automaton representing the central reconfiguration service (presented in Figure 4.9). The composition of these four timed I/O automata represents a monolithic analysis model of system system_1 of Chapter 3, and we simply call this model the monolithic model. Configurations of system system_1 of Chapter 3 are combinations of different worst-case

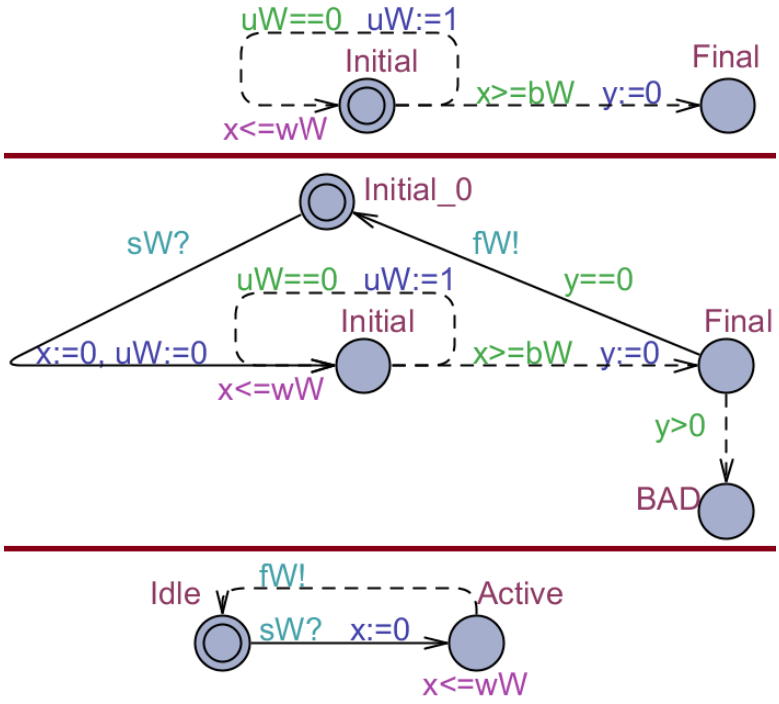


Figure 4.11: Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task `W` of Chapter 3

loads of tasks on different cores, different worst-case execution times of tasks, different best-case execution times of tasks, and different release periods of tasks. Existence of a central reconfiguration service (or controller) depends on the current configuration. We analyze the monolithic model against 20 configurations of Table 4.1[4], which is a copy of Table 3.1. Like Section 3.5, all the analyses were performed by Uppaal Tiga-0.17 on a PC with an Intel Core i3 CPU at 2.4 GHz, 4 GB of RAM, and running 64-bit Windows 7. Table 4.2 represents the analysis results in the form of controller synthesis time (in seconds) and the strategy size (in kilobytes). Unlike the previous chapter, we are not mainly concerned with controller synthesis from timed

[4]To show clearer impacts of different modeling aspects on the analysis, we picked some imaginary system configurations instead of some actual system configurations.

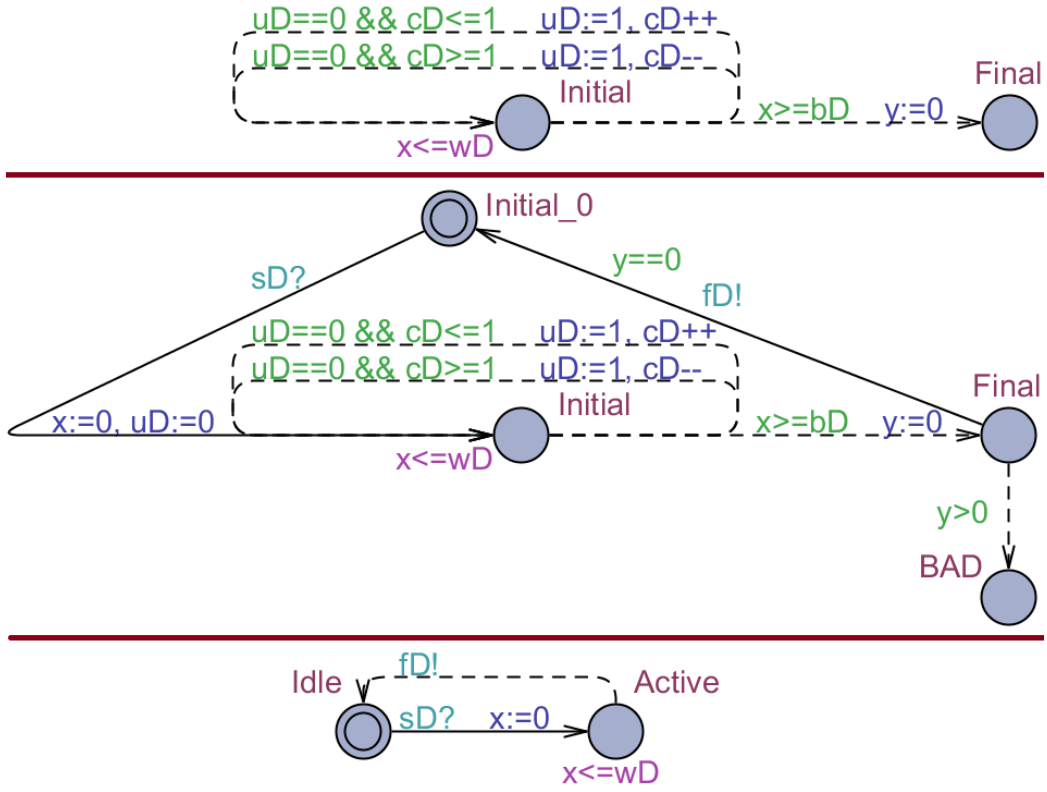


Figure 4.12: Timed process automaton (in the top), standalone automaton (in the middle), and duration automaton (in the bottom) of task D of Chapter 3

process automata—rather only checking the existence of a controller. We, however, also synthesized the controller because the synthesis time and the strategy size convey a clearer idea regarding the size of the state space. Moreover, they allow us to compare the models of this chapter with the models of the previous chapter. The monolithic model produces large state spaces, and for many configurations state-space explosion occurred, such as for configurations C3 (for CFL 1), C4, C5, C7, C8 (for CFL 2), C9, C10, C11, C12, C13, C14 (for CFL 2), C15 (for CFL 2), C16, C17 (for CFL 2), C18 (for CFL 2), and C19 (for CFL 2).

Configuration	Period of task			WCET of task			BCET of task			Load on $core_1$ of task			Load on $core_2$ of task			Load on $core_3$ of task		
	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D	S	W	D
C1	10	10	10	5	5	5	4	4	4	60	45	5	10	80	5	10	5	85
C2	10	10	10	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C3	10	15	20	5	5	5	0	0	0	60	45	5	10	80	5	10	5	85
C4	10	15	20	5	5	5	0	0	0	60	35	5	10	80	5	10	5	85
C5	10	15	20	5	5	5	0	0	0	43	37	7	11	67	19	23	13	59
C6	10	15	20	5	5	5	0	0	0	43	37	59	11	67	39	23	13	59
C7	10	15	20	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C8	10	15	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C9	10	20	30	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C10	11	19	31	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C11	5	7	11	5	5	5	0	0	0	33	33	33	33	33	33	33	33	33
C12	5	7	11	5	3	2	0	0	0	33	33	33	33	33	33	33	33	33
C13	5	7	11	5	3	2	5	3	2	33	33	33	33	33	33	33	33	33
C14	10	15	20	5	5	5	5	5	5	33	33	33	33	33	33	33	33	33
C15	10	15	20	5	7	11	5	7	11	33	33	33	33	33	33	33	33	33
C16	10	15	20	5	7	11	0	0	0	33	33	33	33	33	33	33	33	33
C17	10	15	20	7	7	7	7	7	7	33	33	33	33	33	33	33	33	33
C18	10	15	20	5	7	7	5	7	7	33	33	33	33	33	33	33	33	33
C19	10	15	20	7	7	11	7	7	11	33	33	33	33	33	33	33	33	33
C20	10	15	20	9	13	19	9	13	19	33	33	33	33	33	33	33	33	33

Table 4.1: Different configurations: combinations of release period, WCET, and BCET have abstract time units; and loads are in % of the respective core

- At the end, according to the construction technique of Section 4.5, we construct the compositional model, which is a composition of the root automaton of the previous step and three duration automata (presented in the bottom of Figures 4.10–4.12). We performed the same experiments on the compositional model that we performed on the concrete model (in Chapter 3), the abstract model (in Chapter 3), and the monolithic model (in the previous step). Table 4.2 shows that the compositional model produces a much smaller state space than the monolithic model.

Experimental results of the monolithic and compositional models show:

1. Abstraction improves the scalability dramatically for every configuration of Table 4.1.

Configurations of Table 4.1	CFL	Comparison			
		monolithic model		compositional model	
		time	size	time	size
C1	2	No controller exists			
	1	39.08	72608	0.09	76
C2	2	No controller exists			
	1	71.41	83971	0.09	76
C3	2	No controller exists			
	1	Out of Memory		0.12	136
C4	2	Out of memory		0.19	439
	1			0.08	154
C5	2	Out of memory		0.19	439
	1			0.08	154
C6	2	No controller exists			
	1				
C7	2	Out of memory		0.20	439
	1			0.11	154
C8	2	Out of memory		0.14	278
	1	95.76	106960	0.09	101
C9	2	Out of memory		0.15	346
	1			0.10	124
C10	2	Out of memory		64.60	18321
	1			22.53	5868
C11	2	Out of memory		5.05	5517
	1			1.87	1783
C12	2	Out of memory		3.18	4124
	1			1.19	1338
C13	2	Out of memory		3.18	4124
	1			1.19	1338
C14	2	Out of memory		0.20	439
	1	78.12	118477	0.11	154
C15	2	Out of memory		0.21	530
	1	47.30	77548	0.13	183
C16	2	Out of memory		0.21	530
	1			0.13	183
C17	2	Out of memory		0.20	462
	1	59.26	109982	0.13	161
C18	2	Out of memory		0.20	453
	1	50.29	73914	0.12	158
C19	2	Out of memory		0.21	540
	1	45.14	84370	0.13	186
C20	2	94.07	179479	0.26	633
	1	34.14	63791	0.15	216

Table 4.2: Comparisons of the monolithic and compositional models with respect to controller synthesis time (in seconds) and the strategy size (in kilobytes)

Experiments for different configurations for the same system revealed that speed up of two orders of magnitude is possible with the compositional technique, while maintaining enough precision. The size of composition in the monolithic analysis is exponential in the depth of the hierarchy, due to a product construction (and it is also linear in the multiplication of sizes of all included standalone automata). In the compositional analysis, the depth of the hierarchy is constant (only two layers) and we only take a product of one root automaton with several constant size duration automata; this explains why the obtained speed-ups are so dramatic. The efficiency gains are primarily due to the coarse abstraction of safety and reachability properties of an arbitrarily large callee into a tiny duration automaton. Abstraction and compositional reasoning together might provide similar speed ups for timed I/O automata in Chapter 3; and the restrictions that timed process automata impose on models allow one to automate the procedure.

2. For the monolithic models, the larger the difference between WCET and BECT the longer the analysis time, and the sparser the strategy, for example, configuration C1 versus configuration C2, C7 versus C14, and C15 versus C16. Unlike the other models, differences between the WCET and the corresponding BCET in the compositional model has no impact on the controller synthesis time or on the strategy size—for example, C1 versus C2, C7 versus C14, C12 versus C13, and C15 versus C16—because duration automata do not keep details regarding the best-case execution times.

3. The smaller the least common multiples of release periods the smaller state space, the shorter analysis time, and the more compact strategy, for instance, C2 versus C3, C8 versus C9, C9 versus C10, C10 versus C11, and so forth.

4. The least common multiples of the execution times have no clear impact on the analysis time or the size of the strategy, for example, C14 versus C15, C15 versus C17, C17 versus C18, C18 versus C19, C19 versus C20, and so forth.

5. Variations in the least common denominator of non-clock variables, such as different loads, do not have any significant impact on the analysis, for instance, C4 versus C5 and C5 versus C7.

6. Uppaal Tiga takes less time and generates a smaller strategy for a higher value for CFL, for instance, configurations C4, C5, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, and C20.

Observations in the above match with the observations presented in Section 3.5. Depending on controller synthesis times or strategy sizes of Table 3.2 and Table 4.2, the following two relationships clearly hold: abstract $\leq$ monolithic $\leq$ concrete and compositional $\leq$ monolithic $\leq$ concrete. However, analyses results do not exhibit such explicit relationship between the abstract model and the compositional model. The compositional model has better outcomes than the abstract model for most of the cases, for example, configurations C2 (for CFL 2), C3 (for CFL 2), C4, C5, C7, C8, C9, C10, C11, C12, C14, C16, C17 (for CFL 2), and C18.

4.7 Discussion

Classical timed automata [14, 15] and timed I/O automata [4, 95] have explicit modeling support only for static non-hierarchical structures. In 2011, we identified and classified eighty variants of timed automata into eleven classes in Chapter 2 [230] and there may be many more. Timed process automata fall in the class of *timed automata with resources*

[230] because of their ability to model dynamic behaviors, which is required when resource constraints do not permit one to activate all the components at the same time. More precisely, the model is a direct generalization of *task automata* [121], *dynamic networks of timed automata* [77], and *callable timed automata* [58]. These three variants model only closed systems, while timed process automata can model both closed and open systems. Task automata model only two layers (a scheduler and its tasks) of hierarchy, while timed process automata, dynamic networks of timed automata [77], and callable timed automata are able to model any numbers of hierarchies. Unlike timed process automata, none of them supports private communication, provides compositional modeling with reusable designs for different contexts, or supports automated state-space reduction technique.

Dynamic networks of continuous-time automata have also been studied in the context of hybrid automata [128, 96]. These works model physical environments using differential equations, which restrict the kinds of environments that can be described. In practice, large differential equations make analyses unmanageable, or can only give statistical guarantees [96]. These works focus on system dynamics, and do not support private communication. Timed process automata can be considered as a member of the class of *timed automata with succinctness* [230] because they hide many design details from the designers to achieve succinctness (like *timed automata variants with urgency* [56, 33, 230]). Timed process automata are also timed game automata [190, 100, 4, 95] because the new variant uses timed games for analysis.

Chapter 5

Conclusions

We have developed an approach compositional modeling with reuse and an automatable-state-space reduction technique for timed games-based analysis of dynamic hierarchical open systems, which are common in practice. The development can be divided into five sequential phases:

1. Studying background in Chapter 2,
2. Pioneering automatable synthesis of reconfiguration services in Chapter 3,
3. Developing an abstraction-based manual state-space reduction technique for timed I/O automata-based analysis in Chapter 3,
4. Introducing an approach for compositional modeling with reuse in Chapter 4, and
5. Developing an abstraction-based automatable state-space reduction technique for timed process automata-based analysis in Chapter 4.

5.1 Summary

We have presented a survey on semantics, decision problems, variants, implementability, and tools of timed automata in Chapter 2. Section 2.2.2 is interesting as it informs how symbolic semantics-based analysis has evolved with time to make timed automata more suitable for practical uses such as tool development. There are many data structures, such as *CDD* [179], *CDR* [235], *NDD* [26], other than DBM [41, 44, 106] for symbolic semantics of timed automata; however, our survey did not explore these data structures. We have listed major linguistic properties and decision problems of classical timed automata in Section 2.3. Decidability of emptiness checking and the undecidability of inclusion problem are the two major results of this survey. When we started working in this area we first thought that timed automata would have at most thirty variants, and then after performing this extensive survey we came to realize that timed automata had many more variants. We have listed around eighty variants in Section 2.4. We believe there are more variants of timed automata that exist, and that the number is increasing with time. No previous survey on timed automata exists which lists at least twenty variants. We also classify these variants into eleven classes depending on their construction and functionality. This classification will help a reader to understand similarities and differences among timed automata variants. This survey discusses only major variants of each class. We hope these discussions on major variants will give a reader a rough idea of other variants of the same class. An interested reader can learn more about an undescribed (but listed) variant from its related citation. Section 2.5 identifies, describes, and classifies forty timed automata-based research and academic tools.

In Chapter 3, we have presented the synthesis process using a mixed-criticality AMP system having a fault-intolerant criticality-unaware scheduler with fixed allocation. This

includes two different design principles to model the problem using timed games, based on a selection of simplifications and abstractions. We compared the models for scalability, showing that solving the problem using strategy synthesis for timed games is feasible. We have observed that reducing action based synchronization, the state space, and especially shared states, improves efficiency of algorithms. Our reconfiguration services are distributed, and the synthesis process applies to mixed-criticality systems, both in symmetric and asymmetric scenarios. We demonstrated this on a case study from the automotive domain. This is the first case study applying timed games to the synthesis reconfiguration services for fault-tolerance.

In Chapter 4, we have presented timed process automata that captures dynamic activation and deactivation of continuous-time control processes and private communication among the active processes. We have provided a safety and reachability analysis technique for non-recursive well-formed timed process automata. We have also designed an abstraction- and compositional reasoning-based state-space reduction technique for automated analysis of large industrial systems. Our analysis techniques can be applied in practice using any standard timed games solver such as Uppaal Tiga [35] and Synthia [118].

Timed process automata can model private communication and open systems. Moreover, timed process automata provide two important features for industrial dynamic open time-critical systems development: (i) compositional modeling with reusable designs for different contexts and (ii) automated state-space reduction technique.

5.2 Limitations and Future Works

Timed automata have huge potential to be a prominent industrial model. Unfortunately, they are barely used in practice. Use of timed automata in the industry will be widespread

if researchers can triumph over timed automata's state-space explosion problem and timed automata's realizability problem. Section 2.2.2 briefly discusses symbolic semantics which is only one of the frontiers in the war on state-space explosion. Accurate timed automata implementability is getting more attention every day. Usually robustness analysis introduces larger state-spaces for example, Figure 3 of [177]. A study on the comparison and relation between these two problems—state-space explosion and robust analysis—of timed automata would be an interesting work for the research community. Our strong involvement with (automotive) industry and long experience in timed automata helped us to understand that state-space explosion is the biggest obstacle for timed automata. After our development of an automatable state-space reduction, the main challenge for timed (game) automata, therefore, is to improve computational efficiency of their symbolic semantics and data structures in a way that their computational complexity should be almost as expensive as their discrete-time counterpart.

In only two decades the theory of timed automata has established itself as an integral part of dense-time-based analyses. This area is becoming more and more active. Our survey in Chapter 2 is only a snapshot of this area. The main motivation of this survey was to provide a coherent picture of this scattered arena. This survey did not discuss real-time temporal logics, real-time formal verification, and real-time controller synthesis because these topics are mostly related to real-time formal models in general instead of being specific to timed automata. A rigorous survey on timed automata-based tools including case studies, performance analyses, and commercial tools would be a good step to convince people to increase the use of timed automata in industry.

Section 3.6 discusses generalization of our reconfiguration synthesis technique of Chapter 3 to several other scenarios. We did not implement our technique for those scenarios

Configurations of Table 3.2 or 4.1	CFL	Comparison							
		concrete model		abstract model		monolithic model		compositional model	
		time	size	time	size	time	size	time	size
C1	2	No controller exists							
	1	94.20	290663	0.08	73	39.08	72608	0.09	76
C2	2	No controller exists							
	1	115.71	296524	0.11	107	71.41	83971	0.09	76
C3	2	No controller exists							
	1	Out of memory		0.14	242	Out of Memory		0.12	136
C4	2	Out of memory		0.25	712	Out of memory		0.19	439
	1			0.14	266			0.08	154
C5	2	Out of memory		0.25	712	Out of memory		0.19	439
	1			0.14	266			0.08	154
C6	2	No controller exists							
	1								
C7	2	Out of memory		0.25	712	Out of memory		0.20	439
	1			0.14	266			0.11	154
C8	2	Out of memory		0.15	420	Out of memory		0.14	278
	1			0.11	159	95.76	106960	0.09	101
C9	2	Out of memory		0.22	632	Out of memory		0.15	346
	1			0.14	234			0.10	124
C10	2	Out of memory		178.54	40668	Out of memory		64.60	18321
	1			73.32	14647			22.53	5868
C11	2	Out of memory		4.91	6274	Out of memory		5.05	5517
	1			1.65	2277			1.87	1783
C12	2	Out of memory		4.07	6272	Out of memory		3.18	4124
	1			1.65	2275			1.19	1338
C13	2	Out of memory		1.93	3639	Out of memory		3.18	4124
	1			0.81	1332			1.19	1338
C14	2	Out of memory		0.20	539	Out of memory		0.20	439
	1			0.14	204	78.12	118477	0.11	154
C15	2	Out of memory		0.15	431	Out of memory		0.21	530
	1			0.11	164	47.30	77548	0.13	183
C16	2	Out of memory		0.24	718	Out of memory		0.21	530
	1			0.14	270			0.13	183
C17	2	Out of memory		0.16	458	Out of memory		0.20	462
	1			0.12	173	59.26	109982	0.13	161
C18	2	Out of memory		0.16	485	Out of memory		0.20	453
	1			0.10	184	50.29	73914	0.12	158
C19	2	Out of memory		0.14	406	Out of memory		0.21	540
	1			0.10	154	45.14	84370	0.13	186
C20	2	Out of memory		0.14	358	94.07	179479	0.26	633
	1			0.09	135	34.14	63791	0.15	216

Table 5.1: Comparisons of the concrete, abstract, monolithic and compositional models with respect to synthesis time (in seconds) and the strategy size (in kilobytes)

but we are interested to know others' experience for those scenarios if they follow our generalization.

Timed process automata allow compositional modeling with reuse by using channel-based dynamic renaming. One may explore this renaming process for other types of timed or hybrid or untimed automata to develop compositional modeling with reuse for the respective automata. One limitations for our compositional modeling with reuse is it handles only three representations mentioned in Section 1.1.2. We, however, do not know other design aspects for which manual design alterations can be replaced by automated techniques. Investigating numerous large industrial models and surveying modeling experts might help one to find other design aspects that can be automated.Such type of investigation may also provide evidence that compositional modeling with reuse of timed process automata reduces modeling errors in practice. Findings of these investigations may encourage researchers to extend (timed process or other) automata's capability for compositional modeling with reuse.

Timed process automata facilitate automatable state-space reduction technique for timed games-based analysis of dynamic hierarchical systems. Theoretically manual state-space reduction may achieve similar or smaller state-spaces than automated state-space reduction. Even practically it is usually true for smaller systems for example, the comparisons in Table 5.1. However, efficiency of automated state-space reduction increases with depth of the control hierarchies in practice. Dynamic hierarchical systems with deep control hierarchies make up a small portion of all types of systems. Therefore, automated state-space reduction techniques for standard timed I/O automata are much more important and desirable than automated state-space reduction techniques for timed process automata. We strongly encourage researchers to develop an automated state-space reduction technique

for standard timed I/O automata. A similar but larger challenge is to develop a general automated state-space reduction technique for all types of timed automata.

This thesis considers only location-based safety and location-based reachability properties of timed process automata. Investigation of other types of properties—including more general safety and reachability properties—may produce interesting outcomes. We use simple abstract model duration automata for our automatable state-space reduction technique. Others may prefer to use different kinds of abstract models for this purpose. Even for some scenarios or properties our duration automata might be too abstract to analyze. One may consider other state-space reduction techniques for timed process automata. For example, compositional model reduction of *discrete time systems (DES)* has been done by generalizing observers for deterministic DES to nondeterministic DES and characterizing using the join semilattice of compatible partitions of a transition system to achieve efficient algorithms [183, 184].

It would be interesting to consider a model transformation from a subset of the *real-time π-calculus* [203, 31] to timed process automata. This transformation might enable controllability analysis of π-calculus for open systems. The converse reduction from timed process automata to real-time π-calculus could also give several advantages: understanding timed process automata semantics in terms of the well-established π-calculus formalism, access to tools developed for real-time π-calculus [203], which might permit the analysis of recursive processes; it would also give a familiar automata-like syntax to π-calculus formalisms. It would also be relevant to minimize the number of subprocesses in controller synthesis. One may consider synthesis under this objective in the future, possibly by reduction to *priced/weighted timed automata* [22, 38].

Bibliography

[1] IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-1995*, 1996.

[2] Luca Aceto and François Laroussinie. Is your model checker on time? On the complexity of model checking for timed modal logics. *Journal of Logic and Algebraic Programming*, 52-53:7–51, 2002.

[3] S. Akshay, Benedikt Bollig, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. In *Proceedings of the 19th International Conference on Concurrency Theory*, CONCUR '08, pages 82–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 108–122, London, UK, 2002. Springer-Verlag.

[5] Alejandra Alfonso, Víctor Braberman, Diego Garbervetsky, Nicolas Kicillof, Alfredo Olivero, and Fernando Schapachnik. VInTiMe: Combining high-level finesse with low-level muscle to verify real-time systems. In *Proceeding of the 1st International Conference on Principles of Software Engineering*, October 2004.

[6] Alejandra Alfonso, Víctor Braberman, Nicolas Kicillof, and Alfredo Olivero. Visual timed event scenarios. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In *Proceedings of the 2nd Workshop on Real-Time Tools*, July 2002.

[8] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer Berlin Heidelberg, 1999.

[9] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for probabilistic real-time systems (extended abstract). In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pages 115–126, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[10] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, May 1993.

[11] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, David L. Dill, and Howard Wong-Toi. Minimization of timed transition systems. In W. R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 340–354. Springer Berlin Heidelberg, 1992.

[12] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. The observational power of clocks. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94:*

Concurrency Theory, volume 836 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg.

[13] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Berlin Heidelberg, 1993.

[14] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

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

[16] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211:253–273, January 1999.

[17] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, STOC '93, pages 592–601, New York, NY, USA, 1993. ACM.

[18] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *Proceedings of the 26th International Colloquium on*

Automata, Languages and Programming, ICAL '99, pages 169–178, London, UK, 1999. Springer-Verlag.

[19] Rajeev Alur, Salvatore La Torre, and Parthasarathy Madhusudan. Perturbed timed automata. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 70–85. Springer Berlin Heidelberg, 2005.

[20] Rajeev Alur and Parthasarathy Madhusudan. Decision problems for timed automata: A survey. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems — Revised Lectures of the International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT'04)*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2004.

[21] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[22] Rajeev Alur, Salvatore La Torre, and George J. Pappas. Optimal paths in weighted timed automata. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, HSCC '01, pages 49–62, London, UK, 2001. Springer-Verlag.

[23] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - a tool for modelling and implementation of embedded systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 460–464, London, UK, 2002.

Springer-Verlag.

[24] Étienne André. IMITATOR II: A tool for solving the good parameters problem in timed automata. In *Proceedings 12th International Workshop on Verification of Infinite-State Systems*, pages 91–99, 2010.

[25] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer Berlin Heidelberg, 2001.

[26] Eugene Asarin, Marius Bozga, Alain Kerbrat, Oded Maler, Amir Pnueli, and Anne Rasse. Data-structures for the verification of timed automata. In Oded Maler, editor, *Proceedings of the 1997 International Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, 1997.

[27] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[28] Eugene Asarin and Oded Maler. As soon as possible: Time optimal control for timed automata. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin Heidelberg, 1999.

[29] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proceedings of the 5th IFAC Conference on System Structure and Control (SSC'98)*, pages 469–474. Elsevier Science, July 1998.

[30] Christel Baier, Nathalie Bertrand, Patricia Bouyer, Thomas Brihaye, and Marcus Größer. Probabilistic and topological semantics for timed automata. In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 179–191. Springer Berlin Heidelberg, 2007.

[31] Kamal Barakat, Stefan Kowalewski, and Thomas Noll. A native approach to modeling timed behavior in the Pi-calculus. In *6th International Symposium on Theoretical Aspects of Software Engineering*, pages 253–256, July 2012.

[32] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, and Luca Tesei. Timed P automata. *Electronic Notes Theoretical Computer Science*, 227:21–36, January 2009.

[33] Roberto Barbuti and Luca Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40:317–347, March 2004.

[34] Danièle Beauquier. On probabilistic timed automata. *Theoretical Computer Science*, 292:65–84, January 2003.

[35] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Li Didier. UPPAAL-Tiga: Time for playing games! In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer Berlin Heidelberg, 2007.

[36] Gerd Behrmann, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Software: Practice and Experience*, 41(2):133–142, 2011.

[37] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in uppaal. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2001.

[38] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, HSCC '01, pages 147–161, London, UK, 2001. Springer-Verlag.

[39] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS - Performance Evaluation Review*, 32:34–40, March 2005.

[40] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Priced timed automata: Algorithms and applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 162–182. Springer Berlin Heidelberg, 2005.

[41] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[42] Leon Bendiksen and Peter C. Ölveczky. The priced-timed Maude tool. In *Proceedings of the 3rd International Conference on Algebra and Coalgebra in Computer Science*, CALCO'09, pages 443–448, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Massimo Benerecetti, Stefano Minopoli, and Adriano Peron. Analysis of timed recursive state machines. September.

[44] Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Department of Information Technology, Uppsala University, Uppsala, Sweeden, 2002.

[45] Johan Bengtsson and Wang Yi. On clock difference constraints and termination in reachability analysis of timed automata. In JinSong Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering*, volume 2885 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2003.

[46] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004.

[47] Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1-2):1–7, July 2000.

[48] Béatrice Bérard, Paul Gastin, and Antoine Petit. On the power of non-observable actions in timed automata. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '96, pages 257–268, London, UK, 1996. Springer-Verlag.

[49] Béatrice Bérard, Serge Haddad, and Mathieu Sassolas. Interrupt timed automata: Verification and expressiveness. *Formal Methods in System Design*, 40(1):41–87, 2012.

[50] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36:145–182, August 1998.

[51] Jasper Berendsen, David N. Jansen, and Joost-Pieter Katoen. Probably on time and within budget: On reachability in priced probabilistic timed automata. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, pages 311–322, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Jasper Berendsen, David N. Jansen, and Frits W. Vaandrager. Fortuna: Model checking priced probabilistic timed automata. In *Proceedings of the 7th International Conference on the Quantitative Evaluation of Systems*, pages 273–281, September 2010.

[53] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for bdd-based verification of real-time systems. In Jr. Hunt, Warren A. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 122–125. Springer Berlin Heidelberg, 2003.

[54] Dirk Beyer and Heinrich Rust. Cottbus timed automata: Formal definition and semantics. In *Proceedings of the 2nd IEEE/IFIP Joint Workshop on Formal Specifications of Computer-Based Systems*, pages 75–87, 2001.

[55] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, third edition, 2007.

[56] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer Berlin Heidelberg, 1998.

[57] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In *Proceedings of the 18th IEEE Symposium on Real-Time Systems*, RTSS '97, pages 25–34, Washington, DC, USA, December 1997. IEEE Computer Society Press.

[58] Abdeldjalil Boudjadar, Frits Vaandrager, Jean-Paul Bodeveix, and Mamoun Filali. Extending UPPAAL for the modeling and verification of dynamic real-time systems. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, Lecture Notes in Computer Science, pages 111–132. Springer Berlin Heidelberg, 2013.

[59] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24:281–320, May 2004.

[60] Patricia Bouyer, Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. On the optimal reachability problem of weighted timed automata. *Formal Methods in System Design*, 31:135–175, October 2007.

[61] Patricia Bouyer, Franck Cassez, Emmanuel Fleury, and Kim G. Larsen. Optimal Strategies in Priced Timed Game Automata. In Kamal Lodaya and Meena Mahajan, editors, *Proceedings of the 24th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *Lecture Notes in Computer Science*, pages 148–160, Chennai, India, December 2004. Springer.

[62] Patricia Bouyer and Fabrice Chevalier. On conciseness of extensions of timed automata. *Journal of Automata, Languages and Combinatorics*, 10:393–405, April 2005.

[63] Patricia Bouyer, Fabrice Chevalier, and Deepak D'Souza. Fault diagnosis using timed automata. In Vladimiro Sassone, editor, *Foundations of Software Science and Computational Structures*, volume 3441 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin Heidelberg, 2005.

[64] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theoretical Computer Science*, 321:291–345, August 2004.

[65] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, and Nicolas Markey. Quantitative analysis of real-time systems using priced timed automata. *Communications of the ACM*, 54:78–87, September 2011.

[66] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'05)*, volume 3829 of *Lecture Notes in Computer Science*, pages 112–126, Uppsala, Sweden, 2005. Springer Berlin Heidelberg.

[67] Patricia Bouyer, Nicolas Markey, Joël Ouaknine, and James Worrell. On expressiveness and complexity in real-time model checking. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 124–135, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] Patricia Bouyer and Antoine Petit. A Kleene/Büchi-like theorem for clock languages. *Journal of Automata, Languages and Combinatorics*, 7(2):167–186, 2001.

[69] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer Berlin Heidelberg, 2004.

[70] Laura Bozzelli and Salvatore La Torre. Decision problems for lower/upper bound parametric timed automata. *Formal Methods in System Design*, 35:121–151, October 2009.

[71] Víctor Braberman, Diego Garbervetsky, and Alfredo Olivero. ObsSlice: A timed automata slicer based on observers. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 470–474. Springer Berlin Heidelberg, 2004.

[72] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Issues in distributed timed model checking: Building Zeus. *International Journal on Software Tools for Technology Transfer*, 7:4–18, February 2005.

[73] Victor A. Braberman and Miguel Felder. Verification of real-time designs: Combining scheduling theory with automatic formal verification. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering—ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 494–510. Springer Berlin Heidelberg, 1999.

[74] Thomas Brihaye, Thomas A. Henzinger, Vinayak S. Prabhu, and Jean-François Raskin. Minimum-time reachability in timed games. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 825–837. Springer Berlin Heidelberg, 2007.

[75] María-Emilia Cambronero, Gregorio Díaz, Valentin Valero, and Enrique Martínez. Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[76] Salvatore Campana, Luca Spalazzi, and Francesco Spegni. XAL: A web oriented programming language based on timed-automata. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, pages 862–868, Washington, DC, USA, 2008. IEEE Computer Society.

[77] Salvatore Campana, Luca Spalazzi, and Francesco Spegni. Dynamic networks of timed automata for collaborative systems: A network monitoring case study. In *2010 International Symposium on Collaborative Technologies and Systems*, pages 113–122, May 2010.

[78] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1:1–193, January 2006.

[79] Franck Cassez. Timed games for computing WCET for pipelined processors with caches. In *Proceedings of the 2011 Eleventh International Conference on Application of Concurrency to System Design*, ACSD '11, pages 195–204, Washington, DC,

USA, 2011. IEEE Computer Society.

[80] Franck Cassez and Kim G. Larsen. The impressive power of stopwatches. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR '00, pages 138–152, London, UK, 2000. Springer-Verlag.

[81] Franck Cassez and Nicolas Markey. *Communicating Embedded Systems – Software and Design*, chapter Control of Timed Systems, pages 83–120. 2009.

[82] Matteo Cavaliere and Dragoş Sburlan. Time-independent P systems. In *Proceedings of the 5th International Conference on Membrane Computing*, WMC'04, pages 239–258, Berlin, Heidelberg, 2005. Springer-Verlag.

[83] Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *Proceedings of the 4th International Workshop on Computer Aided Verification*, pages 302–315, London, UK, 1993. Springer-Verlag.

[84] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, January 1981.

[85] Fabrice Chevalier, Deepak D'Souza, and Pavithra Prabhakar. On continuous timed automata with input-determined guards. In *Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'06, pages 369–380, Berlin, Heidelberg, 2006. Springer-Verlag.

[86] Fabrice Chevalier, Deepak D'Souza, and Pavithra Prabhakar. Counter-free input-determined timed automata. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '07, pages 82–97, Berlin, Heidelberg, 2007. Springer-Verlag.

[87] Christian Choffrut and Massimiliano Goldwurm. Timed automata with periodic clock constraints. *Journal of Automata, Languages and Combinatorics*, 5:371–403, October 2000.

[88] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, Daniel Weil, and Sergio Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 391–395, London, UK, 2001. Springer-Verlag.

[89] Hubert Comon and Yan Jurski. Timed automata and the theory of real numbers. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 242–257, London, UK, 1999. Springer-Verlag.

[90] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1:385–415, December 1992.

[91] Andreas E. Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads C. Olesen, and Jaco van de Pol. Multi-core reachability for timed automata. In Marcin Jurdziński and Dejan Ničković, editors, *Formal Modeling and Analysis of Timed Systems*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106. Springer Berlin Heidelberg, 2012.

[92] Zhe Dang. Pushdown timed automata: a binary reachability characterization and safety verification. *Theoretical Computer Science*, 302:93–121, June 2003.

[93] Zhe Dang, Tevfik Bultan, Oscar H. Ibarra, and Richard A. Kemmerer. Past pushdown timed automata and safety verification. *Theoretical Computer Science*, 313:57–71, February 2004.

[94] Alexandre David, Jacob D. Grunnet, Jan J. Jessen, Kim G. Larsen, and Jacob I. Rasmussen. Application of model-checking technology to controller synthesis. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 336–351. Springer Berlin Heidelberg, 2012.

[95] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wąsowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[96] Alexandre David, Kim G. Larsen, Axel Legay, and Danny Bøgsted Poulsen. Statistical model checking of dynamic networks of stochastic hybrid automata. In Steve Schneider and Helen Treharne, editors, *Proceedings of the 13th International Workshop on Automated Verification of Critical Systems*, volume 10 of *Electronic Communications of the EASST*, Guildford, UK, 2013. EASST.

[97] Jennifer M. Davoren and Anil Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88(7):985–1010, July 2000.

[98] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and SergioYovine. The tool Kronos. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid*

Systems III, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 1996.

[99] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 313–329, London, UK, 1998. Springer-Verlag.

[100] Luca de Alfaro, Marco Faella, Thomas A. Henzinger, Rupak Majumdar, and Mariëlle Stoelinga. The element of surprise in timed games. In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 144–158. Springer Berlin Heidelberg, 2003.

[101] Martin De Wulf. *From Timed Models to Timed Implementations*. Thèse de doctorat, Département d'Informatique, Université Libre de Bruxelles, Belgium, December 2006.

[102] François Demichelis and Wieslaw Zielonka. Controlled timed automata. In *Proceedings of the 9th International Conference on Concurrency Theory*, CONCUR '98, pages 455–469, London, UK, 1998. Springer-Verlag.

[103] Martin Dickhöfer and Thomas Wilke. Timed alternating tree automata: The automata-theoretic solution to the TCTL model checking problem. In Jiří Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 281–290. Springer Berlin Heidelberg, 1999.

[104] Jean-Yves Didier, Bachir Djafri, and Hanna Klaudel. The mirela framework: modeling and analyzing mixed reality applications using timed automata. *Journal of Virtual Reality and Broadcasting*, 6(1), February 2009.

[105] Volker Diekert, Paul Gastin, and Antoine Petit. Removing epsilon-transitions in timed automata. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '97, pages 583–594, London, UK, 1997. Springer-Verlag.

[106] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[107] Cătălin Dima. Kleene theorems for event-clock automata. In Gabriel Ciobanu and Gheorghe Păun, editors, *Fundamentals of Computation Theory*, volume 1684 of *Lecture Notes in Computer Science*, pages 215–225. Springer Berlin Heidelberg, 1999.

[108] Cătălin Dima. Regular expressions with timed dominoes. In *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science*, DMTCS'03, pages 141–154, Berlin, Heidelberg, 2003. Springer-Verlag.

[109] Cătălin Dima. Timed shuffle expressions. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005–Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin Heidelberg, 2005.

[110] Cătălin Dima and Ruggero Lanotte. Distributed time-asynchronous automata. In *Proceedings of the 4th International Conference on Theoretical Aspects of Computing*, ICTAC'07, pages 185–200, Berlin, Heidelberg, 2007. Springer-Verlag.

[111] Cătălin Dima and Ruggero Lanotte. Removing all silent transitions from timed automata. In Joél Ouaknine and Frits W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems*, volume 5813 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 2009.

[112] Doron Drusinsky and David Harel. On the power of bounded concurrency i: Finite automata. *Journal of the ACM*, 41:517–539, May 1994.

[113] Deepak D'Souza. A logical characterisation of event recording automata. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 240–251. Springer Berlin Heidelberg, 2000.

[114] Deepak D'Souza. A logical characterisation of event clock automata. *International Journal Foundations Computer Science*, 14(4):625–640, 2003.

[115] Deepak D'Souza and M. Raj Mohan. Eventual timed automata. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 322–334. Springer Berlin Heidelberg, 2005.

[116] Deepak D'Souza and Nicolas Tabareau. On timed automata with input-determined guards. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture*

Notes in Computer Science, pages 68–83. Springer Berlin Heidelberg, 2004.

[117] Deepak D'Souza and P. S. Thiagarajan. Product interval automata: A subclass of timed automata. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg, 1999.

[118] Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter. Synthia: Verification and synthesis for timed automata. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 649–655. Springer Berlin Heidelberg, 2011.

[119] Michael Emmi and Rupak Majumdar. Decision problems for the verification of real-time software. In João P. Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control*, volume 3927 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin Heidelberg, 2006.

[120] Abdelaziz Fellah and Soufiane Noureddine. Some succinctness properties of ω-DTAFA. In *Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, SEPADS'06, pages 97–103, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society.

[121] Elena Fersman, Pavel Krčál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *International Journal of Information and Computation*, 205:1149–1172, August 2007.

[122] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 67–82, London, UK, UK, 2002. Springer-Verlag.

[123] Arnaud Fietzke, Holger Hermanns, and Christoph Weidenbach. Superposition-based analysis of first-order probabilistic timed automata. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 302–316, Berlin, Heidelberg, 2010. Springer-Verlag.

[124] Olivier Finkel. On the shuffle of timed regular languages. *Bulletin of the European Association for Theoretical Computer Science*, 88:182–184, February 2006.

[125] Olivier Finkel. Undecidable problems about timed automata. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 187–199. Springer Berlin Heidelberg, 2006.

[126] Martin Fränzle. What will be eventually true of polynomial hybrid automata? In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 340–359. Springer Berlin Heidelberg, 2001.

[127] Ronojoy Ghosh and Claire Tomlin. Symbolic reachable set computation of piecewise affine hybrid automata and its application to biological modelling: Delta-notch protein signalling. *Systems Biology*, 1(1):170–183, June 2004.

[128] Aleks Göllü and Pravin Varaiya. A dynamic network of hybrid automata. In *5th annual conference on AI, simulation, and planning in high autonomy systems*, pages 244–251, 1994.

[129] Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In Oded Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 1997.

[130] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards wcet analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis*, volume 15 of *OASIcs*, pages 101–112, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[131] Niusha Hakimipour, Paul Strooper, and Andy Wellings. TART: Timed-automata to real-time Java tool. In José L. Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini, editors, *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 299–309, Washington, DC, USA, 2010. IEEE Computer Society.

[132] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[133] Arnd Hartmanns and Holger Hermanns. A modest approach to checking probabilistic timed automata. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems*, QEST '09, pages 187–196, Washington, DC, USA, 2009. IEEE Computer Society.

[134] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–292, Washington, DC, USA, 1996. IEEE Computer Society.

[135] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer Berlin Heidelberg, 1997.

[136] Thomas A. Henzinger and Peter W. Kopke. State equivalences for rectangular hybrid automata. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 530–545. Springer Berlin Heidelberg, 1996.

[137] Thomas A. Henzinger and Peter W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221:369–392, June 1999.

[138] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, STOC '95, pages 373–382, New York, NY, USA, 1995. ACM.

[139] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer Berlin Heidelberg, 1992.

[140] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1994.

[141] Thomas A. Henzinger, Jean-François Raskin, and Pierre-Yves Schobbens. The regular real-time languages. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 580–591. Springer Berlin Heidelberg, 1998.

[142] Anders Hessel and Paul Pettersson. CoVer - a real-time test case generation tool. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, 2007.

[143] Jochen Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.

[144] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-based design and analysis of embedded systems with UPPAAL PORT. In Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Automated Technology for Verification and Analysis*, volume 5311 of *Lecture Notes in Computer Science*, pages 252–257. Springer Berlin Heidelberg, 2008.

[145] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. *The Journal of Logic and Algebraic Programming*, 52–53(0):183 – 220, 2002.

[146] Inseok Hwang, Sungwan Kim, Youdan Kim, and Chze E. Seah. A survey of fault detection, isolation, and reconfiguration methods. *IEEE Transactions on Control Systems Technology*, 18(3):636–653, May 2010.

[147] Oscar H. Ibarra, Zhe Dang, and Pierluigi S. Pietro. Verification in loosely synchronous queue-connected discrete timed automata. *Theoretical Computer Science*, 290:1713–1735, January 2003.

[148] Dinko Ivanov, Marin Orlić, Cristina Seceleanu, and Aneta Vulgarakis. REMES toolchain: A set of integrated tools for behavioral modeling and analysis of embedded systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 361–362, New York, NY, USA, 2010. ACM.

[149] Mark Jenkins, Joel Ouaknine, Alexander Rabinovich, and James Worrell. Alternating timed automata over bounded time. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, LICS '10, pages 60–69, Washington, DC, USA, 2010. IEEE Computer Society.

[150] Henrik Ejersbo Jensen. Model checking probabilistic real time systems. In B. Bjerner, M. Larsson, and B. Nordström, editors, *Proceedings of the 7th Nordic Workshop on Programming Theory*, Report 86, pages 247–261. Chalmers Institute of Technology, 1996.

[151] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977.

[152] Marcin Jurdziński, Marta Kwiatkowska, Gethin Norman, and Ashutosh Trivedi. Concavely-priced probabilistic timed automata. In Mario Bravetti and Gianluigi

Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*, pages 415–430. Springer Berlin Heidelberg, 2009.

[153] Marcin Jurdziński, François Laroussinie, and Jeremy Sproston. Model checking probabilistic timed automata with one or two clocks. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 170–184, Berlin, Heidelberg, 2007. Springer-Verlag.

[154] Marcin Jurdzinski and Ashutosh Trivedi. Reachability-time games on timed automata. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, pages 838–849, 2007.

[155] Marcin Jurdziński and Ashutosh Trivedi. Concavely-priced timed automata. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2008.

[156] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

[157] Michal Knapik, Artur Niewiadomski, Wojciech Penczek, Agata Pólrola, Maciej Szreter, and Andrzej Zbrzezny. Parametric model checking with VerICS. *Transactions on Petri Nets and Other Models of Concurrency*, 4:98–120, 2010.

[158] Pavel Krcál and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In Thomas Ball and Robert B. Jones, editors, *Computer*

Aided Verification, volume 4144 of *Lecture Notes in Computer Science*, pages 249–262. Springer Berlin Heidelberg, 2006.

[159] Saul Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[160] Pavel Krčál, Leonid Mokrushin, and Wang Yi. A tool for compositional analysis of timed systems by abstraction (extended abstract). In Einar B. Johnsen, Olaf Owe, and Gerardo Schneider, editors, *Proceedings of the 19th Nordic Workshop on Programming Theory*, 2007.

[161] Pavel Krčál, Martin Stigge, and Wang Yi. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In Jean-François Raskin and P. S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4763 of *Lecture Notes in Computer Science*, pages 274–289. Springer Berlin Heidelberg, 2007.

[162] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 2004.

[163] Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than UPPAAL? In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 552–555, Berlin, Heidelberg, 2008. Springer-Verlag.

[164] Pavel Kučera, Ondřej Hynčica, and Petr Honzík. Implementation of timed automata in a real-time operating system. In *Proceedings of World Congress on Engineering and Computer Science*, volume I, pages 56–60, October 2010.

[165] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer Berlin Heidelberg, 2011.

[166] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 75–95. Springer Berlin Heidelberg, 1999.

[167] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In Catuscia Palamidessi, editor, *CONCUR 2000 — Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 123–137. Springer Berlin Heidelberg, 2000.

[168] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Verifying soft deadlines with probabilistic timed automata. In *Proceedings of the Workshop on Advances in Verification (WAVe 2000)*, July 2000.

[169] Gino Labinaz, Mohamed M. Bayoumi, and Karen Rudie. A survey of modeling and control of hybrid systems. *Annual Reviews in Control*, 21:79–92, 1997.

[170] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5:1–11, January 1987.

[171] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. Modeling long–running transactions with communicating hierarchical timed automata. In Roberto Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2006.

[172] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. Design and verification of long-running transactions in a timed framework. *Science of Computer Programming*, 73:76–94, October 2008.

[173] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Adriano Peron. Timed cooperating automata. *Fundamenta Informaticae*, 43:153–173, August 2000.

[174] Ruggero Lanotte, Andrea Maggiolo-Schettini, Simone Tini, and Adriano Peron. Transformations of timed cooperating automata. *Fundamenta Informaticae*, 47:271–282, October 2001.

[175] Francois Laroussinie and Kim G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In Stan Budkowski, Ana Cavalli, and Elie Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, volume 6 of *IFIP—The International Federation for Information Processing*, pages 439–456. Springer US, 1998.

[176] Kim G. Larsen. Priced timed automata: Theory and tools. In Ravi Kannan and K Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software*

Technology and Theoretical Computer Science (FSTTCS 2009), volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 417–425, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[177] Kim G. Larsen, Axel Legay, Louis-Marie Traonouez, and Andrzej Wąsowski. Robust specification of real time components. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '11, pages 129–144, Berlin, Heidelberg, 2011. Springer-Verlag.

[178] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306, New York, NY, USA, 2005. ACM.

[179] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6:271–298, September 1999.

[180] Kim G. Larsen and Jacob I. Rasmussen. Optimal reachability for multi-priced timed automata. *Theoretical Computer Science*, 390:197–213, January 2008.

[181] Kim G. Larsen and Yi Wang. Time-abstracted bisimulation: implicit specifications and decidability. *Information and Computation*, 134:75–101, May 1997.

[182] Slawomir Lasota and Igor Walukiewicz. Alternating timed automata. *ACM Transactions on Computational Logic*, 9:10:1–10:27, April 2008.

[183] Mark Lawford. *Model Reduction of Discrete Real-Time Systems*. PhD thesis, Department of Electrical Computer Engineering, University of Toronto, Toronto, ON, Canada, 1997.

[184] Mark Lawford, William Murray Wonham, and Jonathan S. Ostroff. State-event observers for labeled transition systems. In *Proceedings of the 33rd IEEE Conference on Decision and Control*, volume 4, pages 3642–3648, December 1994.

[185] Shang-Wei Lin, Pao-Ann Hsiung, Chun-Hsian Huang, and Yean-Ru Chen. Model checking prioritized timed automata. In Doron A. Peled and Yih-Kuen Tsay, editors, *Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 370–384. Springer Berlin Heidelberg, 2005.

[186] Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. McAiT: a timing analyzer for multicore real-time software. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 414–417, Berlin, Heidelberg, 2011. Springer-Verlag.

[187] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, Washington, DC, USA, 2010. IEEE Computer Society.

[188] Gabor Madl and Nikil Dutt. Tutorial for the Open-source Dream Tool. In *CECS Technical Report*, 2006.

[189] Oded Maler, Dejan Nickovic, and Amir Pnueli. Pillars of computer science. chapter Checking Temporal Properties of Discrete, Timed and Continuous Behaviors, pages 475–505. Springer-Verlag, Berlin, Heidelberg, 2008.

[190] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *Symposium on Theoretical Aspects of Computer Science*, pages 229–242, 1995.

[191] Lakshmi Manasa, Shankara Narayanan Krishna, and Chinmay Jain. Model checking weighted integer reset timed automata. *Theory of Computing Systems*, 48(3):648–679, April 2011.

[192] Nicolas Markey. *Verification of Embedded Systems – Algorithms and Complexity*. Mémoire d'habilitation, École Normale Supérieure de Cachan, France, 2011.

[193] Jennifer McManis and Pravin Varaiya. Suspension automata: A decidable class of hybrid automata. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 105–117, London, UK, 1994. Springer-Verlag.

[194] Joseph S. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control*, HSCC '00, pages 296–309, London, UK, 2000. Springer-Verlag.

[195] Peter Niebert, Stavros Tripakis, and Sergio Yovine. Minimum-time reachability for timed automata. In *Proceedings of the 8th Mediteranean Conference on Control and Automation*, MED'2000.

[196] Christer Norström, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, RTCSA '99, pages 182–189, Washington, DC, USA, 1999. IEEE Computer Society.

[197] Jonathan S. Ostroff. *Temporal Logic for Real Time Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[198] Joël Ouaknine and James Worrell. On the decidability and complexity of metric temporal logic over finite words. *Logical Methods in Computer Science*, 3(1), 2007.

[199] Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 1–13, Berlin, Heidelberg, 2008. Springer-Verlag.

[200] Martin Ouimet and Kristina Lundqvist. The TASM toolset: Specification, simulation, and formal verification of real-time systems. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 126–130, Berlin, Heidelberg, 2007. Springer-Verlag.

[201] Pawel Parys and Igor Walukiewicz. Weak alternating timed automata. In *Proceedings of the 36th Internatilonal Collogquium on Automata, Languages and Programming: Part II*, ICALP '09, pages 273–284, Berlin, Heidelberg, 2009. Springer-Verlag.

[202] Wojciech Penczek and Bożena Woźna. Towards bounded model checking for Timed Automata. In L. Czaja, editor, *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'01)*, pages 195–209. Warsaw University, 2001.

[203] Ernesto Posse and Juergen Dingel. Theory and implementation of a real-time extension to the π-calculus. In John Hatcliff and Elena Zucca, editors, *Formal Techniques*

for Distributed Systems, volume 6117 of *Lecture Notes in Computer Science*, pages 125–139. Springer Berlin Heidelberg, 2010.

[204] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, August 2000.

[205] Anuj Puri. Dynamical properties of timed automata. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 210–227. Springer Berlin Heidelberg, 1998.

[206] Chander Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[207] Pierre-Alain Reynier. Diagonal constraints handled efficiently in UPPAAL. Technical Report LSV-07-02, Laboratoire Spécification et Vérification, ENS Cachan, France, 2007.

[208] Tomas G. Rokicki. *Representing and modeling digital circuits*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, USA, 1993.

[209] Pierluigi San P. and Zhe Dang. Automatic verification of multi-queue discrete timed automata. In Tandy Warnow and Binhai Zhu, editors, *Computing and Combinatorics*, volume 2697 of *Lecture Notes in Computer Science*, pages 159–171. Springer Berlin Heidelberg, 2003.

[210] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A resource model for embedded systems. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 84–94, Washington, DC, USA, 2009. IEEE Computer Society.

[211] Severine Sentilles, Anders Pettersson, Dag Nystrom, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-IDE - a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 607–610, Washington, DC, USA, 2009. IEEE Computer Society.

[212] Alan C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18:805–816, September 1992.

[213] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Modeling mixed-critical systems in real-time BIP. In *Proceedings of the Workshop on Real-Time Mixed Criticality Systems,*, 2013.

[214] Maria Sorea. Tempo: A model-checker for event-recording automata. In *Proceedings of the 1st Workshop on Real-Time Tools*, Aalborg, Denmark, August 2001.

[215] Maria Sorea. *Verification of Real-Time Systems through Lazy Approximations*. PhD thesis, Universität Ulm, Germany, 2003.

[216] P. Vijay Suman and Paritosh K. Pandya. Determinization and expressiveness of integer reset timed automata with silent transitions. In Adrian Horia Dediu, Armand Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and*

Applications, volume 5457 of *Lecture Notes in Computer Science*, pages 728–739. Springer Berlin Heidelberg, 2009.

[217] P. Vijay Suman, Paritosh K. Pandya, Shankara Narayanan Krishna, and Lakshmi Manasa. Timed automata with integer resets: Language inclusion and expressiveness. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin Heidelberg, 2008.

[218] Serdar Tasiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying abstractions of timed systems. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 546–562. Springer Berlin Heidelberg, 1996.

[219] Omer Nguena Timo and Antoine Rollet. Conformance testing of variable driven automata. In *Proceedings of the 8th IEEE International Workshop on Factory Communication Systems Communication in Automation*, pages 241–248, May 2010.

[220] Claire Tomlin, Ian Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of The IEEE*, 91(7):986–1001, July 2003.

[221] Stavros Tripakis. Fault diagnosis for timed automata. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 205–221. Springer Berlin Heidelberg, 2002.

[222] Stavros Tripakis. Folk theorems on the determinization and minimization of timed automata. *Information Processing Letters*, 99(6):222–226, 2006.

[223] Stavros Tripakis. Checking timed Büchi automata emptiness on simulation graphs. *ACM Transactions on Computational Logic*, 10:15:1–15:19, April 2009.

[224] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '96, pages 329–348, London, UK, 1996. Springer-Verlag.

[225] Stavros Tripakis and Thao Dang. *Model-based Design of Heterogeneous Systems*, chapter Modeling, Verification and Testing using Timed and Hybrid Automata. CRC Press, 2009.

[226] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18:25–68, January 2001.

[227] Stavros Tripakis, Sergio Yovine, and Ahmed Bouajjani. Checking timed Büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, May 2005.

[228] Ashutosh Trivedi and Dominik Wojtczak. Recursive timed automata. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Automated Technology for Verification and Analysis*, volume 6252 of *Lecture Notes in Computer Science*, pages 306–324. Springer Berlin Heidelberg, 2010.

[229] Nguyen Van Tang and Mizuhito Ogawa. Event-clock visibly pushdown automata. In Mogens Nielsen, Antonín Kučera, PeterBro Miltersen, Catuscia Palamidessi, Petr Tůma, and Frank Valencia, editors, *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 558–569. Springer Berlin Heidelberg, 2009.

[230] Md T. B. Waez, Juergen Dingel, and Karen Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9(0):1–26, 2013.

[231] Md T. B. Waez, Andrzej Wąsowski, Juergen Dingel, and Karen Rudie. A model for industrial real-time systems. Technical Report 2014-622, Queen's University, ON, 2014. `http://research.cs.queensu.ca/TechReports/Reports/2014-622.pdf`.

[232] Md T. B. Waez, Andrzej Wąsowski, Juergen Dingel, and Karen Rudie. Synthesis of a reconfiguration service for mixed-criticality multi-core systems. Technical Report 2014-619, Queen's University, ON, 2014. `http://research.cs.queensu.ca/TechReports/Reports/2014-619.pdf`.

[233] Md T. B. Waez, Andrzej Wąsowski, Juergen Dingel, and Karen Rudie. Synthesis of a reconfiguration service for mixed-criticality multi-core systems: An experience report. In Ivan Lanese and Eric Madelaine, editors, *Formal Aspects of Component Software*, Lecture Notes in Computer Science, pages 162–180. Springer International Publishing, 2015.

[234] Farn Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In *Proceedings of the IFIP TC6/WG6.1 - 21st International*

Conference on Formal Techniques for Networked and Distributed Systems, FORTE '01, pages 235–250, Deventer, Netherlands, 2001. Kluwer, B.V.

[235] Farn Wang. Efficient verification of timed automata with BDD-like data structures. *International Journal on Software Tools for Technology Transfer*, 6:77–97, July 2004.

[236] Farn Wang. Formal verification of timed systems: A survey and perspective. volume 92, pages 1283–1305. IEEE, August 2004.

[237] Farn Wang. REDLIB for the formal verification of embedded systems. In *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ISOLA '06, pages 341–346, Washington, DC, USA, 2006. IEEE Computer Society.

[238] Libor Waszniowski, Jan Krákora, and Zdenk Hanzálek. Case study on distributed and fault tolerant system modeling based on timed automata. *Journal of Systems and Software*, 82(10):1678–1694, October 2009.

[239] Thomas Wilke. *Automaten und Logiken für zeitabhängige Systeme*. Dissertation, Christian-Albrechts-Universität zu Kiel, 1994.

[240] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.

[241] Youmin Zhang and Jin Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual Reviews in Control*, 32(2):229–252, 2008.

Appendix A

Appendix for Chapter 1

Name	Type	Purpose
P	Constant	To store total processing units
D1	Constant	To store the deadline for tasks T1, T4, T7, T10
D2	Constant	To store the deadline for tasks T2, T5, T8, T11
D3	Constant	To store the deadline for tasks T3, T6, T9, T12
W1	Constant	To store the WCET of task T1
W2	Constant	To store the WCET of task T2
W3	Constant	To store the WCET of task T3
W4	Constant	To store the WCET of task T4
W5	Constant	To store the WCET of task T5
W6	Constant	To store the WCET of task T6
W7	Constant	To store the WCET of task T7
W8	Constant	To store the WCET of task T8
W9	Constant	To store the WCET of task T9
W10	Constant	To store the WCET of task T10
W11	Constant	To store the WCET of task T11
W12	Constant	To store the WCET of task T12

Table A.1: Constants in the models

Name	Type	Purpose
x1	Clock	To record time passed since x1 equals to dealine D1
x2	Clock	To record time passed since x2 equals to dealine D2
x3	Clock	To record time passed since x3 equals to dealine D3
y	Clock	To record time passed since the last discrete time unit
B	Integer	Buffer B=1 when the executing task on the failed core need to execute 1 time unit more, B=2 when the extra time unit is currently, executing, & B=0 is otherwise
F	Bool	Fault F=0 if no fault has occurred yet, otherwise F=1
h	Integer	Halt h=0 if no runaway task present, otherwise h=1
p	Integer	Number of occupied (or broken) processing units
e1	Integer	Remaining execution time of task T1
e2	Integer	Remaining execution time of task T2
e3	Integer	Remaining execution time of task T3
e4	Integer	Remaining execution time of task T4
e5	Integer	Remaining execution time of task T5
e6	Integer	Remaining execution time of task T6
e7	Integer	Remaining execution time of task T7
e8	Integer	Remaining execution time of task T8
e9	Integer	Remaining execution time of task T9
e10	Integer	Remaining execution time of task T10
e11	Integer	Remaining execution time of task T11
e12	Integer	Remaining execution time of task T12
s1	Integer	State of T1: 0 = suspend or runaway, 1 = ready, 2 = running
s2	Integer	State of T2: 0 = suspend or runaway, 1 = ready, 2 = running
s3	Integer	State of T3: 0 = suspend or runaway, 1 = ready, 2 = running
s4	Integer	State of T4: 0 = suspend or runaway, 1 = ready, 2 = running
s5	Integer	State of T5: 0 = suspend or runaway, 1 = ready, 2 = running
s6	Integer	State of T6: 0 = suspend or runaway, 1 = ready, 2 = running
s7	Integer	State of T7: 0 = suspend or runaway, 1 = ready, 2 = running
s8	Integer	State of T8: 0 = suspend or runaway, 1 = ready, 2 = running
s9	Integer	State of T9: 0 = suspend or runaway, 1 = ready, 2 = running
s10	Integer	State of T10: 0 = suspend or runaway, 1 = ready, 2 = running
s11	Integer	State of T11: 0 = suspend or runaway, 1 = ready, 2 = running
s12	Integer	State of T12: 0 = suspend or runaway, 1 = ready, 2 = running

Table A.2: Variables in the models

```
void set1(){
if (h!=1)        {s1=1;   e1=W1;}
if (h!=4)        {s4=1;   e4=W4;}
if (h!=7)        {s7=1;   e7=W7;}
if (h!=10)       {s10=1;  e10=W10;}}

void set2(){
if (h!=2)        {s2=1;   e2=W2;}
if (h!=5)        {s5=1;   e5=W5;}
if (h!=8)        {s8=1;   e8=W8;}
if (h!=11)       {s11=1;  e11=W11;}}

void set3(){
if (h!=3)        {s3=1;   e3=W3;}
if (h!=6)        {s6=1;   e6=W6;}
if (h!=9)        {s9=1;   e9=W9;}
if (h!=12)       {s12=1;  e12=W12;}}
```

Figure A.1: Functions `set1`, `set2`, and `set3`

```
void update(){
if (B==2)                {B=0;   p--;}
if (h!=1 && s1==2)       {e1--;  p--;    if (e1==0)s1=0; else s1=1;}
else if (h==1 && s1==2) {        p--;    s1=1;}
if (h!=2 && s2==2)       {e2--;  p--;    if (e2==0)s2=0; else s2=1;}
else if (h==2 && s2==2) {        p--;    s2=1;}
if (h!=3 && s3==2)       {e3--;  p--;    if (e3==0)s3=0; else s3=1;}
else if (h==3 && s3==2) {        p--;    s3=1;}
if (h!=4 && s4==2)       {e4--;  p--;    if (e4==0)s4=0; else s4=1;}
else if (h==4 && s4==2) {        p--;    s4=1;}
if (h!=5 && s5==2)       {e5--;  p--;    if (e5==0)s5=0; else s5=1;}
else if (h==5 && s5==2) {        p--;    s5=1;}
if (h!=6 && s6==2)       {e6--;  p--;    if (e6==0)s6=0; else s6=1;}
else if (h==6 && s6==2) {        p--;    s6=1;}
if (h!=7 && s7==2)       {e7--;  p--;    if (e7==0)s7=0; else s7=1;}
else if (h==7 && s7==2) {        p--;    s7=1;}
if (h!=8 && s8==2)       {e8--;  p--;    if (e8==0)s8=0; else s8=1;}
else if (h==8 && s8==2) {        p--;    s8=1;}
if (h!=9 && s9==2)       {e9--;  p--;    if (e9==0)s9=0; else s9=1;}
else if (h==9 && s9==2) {        p--;    s9=1;}
if (h!=10 && s10==2)     {e10--; p--;    if (e10==0)s10=0; else s10=1;}
else if (h==10 && s10==2){       p--;    s10=1;}
if (h!=11 && s11==2)     {e11--; p--;    if (e11==0)s11=0; else s11=1;}
else if (h==11 && s11==2){       p--;    s11=1;}
if (h!=12 && s12==2)     {e12--; p--;    if (e12==0)s12=0; else s12=1;}
else if (h==12 && s12==2){       p--;    s12=1;}}
```

Figure A.2: Function `update`

```
void assignment(){
if (B==1 && p<P)          {B=2;   p++;}
if (s1==1 && p<P)         {s1=2;  p++;}
if (s4==1 && p<P)         {s4=2;  p++;}
if (s7==1 && p<P)         {s7=2;  p++;}
if (s10==1 && p<P)        {s10=2; p++;}
if (s2==1 && p<P)         {s2=2;  p++;}
if (s5==1 && p<P)         {s5=2;  p++;}
if (s8==1 && p<P)         {s8=2;  p++;}
if (s11==1 && p<P)        {s11=2; p++;}
if (s3==1 && p<P)         {s3=2;  p++;}
if (s6==1 && p<P)         {s6=2;  p++;}
if (s9==1 && p<P)         {s9=2;  p++;}
if (s12==1 && p<P)        {s12=2; p++;}}
```

Figure A.3: Function `assignment`

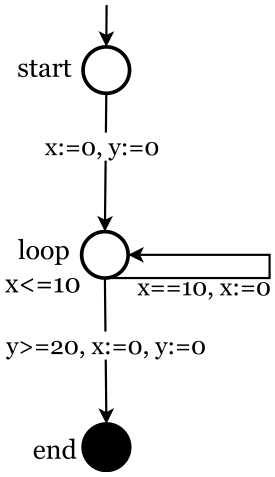
Appendix B

Appendix for Chapter 2

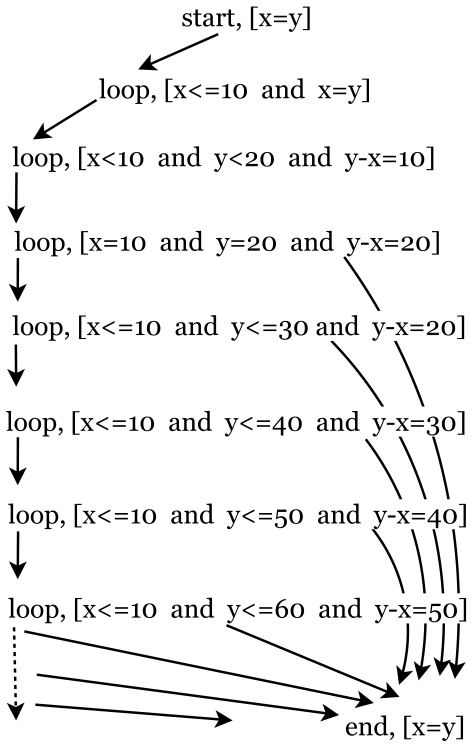
B.1 Finiteness of Zone Graph

Zone graphs are not always finite [46, 99], which makes exhaustive exploration impossible. To remedy this problem, one approach is to construct a *region-closed zone graph* [57, 227, 223]: replace each $[\delta] \in \mathcal{Z}(A)$ by the union of the regions of $\mathcal{R}(A)$ which intersect $[\delta]$. Since the number of regions is finite, there is a finite number of zones after this operation. The region closure of a zone may not be convex. As a result, DBM cannot be used. For this reason, the region-closed zone graph is not used in practice.

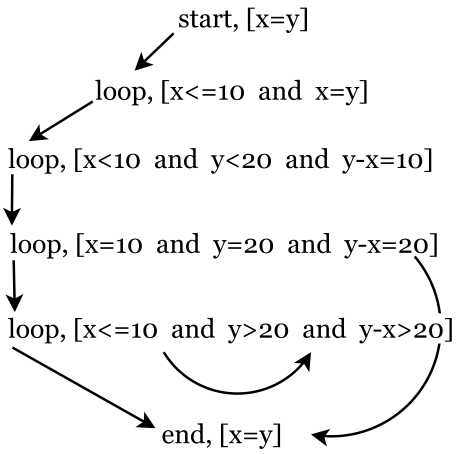
Another approach to guarantee finiteness of zone graph is the use of an abstraction operator called the *k-extrapolation*, where k is a constant supposed to be greater than the maximal constant occurring in the automaton A [46, 99, 208]. The k-extrapolation operator abstracts $\mathcal{Z}(A)$ into another zone graph $\mathcal{Z}'(A)$, denoted *k-extrapolated zone graph*, such that all constraints defined in $\mathcal{Z}'(A)$ are k-bounded. The k-extrapolated zone graph is finite, since the number of clock zones with bounded constraints is finite. As an example, a finite k-extrapolated zone graph of the infinite zone graph of Figure B.1(b) is shown in Figure B.1(c).



(a) A timed automaton



(b) An infinite zone graph



(c) A k-extrapolated zone graph

Figure B.1: A timed automaton with its infinite zone graph and its k-extrapolated (here, $k = 20$) zone graph [46]

A k-extrapolated zone graph is *correct for reachability*[1] only for diagonal-free timed automata [45, 59]. If automaton A has any diagonal constraint, zone graph $\mathcal{Z}(A)$ may have a reachable zone $\langle l, [\delta] \rangle$ but l is not a reachable location in A. A k-extrapolated zone graph is correct for reachability if clock valuation ν satisfies a diagonal constraint δ *if and only if* clock valuation μ satisfies δ, where ν and μ are k-extrapolated zone equivalent clock valuations [46]. One method to ensure correctness for reachability of a k-extrapolated zone graph is to check this property [46]. However, checking this property may suffer from an exponential blow-up in the number of zones. The number of zones is multiplied by 2^n, where n is the number of diagonal constraints. To remedy this problem, a new method has been proposed based on counter-example[2] guided abstraction refinement [66] and has been applied [207] in Uppaal [36]. Not all the diagonal constraints cause incorrectness for reachability. In practice, diagonal constraints produce an incorrect result only rarely. Since counter-examples are rare, this refinement method causes very little overhead in practice.

[1] We say that a symbolic transition system T' of an original transition system T is correct for reachability *if and only if* a state s is reachable in T then there is a reachable symbolic state s' in T' which contains s.

[2] A counter-example is a trace where location l in A is not reachable, but zone $\langle l, [\delta] \rangle$ in $\mathcal{Z}(A)$ is reachable.

Property	Closure Under	Property	Closure Under
Union	Yes	Kleene-star	Yes
Intersection	Yes	Projection	Yes
Concatenation	Yes	Shuffle	No
Renaming	Yes	Complementation	No

Table B.1: Closure properties of timed automata

Problem	Complexity
Emptiness checking	PSPACE-complete
Timed bisimulation	EXPTIME
Timed simulation	EXPTIME
Universality	Undecidable
Language inclusion	Undecidable
Determinizability	Undecidable
Minimum-time reachability	PSPACE-hard
Computing the clock degree	Undecidable
Language equivalence	Undecidable
Reducing the size of constants	Undecidable
Minimization of the number of clocks	Undecidable
Binary reachability	Decidable

Table B.2: Decision problems of timed automata

Property	Closed Under	Property	Closed Under
Union	Yes	Complementation	Yes
Intersection	Yes	Projection	No
Renaming	No		

Table B.3: Closure properties of deterministic timed automata

Problem	Complexity	Problem	Complexity
Emptiness checking	PSPACE-complete	Language inclusion	PSPACE-complete
Universality	PSPACE-complete	Language equivalence	PSPACE-complete

Table B.4: Complexity of decision problems for deterministic timed automata

Clock Constraint	Reachability
$x \sim q$	PSPACE-complete
$x - y \sim q$	PSPACE-complete
$x \sim e$	Undecidable
$x - y \sim e$	Undecidable
$d + n \cdot \theta \leq x \leq e + n \cdot \theta$	PSPACE-complete
$d + n \cdot \theta \leq x - y \leq e + n \cdot \theta$	PSPACE-complete
$x \sim q \cdot y$	Undecidable
$x - y \sim q \cdot z$	Undecidable

Table B.5: Complexity of reachability checking using different clock constraints

Clock Update	Diagonal-Free	With Diagonal
$x := c$	PSPACE-complete	PSPACE-complete
$x := y$	PSPACE-complete	PSPACE-complete
$x := x + 1$	PSPACE-complete	Undecidable
$x := y + c$	PSPACE-complete	Undecidable
$x := x - 1$	Undecidable	Undecidable
$x :< c$	PSPACE-complete	PSPACE-complete
$x :> c$	PSPACE-complete	Undecidable
$x :\sim y + c$	PSPACE-complete	Undecidable
$y + c <: x :< y + d$	PSPACE-complete	Undecidable
$y + c <: x :< z + d$	Undecidable	Undecidable

Table B.6: Complexity of reachability checking using different clock updates

Task Automata	Preemptive	Non-Preemptive
Fixed and Feedback	PSPACE-complete	PSPACE-complete
Fixed and Non-Feedback	PSPACE-complete	PSPACE-complete
Flexible and Feedback	Undecidable	PSPACE-complete
Flexible and Non-Feedback	PSPACE-complete	PSPACE-complete

Table B.7: Complexity of preemptive and non-preemptive scheduling of task automata

Appendix C

Appendix for Chapter 3

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS	Integer	To record the core which is currently assigned to execute S
aW	Integer	To record the core which is currently assigned to execute W
aD	Integer	To record the core which is currently assigned to execute D
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
uS	Boolean	To record whether an update in S is performed (1) or not (0)
uW	Boolean	To record whether an update in W is performed (1) or not (0)
uD	Boolean	To record whether an update in D is performed (1) or not (0)
L1	Integer	To record the current worst possible loads on core_1
L2	Integer	To record the current worst possible loads on core_2
L3	Integer	To record the current worst possible loads on core_3
vS	Integer	To record the current value in the speedometer
sD	Integer	To record the current door state
F	Integer	To record the current total number of core failures

Table C.1: Variables in the concrete model

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on core_1
lS2	Constant	To store the worst-case load of S on core_2
lS3	Constant	To store the worst-case load of S on core_3
lW1	Constant	To store the worst-case load of W on core_1
lW2	Constant	To store the worst-case load of W on core_2
lW3	Constant	To store the worst-case load of W on core_3
lD1	Constant	To store the worst-case load of D on core_1
lD2	Constant	To store the worst-case load of D on core_2
lD3	Constant	To store the worst-case load of D on core_3
Limit	Constant	To store the load limit of every core

Table C.2: Constants in the concrete model

Name	From	To	Purpose
iS1	$core_1$	$core_1$.S	To initialize S on $core_1$
iS2	$core_2$	$core_2$.S	To initialize S on $core_2$
iS3	$core_3$	$core_3$.S	To initialize S on $core_3$
iW1	$core_1$	$core_1$.W	To initialize W on $core_1$
iW2	$core_2$	$core_2$.W	To initialize W on $core_2$
iW3	$core_3$	$core_3$.W	To initialize W on $core_3$
iD1	$core_1$	$core_1$.D	To initialize D on $core_1$
iD2	$core_2$	$core_2$.D	To initialize D on $core_2$
iD3	$core_3$	$core_3$.D	To initialize D on $core_3$
rS1	service	$core_1$.S	To resume S on $core_1$
rS2	service	$core_2$.S	To resume S on $core_2$
rS3	service	$core_3$.S	To resume S on $core_3$
rW1	service	$core_1$.W	To resume W on $core_1$
rW2	service	$core_2$.W	To resume W on $core_2$
rW3	service	$core_3$.W	To resume W on $core_3$
rD1	service	$core_1$.D	To resume D on $core_1$
rD2	service	$core_2$.D	To resume D on $core_2$
rD3	service	$core_3$.D	To resume D on $core_3$
kS1	$core_1$, service	$core_1$.S	To kill S on $core_1$
kS2	$core_2$, service	$core_2$.S	To kill S on $core_2$
kS3	$core_3$, service	$core_3$.S	To kill S on $core_3$
kW1	$core_1$, service	$core_1$.W	To kill W on $core_1$
kW2	$core_2$, service	$core_2$.W	To kill W on $core_2$
kW3	$core_3$, service	$core_3$.W	To kill W on $core_3$
kD1	$core_1$, service	$core_1$.D	To kill D on $core_1$
kD2	$core_2$, service	$core_2$.D	To kill D on $core_2$
kD3	$core_3$, service	$core_3$.D	To kill D on $core_3$

Table C.3: Actions in the concrete model (part 1)

Name	From	To	Purpose
tS1	$core_1$.S	$core_1$	To terminate S on $core_1$
tS2	$core_2$.S	$core_2$	To terminate S on $core_2$
tS3	$core_3$.S	$core_3$	To terminate S on $core_3$
tW1	$core_1$.W	$core_1$	To terminate W on $core_1$
tW2	$core_2$.W	$core_2$	To terminate W on $core_2$
tW3	$core_3$.W	$core_3$	To terminate W on $core_3$
tD1	$core_1$.D	$core_1$	To terminate D on $core_1$
tD2	$core_2$.D	$core_2$	To terminate D on $core_2$
tD3	$core_3$.D	$core_3$	To terminate D on $core_3$
mSW	$core_1$, $core_2$	service	To inform that it is assigned to execute S and W
mSD	$core_1$, $core_3$	service	To inform that it is assigned to execute S and D
mWD	$core_2$, $core_3$	service	To inform that it is assigned to execute W and D
mS	$core_1$	service	To inform that it is assigned to execute S
mW	$core_2$	service	To inform that it is assigned to execute W
mD	$core_3$	service	To inform that it is assigned to execute D

Table C.4: Actions in the concrete model (part 2)

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether core_1 is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether core_1 is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether core_1 is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether core_2 is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether core_2 is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether core_2 is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether core_3 is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether core_3 is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether core_3 is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
L1	Integer	To record the current worst possible loads on core_1
L2	Integer	To record the current worst possible loads on core_2
L3	Integer	To record the current worst possible loads on core_3
F	Integer	To record the current total number of core failures

Table C.5: Variables in the abstract model

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on core_1
lS2	Constant	To store the worst-case load of S on core_2
lS3	Constant	To store the worst-case load of S on core_3
lW1	Constant	To store the worst-case load of W on core_1
lW2	Constant	To store the worst-case load of W on core_2
lW3	Constant	To store the worst-case load of W on core_3
lD1	Constant	To store the worst-case load of D on core_1
lD2	Constant	To store the worst-case load of D on core_2
lD3	Constant	To store the worst-case load of D on core_3
Limit	Constant	To store the load limit of every core

Table C.6: Constants in the abstract model

```
void initialize(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         {L1=L1+lS1;     iS=1;}
        else if (task==S && core==2)    {L2=L2+lS2;     iS=1;}
        else if (task==S && core==3)    {L3=L3+lS3;     iS=1;}
        else if (task==W && core==1)    {L1=L1+lW1;     iW=1;}
        else if (task==W && core==2)    {L2=L2+lW2;     iW=1;}
        else if (task==W && core==3)    {L3=L3+lW3;     iW=1;}
        else if (task==D && core==1)    {L1=L1+lD1;     iD=1;}
        else if (task==D && core==2)    {L2=L2+lD2;     iD=1;}
        else if (task==D && core==3)    {L3=L3+lD3;     iD=1;}
}
void terminate(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         {L1=L1-lS1;     iS=0;}
        else if (task==S && core==2)    {L2=L2-lS2;     iS=0;}
        else if (task==S && core==3)    {L3=L3-lS3;     iS=0;}
        else if (task==W && core==1)    {L1=L1-lW1;     iW=0;}
        else if (task==W && core==2)    {L2=L2-lW2;     iW=0;}
        else if (task==W && core==3)    {L3=L3-lW3;     iW=0;}
        else if (task==D && core==1)    {L1=L1-lD1;     iD=0;}
        else if (task==D && core==2)    {L2=L2-lD2;     iD=0;}
        else if (task==D && core==3)    {L3=L3-lD3;     iD=0;}
}
```

Figure C.1: Functions `initialize` and `terminate` in the concrete model

```
void kill(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         L1=L1-lS1;
        else if (task==S && core==2)    L2=L2-lS2;
        else if (task==S && core==3)    L3=L3-lS3;
        else if (task==W && core==1)    L1=L1-lW1;
        else if (task==W && core==2)    L2=L2-lW2;
        else if (task==W && core==3)    L3=L3-lW3;
        else if (task==D && core==1)    L1=L1-lD1;
        else if (task==D && core==2)    L2=L2-lD2;
        else if (task==D && core==3)    L3=L3-lD3;
}
void cancel(int[0,3] task1, int[0,3]  task2)
{
        if (task1==S && task2==0)       aS=0;
        else if (task1==W && task2==0)  aW=0;
        else if (task1==D && task2==0)  aD=0;
        else if (task1==0 && task2==S)  aS=0;
        else if (task1==0 && task2==W)  aW=0;
        else if (task1==0 && task2==D)  aD=0;
        else if (task1==S && task2==W)  {aS=0;  aW=0;}
        else if (task1==W && task2==S)  {aS=0;  aW=0;}
        else if (task1==S && task2==D)  {aS=0;  aD=0;}
        else if (task1==D && task2==S)  {aS=0;  aD=0;}
        else if (task1==W && task2==D)  {aW=0;  aD=0;}
        else if (task1==D && task2==W)  {aW=0;  aD=0;}
}
```

Figure C.2: Functions `kill`, and `cancel` in the concrete model

```
void resume(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         {aS=1;  L1=L1+lS1;}
        else if (task==S && core==2)    {aS=2;  L2=L2+lS2;}
        else if (task==S && core==3)    {aS=3;  L3=L3+lS3;}
        else if (task==W && core==1)    {aW=1;  L1=L1+lW1;}
        else if (task==W && core==2)    {aW=2;  L2=L2+lW2;}
        else if (task==W && core==3)    {aW=3;  L3=L3+lW3;}
        else if (task==D && core==1)    {aD=1;  L1=L1+lD1;}
        else if (task==D && core==2)    {aD=2;  L2=L2+lD2;}
        else if (task==D && core==3)    {aD=3;  L3=L3+lD3;}
}
void reassign(int[1,3] task, int[1,3] core)
{
        if (task==S && core==1)         aS=1;
        else if (task==S && core==2)    aS=2;
        else if (task==S && core==3)    aS=3;
        else if (task==W && core==1)    aW=1;
        else if (task==W && core==2)    aW=2;
        else if (task==W && core==3)    aW=3;
        else if (task==D && core==1)    aD=1;
        else if (task==D && core==2)    aD=2;
        else if (task==D && core==3)    aD=3;
}
```

Figure C.3: Functions `resume` and `reassign` in the concrete model

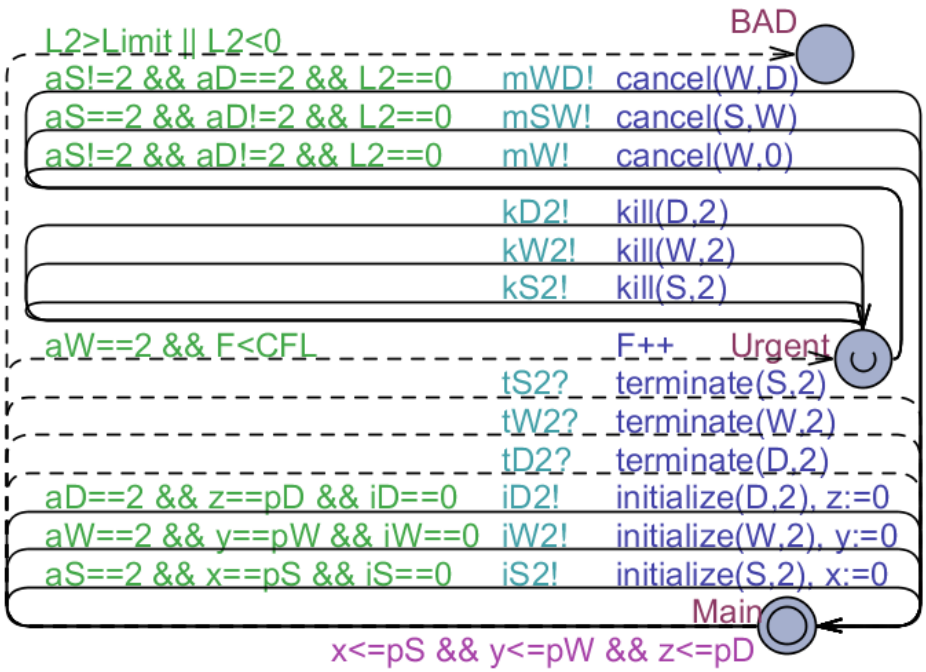


Figure C.4: Automaton core_2 in the concrete model

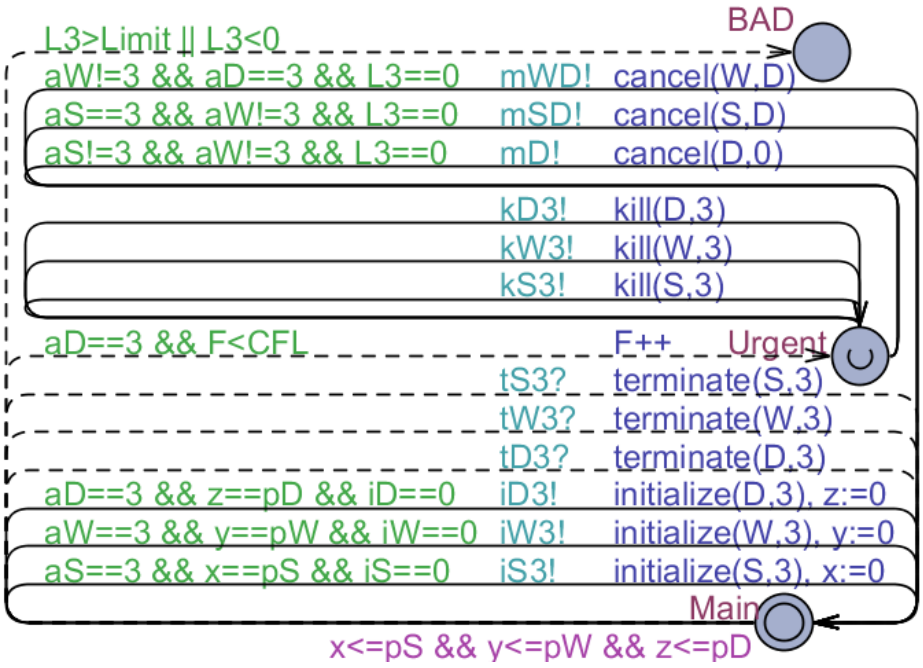


Figure C.5: Automaton core_3 in the concrete model

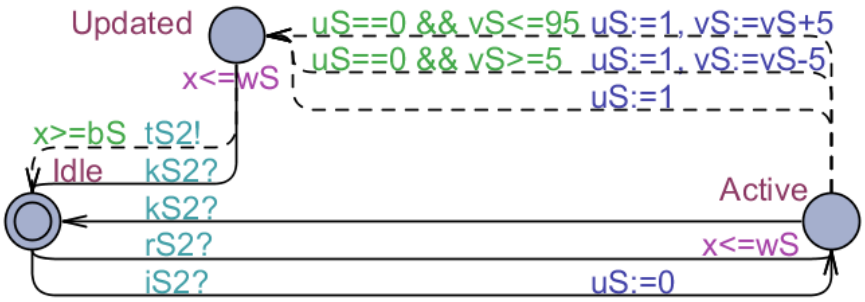


Figure C.6: Automaton core_2.S in the concrete model

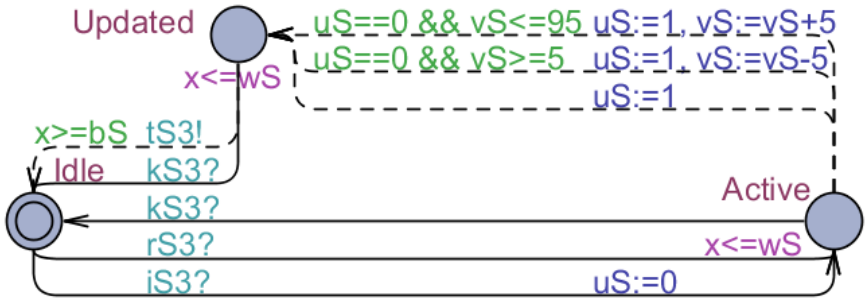


Figure C.7: Automaton core_3.S in the concrete model

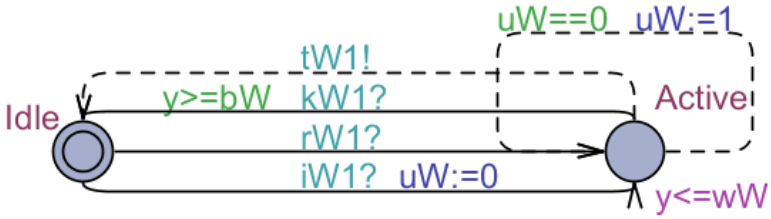


Figure C.8: Automaton core_1.W in the concrete model

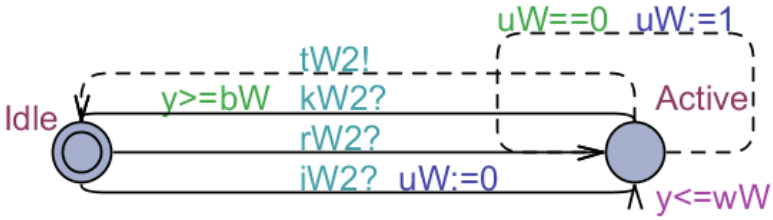


Figure C.9: Automaton core_2.W in the concrete model

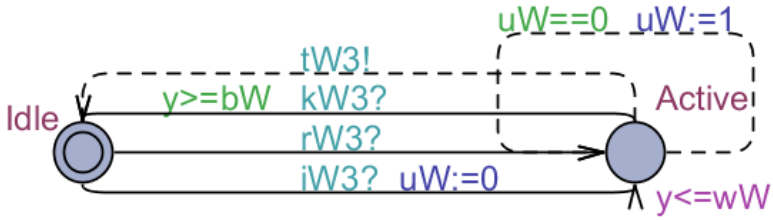


Figure C.10: Automaton core_3.W in the concrete model

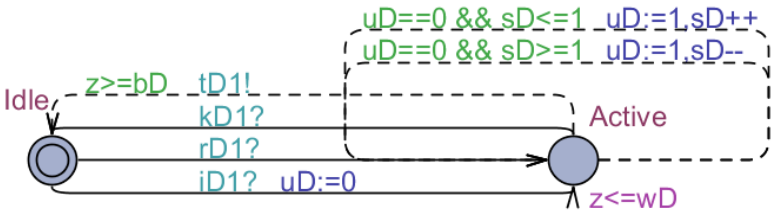


Figure C.11: Automaton $\mathsf{core}_1.\mathtt{D}$ in the concrete model

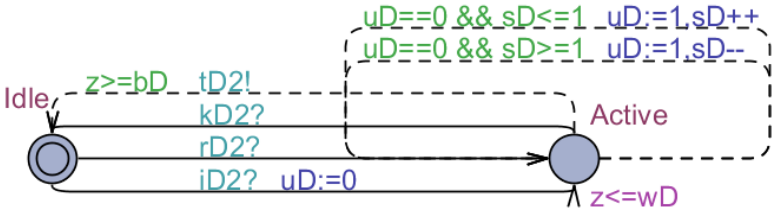


Figure C.12: Automaton $\mathsf{core}_2.\mathtt{D}$ in the concrete model

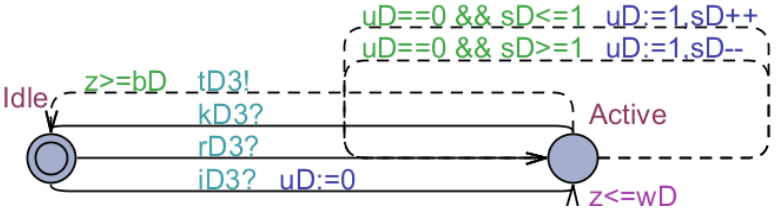


Figure C.13: Automaton $\mathsf{core}_3.\mathtt{D}$ in the concrete model

```
void initializeA(int[1,3] task)
  {
      if (task==S)      {L1=L1+lS1*aS1; L2=L2+lS2*aS2; L3=L3+lS3*aS3;  iS=1;}
      else if (task==W) {L1=L1+lW1*aW1; L2=L2+lW2*aW2; L3=L3+lW3*aW3; iW=1;}
      else if (task==D) {L1=L1+lD1*aD1; L2=L2+lD2*aD2; L3=L3+lD3*aD3; iD=1;}
  }
void terminateA(int[1,3] task)
  {
      if (task==S)      {L1=L1-lS1*aS1; L2=L2-lS2*aS2; L3=L3-lS3*aS3; iS=0;}
      else if (task==W) {L1=L1-lW1*aW1; L2=L2-lW2*aW2; L3=L3-lW3*aW3; iW=0;}
      else if (task==D) {L1=L1-lD1*aD1; L2=L2-lD2*aD2; L3=L3-lD3*aD3; iD=0;}
  }
```

Figure C.14: Functions `initializeA` and `terminateA` in the abstract model

```
void reallocate(int[1,3] task, int[1,3] core){
        if (task==S && core==1)          {aS1=1;          L1=L1+iS*lS1;
                if (aS2==1)              {aS2=0;          L2=L2-iS*lS2;}
                else if (aS3==1)         {aS3=0;          L3=L3-iS*lS3;}}
        else if (task==S && core==2)     {aS2=1;          L2=L2+iS*lS2;
                 if (aS1==1)             {aS1=0;          L1=L1-iS*lS1;}
                else if (aS3==1)         {aS3=0;          L3=L3-iS*lS3;}}
        else if (task==S && core==3)     {aS3=1;          L3=L3+iS*lS3;
                if (aS1==1)              {aS1=0;          L1=L1-iS*lS1;}
                else if (aS2==1)         {aS2=0;          L2=L2-iS*lS2;}}
        else if (task==W && core==1)     {aW1=1;          L1=L1+iW*lW1;
                 if (aW2==1)             {aW2=0;          L2=L2-iW*lW2;}
                else if (aW3==1)         {aW3=0;          L3=L3-iW*lW3;}}
        else if (task==W && core==2)     {aW2=1;          L2=L2+iW*lW2;
                 if (aW1==1)             {aW1=0;          L1=L1-iW*lW1;}
                else if (aW3==1)         {aW3=0;          L3=L3-iW*lW3;}}
        else if (task==W && core==3)     {aW3=1;          L3=L3+iW*lW3;
                 if (aW1==1)             {aW1=0;          L1=L1-iW*lW1;}
                else if (aW2==1)         {aW2=0;          L2=L2-iW*lW2;}}
        else if (task==D && core==1)     {aD1=1;          L1=L1+iD*lD1;
                 if (aD2==1)             {aD2=0;          L2=L2-iD*lD2;}
                else if (aD3==1)         {aD3=0;          L3=L3-iD*lD3;}}
        else if (task==D && core==2)     {aD2=1;          L2=L2+iD*lD2;
                 if (aD1==1)             {aD1=0;          L1=L1-iD*lD1;}
                else if (aD3==1)         {aD3=0;          L3=L3-iD*lD3;}}
        else if (task==D && core==3)     {aD3=1;          L3=L3+iD*lD3;
                if (aD1==1)              {aD1=0;          L1=L1-iD*lD1;}
                else if (aD2==1)         {aD2=0;          L2=L2-iD*lD2;}}}
```

Figure C.15: Function `reallocate` in the abstract model

Appendix D

Appendix for Chapter 4

Sample I/O

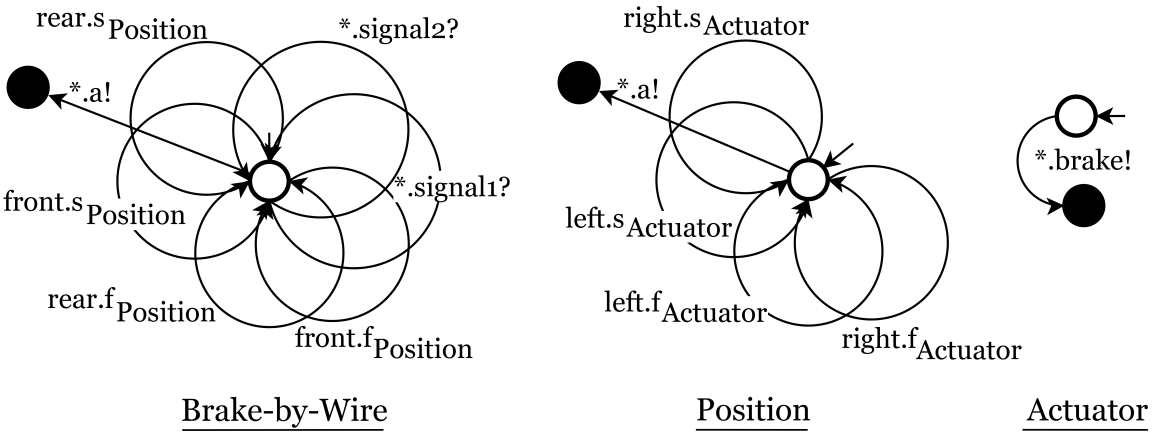


Figure D.1: The Brake-by-Wire system

Input: timed process automaton Actuator creation

1. Name: Actuator

2. Input Actions: ∅

3. Output Actions: {Brake}
4. Callees: ∅
5. Clocks: ∅
6. Channels: ∅
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:*true*, $l1$:*true*
11. Edges:
    - New edge:
      - Source Location: $l0$
      - Action: brake
      - Channel: $*$
      - Clock Constraint: ∅
      - Clock Resets: ∅
      - Destination Location: $l1$

Input: timed process automaton Position creation

1. Name: Position
2. Input Actions: ∅

3. Output Actions: {a}
4. Callees: Actuator
5. Clocks: ∅
6. Channels: {right, left}
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:*true*, $l1$:*true*
11. Edges:
    - New edge:
      - Source Location: $l0$
      - Action: a
      - Channel: $*$
      - Clock Constraint: ∅
      - Clock Resets: ∅
      - Destination Location: $l1$
    - New edge:
      - Source Location: $l0$
      - Action: $s_{Actuator}$
      - Channel: right

  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $s_{Actuator}$
  - Channel: left
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $f_{Actuator}$
  - Channel: right
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $f_{Actuator}$
  - Channel: left
  - Clock Constraint: $\emptyset$

- Clock Resets: $\emptyset$
- Destination Location: $l0$

Input: timed process automaton Brake-by-Wire creation

1. Name: Brake-by-Wire
2. Input Actions: {$signal1$, $signal2$}
3. Output Actions: {a}
4. Callees: Position
5. Clocks: $\emptyset$
6. Channels: {front, rear}
7. Locations: $l0$, $l1$
8. Initial Location: $l0$
9. Final Location: $l1$
10. Invariant: $l0$:*true*, $l1$:*true*
11. Edges:
    - New edge:
      - Source Location: $l0$
      - Action: a
      - Channel: $*$

  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l1$
- New edge:
  - Source Location: $l0$
  - Action: $s_{Position}$
  - Channel: front
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $s_{Position}$
  - Channel: rear
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $f_{Position}$
  - Channel: front
  - Clock Constraint: $\emptyset$

  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: $f_{Position}$
  - Channel: rear
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: signal1
  - Channel: $*$
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$
- New edge:
  - Source Location: $l0$
  - Action: signal2
  - Channel: $*$
  - Clock Constraint: $\emptyset$
  - Clock Resets: $\emptyset$
  - Destination Location: $l0$

Output: timed process automaton Actuator display

- $Name$ = Actuator
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$
- $Edges = \{(l0, brake!, *, \emptyset, \emptyset, l1)\}$

Output: timed process automaton Position display

- $Name$ = Position
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$
- $Edges = \{(l0, a!, *, \emptyset, \emptyset, l1), (l0, \mathsf{s}_{\mathsf{Actuator}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{s}_{\mathsf{Actuator}}, \mathsf{left}, \emptyset, \emptyset, l0),$ $(l0, \mathsf{f}_{\mathsf{Actuator}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{f}_{\mathsf{Actuator}}, \mathsf{left}, \emptyset, \emptyset, l0)\}$

Output: timed process automaton Brake-by-Wire display

- $Name$ = Brake-by-Wire
- $Locations = \{l0 : true, l1 : true\}$
- $initialLocation = l0$
- $finalLocation = l1$

- $Edges = \{(l0, a!, *, \emptyset, \emptyset, l1), (l0, \mathsf{s}_{\mathsf{Position}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{s}_{\mathsf{Position}}, \mathsf{left}, \emptyset, \emptyset, l0), (l0, \mathsf{f}_{\mathsf{Position}}, \mathsf{right}, \emptyset, \emptyset, l0), (l0, \mathsf{f}_{\mathsf{Position}}, \mathsf{left}, \emptyset, \emptyset, l0), (l0, \mathsf{signal1}?, *, \emptyset, \emptyset, l0), (l0, \mathsf{signal2}?, *, \emptyset, \emptyset, l0)\}$

Output: a monolithic analysis model of timed process automaton Actuator

- Timed game analysis of $\mathsf{root}(P_0)$
  - $\mathsf{root}(P_0)$ is
    - $Identifier = P_0$
    - $\mathsf{tpa}(P_0)$=Actuator
    - $\mathsf{channel}(P_0)$=$\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.brake!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l1, \perp.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

Output: a monolithic analysis model of timed process automaton Position

- Timed game analysis of $\mathsf{root}(P_0)||\mathsf{standalone}(P_1)||\mathsf{standalone}(P_2)$
  - $\mathsf{root}(P_0)$ is
    - $Identifier = P_0$
    - $\mathsf{tpa}(P_0)$=Position
    - $\mathsf{channel}(P_0)$=$\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$

* $initialLocation = l0$
* $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\text{right.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_0.\text{left.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\text{right.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_0.\text{left.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\text{f}_{\text{Position}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0 * .u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_1) is

* $Identifier = P_1$
* tpa(P_1)=Actuator
* channel(P_1)=P_0.right
* $Locations = \{l0 : true, l1 : true, l_0^{P_1} : true, BAD : true\}$
* $initialLocation = l_0^{P_1}$
* $Edges = \{(l0, P_1.*.brake!, \emptyset, \emptyset, \{x^{P_1}\}, l1), (l1, P_0.\text{right.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_1} = 0, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_1.*.u?, n = 0 \wedge x^{P_1} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_2) is

* $Identifier = P_2$
* tpa(P_2)=Actuator
* channel(P_2)=P_0.left
* $Locations = \{l0 : true, l1 : true, l_0^{P_2} : true, BAD : true\}$
* $initialLocation = l_0^{P_2}$
* $Edges = \{(l0, P_2.*.brake!, \emptyset, \emptyset, \{x^{P_2}\}, l1), (l1, P_0.\text{left.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_2} = 0, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_2.*u?, n = 0 \wedge x^{P_2} > 0, \emptyset, \emptyset, BAD)\}$

Output: a monolithic analysis model of timed process automaton Brake-by-Wire

- Timed game analysis of root(P_0)||standalone(P_1)||standalone(P_2)||standalone(P_3)||standalone(P_4)||standalone(P_5)||standalone(P_6)
  - root(P_0) is
    - $Identifier = P_0$
    - tpa(P_0)=Brake-by-Wire
    - channel(P_0)=$\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\text{front.s}_{\text{Position}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_0.\text{rear.s}_{\text{Position}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\text{front.f}_{\text{Position}}?, \emptyset, \{n--\}, \emptyset, l0),$ $(l0, P_0.\text{rear.f}_{\text{Position}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\text{f}_{\text{Brake-by-Wire}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*$ $.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD), (l0, P_0.*.\text{signal1}?, \emptyset, \emptyset, \emptyset, l0),$ $(l0, P_0.*.\text{signal2}?, \emptyset, \emptyset, \emptyset, l0)\}$
  - standalone(P_1) is
    - $Identifier = P_1$
    - tpa(P_1)=Position
    - channel(P_1)=P_0.front
    - $Locations = \{l0 : true, l1 : true, l_0^{P_1} : true, BAD : true\}$
    - $initialLocation = l_0^{P_1}$
    - $Edges = \{(l0, P_1.*.a!, \emptyset, \emptyset, \{x^{P_1}\}, l1), (l0, P_1.\text{right.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$ $(l0, P_1.\text{left.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_1\text{right.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$

$(l0, P_1.\text{left.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, P_0.\text{front.f}_{\text{Position}}!, n = 0 \wedge x^{P_1} = 0, \emptyset, \emptyset, l_0^{P_1}),$

$(l_0^{P_1}, P_0.\text{front.s}_{\text{Position}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_1.*.u?, n = 0 \wedge x^{P_1} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_2) is

* $Identifier = P_2$
* tpa(P_2)=Position item channel(P_2)=P_0.rear
* $Locations = \{l0 : true, l1 : true, l_0^{P_2} : true, BAD : true\}$
* $initialLocation = l_0^{P_2}$
* $Edges = \{(l0, P_2.*.a!, \emptyset, \emptyset, \{x^{P_2}\}, l1), (l0, P_2.\text{right.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0),$

$(l0, P_2.\text{left.s}_{\text{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_2\text{right.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0),$

$(l0, P_2.\text{left.f}_{\text{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, P_0.\text{rear.f}_{\text{Position}}!, n = 0 \wedge x^{P_2} = 0, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\text{re}$

$.u?, n = 0 \wedge x^{P_2} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_3) is

* $Identifier = P_3$
* tpa(P_3)=Actuator
* channel(P_3)=P_1.right
* $Locations = \{l0 : true, l1 : true, l_0^{P_3} : true, BAD : true\}$
* $initialLocation = l_0^{P_3}$
* $Edges = \{(l0, P_3.*.brake!, \emptyset, \emptyset, \{x^{P_3}\}, l1), (l1, P_1.\text{right.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_3} = 0, \emptyset, \emptyset, l_0^{P_3}), (l_0^{P_3}, P_1.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_3.*.u?, n = 0 \wedge x^{P_3} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_4) is

* $Identifier = P_4$

* tpa(P_4)=Actuator
* channel(P_4)=P_1.left
* $Locations = \{l0 : true, l1 : true, l_0^{P_4} : true, BAD : true\}$
* $initialLocation = l_0^{P_4}$
* $Edges = \{(l0, P_4.*.brake!, \emptyset, \emptyset, \{x^{P_4}\}, l1), (l1, P_1.\text{left.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_4} = 0, \emptyset, \emptyset, l_0^{P_4}), (l_0^{P_4}, P_1.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_4.*u?, n = 0 \wedge x^{P_4} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_5) is

* $Identifier = P_5$
* tpa(P_5)=Actuator
* channel(P_5)=P_2.right
* $Locations = \{l0 : true, l1 : true, l_0^{P_5} : true, BAD : true\}$
* $initialLocation = l_0^{P_5}$
* $Edges = \{(l0, P_5.*.brake!, \emptyset, \emptyset, \{x^{P_5}\}, l1), (l1, P_2.\text{right.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_5} = 0, \emptyset, \emptyset, l_0^{P_5}), (l_0^{P_5}, P_2.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_5.*.u?, n = 0 \wedge x^{P_5} > 0, \emptyset, \emptyset, BAD)\}$

– standalone(P_6) is

* $Identifier = P_6$
* tpa(P_6)=Actuator
* channel(P_6)=P_2.left
* $Locations = \{l0 : true, l1 : true, l_0^{P_6} : true, BAD : true\}$
* $initialLocation = l_0^{P_6}$
* $Edges = \{(l0, P_6.*.brake!, \emptyset, \emptyset, \{x^{P_6}\}, l1), (l1, P_2.\text{left.f}_{\text{Actuator}}!, n = 0 \wedge x^{P_6} = 0, \emptyset, \emptyset, l_0^{P_6}), (l_0^{P_6}, P_2.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l0), (l1, P_6.*u?, n = 0 \wedge x^{P_6} > 0, \emptyset, \emptyset, BAD)\}$

Output: a compositional analysis model of timed process automaton Actuator

- Timed game analysis of root(P_0)
  - root(P_0) is
    - $Identifier = P_0$
    - tpa(P_0)=Actuator
    - channel(P_0)=$\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.brake!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l1, \perp.\mathsf{f}_{\mathsf{Actuator}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

Output: a compositional analysis model of timed process automaton Position

- Timed game analysis of root(P_0)||duration(P_1)||duration(P_2)
  - root(P_0) is
    - $Identifier = P_0$
    - tpa(P_0)=Position
    - channel(P_0)=$\perp$
    - $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
    - $initialLocation = l0$
    - $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\mathsf{right.s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0.\mathsf{left.s}_{\mathsf{Actuator}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\mathsf{right.f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l0, P_0.\mathsf{left.f}_{\mathsf{Actuator}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\mathsf{f}_{\mathsf{Position}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0 * .u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD)\}$

- duration(P_1) is
  - $Identifier = P_1$
  - tpa(P_1)=Actuator
  - channel(P_1)=P_0.right
  - *The WCET (input from the game analyzer)=Unknown*
  - $Locations = \{l_0^{P_1} : true, l_1^{P_1} : true\}$
  - $initialLocation = l_0^{P_1}$
  - $Edges = \{(l_1^{P_1}, P_0.\text{right.f}_{\text{Actuator}}!, \emptyset, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\text{right.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l_1^{P_1})\}$
- duration(P_2) is
  - $Identifier = P_2$
  - tpa(P_2)=Actuator
  - channel(P_2)=P_0.left
  - *The WCET (input from the game analyzer)=Unknown*
  - $Locations = \{l_0^{P_2} : true, l_1^{P_2} : true\}$
  - $initialLocation = l_0^{P_2}$
  - $Edges = \{(l_1^{P_2}, P_0.\text{left.f}_{\text{Actuator}}!, \emptyset, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\text{left.s}_{\text{Actuator}}?, \emptyset, \emptyset, \emptyset, l_1^{P_2})\}$

Output: a compositional analysis model of timed process automaton Brake-by-Wire

- Timed game analysis of root(P_0)||duration(P_1)||duration(P_2)
  - root(P_0) is
    - $Identifier = P_0$
    - tpa(P_0)=Brake-by-Wire

* channel(P_0)=$\perp$
* $Locations = \{l0 : true, l1 : true, l_0^{P_0} : true, BAD : true\}$
* $initialLocation = l0$
* $Edges = \{(l0, P_0.*.a!, \emptyset, \emptyset, \{x^{P_0}\}, l1), (l0, P_0.\text{front.s}_{\text{Position}}!, \emptyset, \{n++\}, \emptyset, l0),$
  $(l0, P_0.\text{rear.s}_{\text{Position}}!, \emptyset, \{n++\}, \emptyset, l0), (l0, P_0\text{front.f}_{\text{Position}}?, \emptyset, \{n--\}, \emptyset, l0),$
  $(l0, P_0.\text{rear.f}_{\text{Position}}?, \emptyset, \{n--\}, \emptyset, l0), (l1, \perp.\text{f}_{\text{Brake-by-Wire}}!, n = 0 \wedge x^{P_0} = 0, \emptyset, \emptyset, l_0^{P_0}), (l1, P_0.*$
  $.u?, n = 0 \wedge x^{P_0} > 0, \emptyset, \emptyset, BAD), (l0, P_0.*.\text{signal1}?, \emptyset, \emptyset, \emptyset, l0),$
  $(l0, P_0.*.\text{signal2}?, \emptyset, \emptyset, \emptyset, l0)\}$

– duration(P_1) is

* $Identifier = P_1$
* tpa(P_1)=Position
* channel(P_1)=P_0.front
* *The WCET (input from the game analyzer)=Unknown*
* $Locations = \{l_0^{P_1} : true, l_1^{P_1} : true\}$
* $initialLocation = l_0^{P_1}$
* $Edges = \{(l_1^{P_1}, P_0.\text{front.f}_{\text{Position}}!, \emptyset, \emptyset, \emptyset, l_0^{P_1}), (l_0^{P_1}, P_0.\text{front.s}_{\text{Position}}?, \emptyset, \emptyset, \emptyset, l_1^{P_1})\}$

– duration(P_2) is

* $Identifier = P_2$
* tpa(P_2)=Position
* channel(P_2)=P_0.rear
* *The WCET (input from the game analyzer)=Unknown*
* $Locations = \{l_0^{P_2} : true, l_1^{P_2} : true\}$
* $initialLocation = l_0^{P_2}$

$* \ Edges = \{(l_1^{P_2}, P_0.\mathsf{rear.f}_{\mathsf{Position}}!, \emptyset, \emptyset, \emptyset, l_0^{P_2}), (l_0^{P_2}, P_0.\mathsf{rear.s}_{\mathsf{Position}}?, \emptyset, \emptyset, \emptyset, l_1^{P_2})\}$

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether core_1 is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether core_1 is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether core_1 is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether core_2 is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether core_2 is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether core_2 is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether core_3 is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether core_3 is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether core_3 is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
uW	Boolean	To record whether an update in W is performed (1) or not (0)
uD	Boolean	To record whether an update in D is performed (1) or not (0)
L1	Integer	To record the current worst possible loads on core_1
L2	Integer	To record the current worst possible loads on core_2
L3	Integer	To record the current worst possible loads on core_3
vS	Integer	To record the current value in the speedometer
sD	Integer	To record the current door state
F	Integer	To record the current total number of core failures

Table D.1: Variables in the monolithic model

```
void start(int[1,3] task){
if (task==S){L1=L1+S1*aS1;      L2=L2+S2*aS2;   L3=L3+S3*aS3;   iS=1;}
else if (task==W){L1=L1+W1*aW1; L2=L2+W2*aW2;   L3=L3+W3*aW3;   iW=1;}
else if (task==D){L1=L1+D1*aD1; L2=L2+D2*aD2;   L3=L3+D3*aD3;   iD:=1;}}
```

Figure D.2: Function `start` in the monolithic and compositional models

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on $core_1$
lS2	Constant	To store the worst-case load of S on $core_2$
lS3	Constant	To store the worst-case load of S on $core_3$
lW1	Constant	To store the worst-case load of W on $core_1$
lW2	Constant	To store the worst-case load of W on $core_2$
lW3	Constant	To store the worst-case load of W on $core_3$
lD1	Constant	To store the worst-case load of D on $core_1$
lD2	Constant	To store the worst-case load of D on $core_2$
lD3	Constant	To store the worst-case load of D on $core_3$
Limit	Constant	To store the load limit of every core

Table D.2: Constants in the monolithic model

Name	From	To	Purpose
sS	service	S	To start S
sW	service	W	To start W
sD	service	D	To start D
fS	service	S	To finish S
fW	service	W	To finish W
fD	service	D	To finish D

Table D.3: Actions in the monolithic model

Name	Type	Purpose
x	Clock	To record time passed since S was initialized
y	Clock	To record time passed since W was initialized
z	Clock	To record time passed since D was initialized
aS1	Boolean	To record whether $core_1$ is currently assigned (1) to execute S or not (0)
aW1	Boolean	To record whether $core_1$ is currently assigned (1) to execute W or not (0)
aD1	Boolean	To record whether $core_1$ is currently assigned (1) to execute D or not (0)
aS2	Boolean	To record whether $core_2$ is currently assigned (1) to execute S or not (0)
aW2	Boolean	To record whether $core_2$ is currently assigned (1) to execute W or not (0)
aD2	Boolean	To record whether $core_2$ is currently assigned (1) to execute D or not (0)
aS3	Boolean	To record whether $core_3$ is currently assigned (1) to execute S or not (0)
aW3	Boolean	To record whether $core_3$ is currently assigned (1) to execute W or not (0)
aD3	Boolean	To record whether $core_3$ is currently assigned (1) to execute D or not (0)
iS	Boolean	To record whether S is initialized (1) or yet to initialize (0)
iW	Boolean	To record whether W is initialized (1) or yet to initialize (0)
iD	Boolean	To record whether D is initialized (1) or yet to initialize (0)
L1	Integer	To record the current worst possible loads on $core_1$
L2	Integer	To record the current worst possible loads on $core_2$
L3	Integer	To record the current worst possible loads on $core_3$
F	Integer	To record the current total number of core failures

Table D.4: Variables in the compositional model

```
void finish(int[1,3] task){
if (task==S){L1=L1-S1*aS1;      L2=L2-S2*aS2;   L3=L3-S3*aS3;   iS=0;}
else if (task==W){L1=L1-W1*aW1; L2=L2-W2*aW2;   L3=L3-W3*aW3;   iW=0;}
else if (task==D){L1=L1-D1*aD1; L2=L2-D2*aD2;   L3=L3-D3*aD3;   iD=0;}}
```

Figure D.3: Function `finish` in the monolithic and compositional models

Name	Type	Purpose
CFL	Constant	To store CFL
pS	Constant	To store release period of S
pW	Constant	To store release period of W
pD	Constant	To store release period of D
wS	Constant	To store the WCET of S
wW	Constant	To store the WCET of W
wD	Constant	To store the WCET of D
bS	Constant	To store the BCET of S
bW	Constant	To store the BCET of W
bD	Constant	To store the BCET of D
lS1	Constant	To store the worst-case load of S on core_1
lS2	Constant	To store the worst-case load of S on core_2
lS3	Constant	To store the worst-case load of S on core_3
lW1	Constant	To store the worst-case load of W on core_1
lW2	Constant	To store the worst-case load of W on core_2
lW3	Constant	To store the worst-case load of W on core_3
lD1	Constant	To store the worst-case load of D on core_1
lD2	Constant	To store the worst-case load of D on core_2
lD3	Constant	To store the worst-case load of D on core_3
Limit	Constant	To store the load limit of every core

Table D.5: Constants in the compositional model

```
void reassign(int[1,3] task, int[1,3] from, int[1,3] to){
  if (task==S && from==2 && to==1){aS2=0; aS3=0;  aS1=1;  L2=L2-iS*S2;     L1=iS*S1;          F--;}
  else if(task==S && from==3 && to==1){aS2=0;     aS3=0;  aS1=1;  L3=L3-iS*S3;     L1=iS*S1;      F--;}
  else if(task==W && from==1 && to==2){aW2=1;     aW3=0;  aW1=0;  L1=L1-iW*W1;     L2=iW*W2;      F--;}
  else if(task==W && from==3 && to==2){aW2=1;     aW3=0;  aW1=0;  L3=L3-iW*W3;     L2=iW*W2;      F--;}
  else if(task==D && from==1 && to==3){aD3=1;     aD2=0;  aD1=0;  L1=L1-iD*D1;     L3=iD*D3;      F--;}
  else if(task==D && from==2 && to==3){aD3=1;     aD2=0;  aD1=0;  L2=L2-iD*D2;     L3=iD*D3;      F--;}
  else if (task==S && from==1 && to==2){aS2=1;    aS1=0;  aS3=0;  L2=L2+iS*S2;}
  else if (task==S && from==1 && to==3){aS2=0;    aS1=0;  aS3=1;  L3=L3+iS*S3;}
  else if (task==W && from==2 && to==1){aW1=1;    aW2=0;  aW3=0;  L1=L1+iW*W1;}
  else if (task==W && from==2 && to==3){aW1=0;    aW2=0;  aW3=1;  L3=L3+iW*W3;}
  else if (task==D && from==3 && to==1){aD1=1;    aD1=1;  aD3=0;  L1=L1+iD*D1;}
  else if (task==D && from==3 && to==2){aD2=1;    aD1=0;  aD3=0;  L2=L2+iD*D2;}
  else if (task==S && from==2 && to==3){aS1=0;    aS2=0;  aS3=1;  L3=L3+iS*S3;}
  else if (task==S && from==3 && to==2){aS1=0;    aS2=1;  aS3=0;  L2=L2+iS*S2;}
  else if (task==W && from==1 && to==3){aW1=0;    aW2=0;  aW3=1;  L3=L3+iW*W3;}
  else if (task==W && from==3 && to==1){aW1=1;    aW2=0;  aW3=0;  L1=L1+iW*W1;}
  else if (task==D && from==1 && to==2){aD2=1;    aD1=0;  aD3=0;  L2=L2+iD*D2;}
  else if (task==D && from==2 && to==1){aD1=1;    aD2=0;  aD3=0;  L1=L1+iD*D1;}}
```

Figure D.4: Function `reassign` in the monolithic and compositional models