Project no.:	**ICT-FP7-STREP-214755**
Project full title:	**Quantitative System Properties in Model-Driven Design**
Project Acronym:	**QUASIMODO**

Deliverable no.: D5.12

Title of Deliverable: Industrial Handbook

Contractual Date of Delivery to the CEC:	**Month 39**
Actual Date of Delivery to the CEC:	**June 1, 2011**
Organisation name of lead contractor for this deliverable:	**ESI**
Author(s):	
Brian Nielsen, Kim G. Larsen	
Jan Tretmans	
Participants(s):	**All Partners)**
Work package contributing to the deliverable:	**WP 1-5**
Nature:	**R**
Version:	**1**
Total number of pages:	**242**
Start date of project:	1 Jan. 2008 **Duration:** 40 months

Project co-funded by the EC within the Seventh Framework Programme (2007-2013)
Dissemination Level

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for consortium members (including the Commission Services)	

Abstract:

Thus deliverable contains the latest draft of the Quasimodo Industrial Handbook. It is continuously updated on the internal Quasimodo Web-pages, and will be printed by Springer Verlag, Automn 2011.

Keyword list: Modeling, Analysis, Testing, Synthesis, Timing, Probabilities, Performance analysis, Tools, Case Studies

Contents

1. Introduction *by Brian Nielsen, Jan Tretmans, and Kim Larsen* **1**

2. Modelling Real-Time Systems using Uppaal *by Frits Vaandrager* **18**

3. More Features in UPPAAL *by Alexandre David and Kim G. Larsen* **49**

4. Industrial Application of Uppaal: The gMAC Synschronization Protocol *by Mathijs Schuts, Feng Zhu, Faranak Heidarian and Frits Vaandrager* **77**

5. Design of a Sage Real-Time Embedded System in Uppaal: The Self-Balancing Scooter Case *by Bert Bos, Jiansheng Xing, Teun van Kuppeveld, and Marcel Verhoef* **95**

6. An Introduction to Schedulability Analysis using Timed Automata *by Alexandre David, Arne Skou, and Kim Larsen* **107**

7. Schedulability Analysis using UPPAAL: The Herschel/Planck Satellite Software Case *by Marius Mikucionis, Brian Nielsen, Kim Larsen* **119**

New: Time and Cost Optimal Schedulling and Planning *by Nicolas Markey, Patricia Boyer and Kim G. Larsen* **133**

8. An Introduction to Automatic Synthesis of Discrete and Timed Controllers *by Jean-Francois Raskin, Franck Cassez, Pierre-Alain Reynier, Kim Larsen* **133**

9. Timed Controller Synthesis: An Industrial Case Study (The Hydac Case) *by Jean-Francois Raskin, Franck Cassez, Pierre-Alain Reynier, Kim Larsen* **150**

10. Probabilistic Analysis of Embedded Systems *by Arnd Hartmanns, Joost-Pieter Katoen, Holger Hermanns* **170**

11. Energy Consumption in the Chess WSN: A MoDeST Case Study *by Haidi Yue, Joost-Pieter Katoen, Holger Hermanns* **184**

12. Model-Based Testing *by Jan Tretmans, Brian Nielsen* **201**

13. Model-Based Protocol Conformance Testing: The Case of the Chess Wireless Sensor Network Node *by Marcel Verhoef, Jan Tretmans, Axel Belinfante* **226**

14. Experiences with Formal Engineering: Model-Based Specification, Implementation, and Testing of a Software Bus at Neopost *by Mariëlle Stoelinga, Marten Sijtema, Axel Belinfante and Lawrence Marinelli* **226**

15. Perspectives *by Kim Larsen* **240**

Introduction

Kim G. Larsen and Brian Nielsen and Jan Tretmans

Abstract The engineering of high quality embedded systems is challenging due to the ever inreasing demands for intelligent functionality that must operate efficiently and correctly under tight resource constraints.

The resulting complexity and pressure for time to market calls for new engineering methods and tools that supports early and integrated analysis of the systems behavior, performance, and reliability.

This handbook presents a set of promising advanced model-based engineering techniques and tools that goes beyond functionality and also address quantitative system properties. The book introduces specific techniques and tools for model-based analysis, synthesis and testing of timed and stochastic system properties. Moreover, it shows how to apply the techniques to solve complex industrial cast studies.

The handbook is targeted towards the professional embedded systems engineer, who would like to be introduced to promising advanced techniques for quantitative, model-based analysis of embedded systems, what techniques are available, which are mature enough to be applied, how they can be applied, and what benefits can be expected from using such a method, technique, or tool. The book introduces modelling and analysis from first principles, and advances this to more specialised applications.

Brian Nielsen
Aalborg University, Aalborg, Denmark, e-mail: bnielsen@cs.aau.dk

Jan Tretmans
Embedded Systems Institute, Eindhoven, The Netherlands, e-mail: jan.tretmans@esi.nl

1 Motivation

During the design and construction of a modern complex embedded system the engineer determines the best way to turn the requirements into a system design and further materialize this into a concrete implementation. At the same time he must asses whether his solution proposals are safe, efficient, and reliable. The engineer is thus continuously evaluating the consequences of different proposed solutions, and is faced with a number of concrete questions about the design, algorithm, and implementation.

Some examples of such questions that can be addressed by applying the techniques proposed in this book are:

- *Does a set of composed components correctly implement the desired behavior? Are there unforeseen races or deadlocks? Is the end-to-end timing of the system adequate to safely operate under the time constraints of the physical environment of the system. E.g., is it possible that a Segway may throw its user out of balance?*
- *What collision rate can be expected for a given a wireless sensor network medium access protocol? And what minimum gap separation should exist between two slots to avoid collision ?*
- *Is a collection of interaction tasks schedulable such that all tasks meet their deadline? What is the slowest execution time that can be afforded while still meeting deadlines?*
- *What is the best way to control a hydraulic pump safely while minimizing its energy consumption?*
- *How should a set of tasks be mapped to a set of computational resources to achieve optimal speed or minimal energy consumption?*

It is inherent in embedded systems that they in addition to functional requirements have to meet a multitude of *quantitative constraints*. Whereas traditionally development attention and effort has been focused on functional correctness of embedded systems, i.e., *qualitative properties* such as correct system responses to given stimuli, we currently see an increasing importance of *quantitative properties*[1]. These properties include:

- precise timing behavior and sufficient performance
- low energy consumption
- efficient use of computation memory, communication, and other resources
- assumptions about arrival rates of triggers
- availability, and other probabilistic aspects

Quantitative properties are crucial for the overall functioning of embedded systems, yet, they are difficult to design, build, analyze, and test. In addition, the quest for ever better products with stricter quantitative constraints, to be developed in less

[1] In other literature quantitative properties are sometimes referred to as “non-functional “ or “extra-functional” properties.

time and with less costs, puts high demands on the design and development process of future embedded systems.

Model-based, or model-driven[2] development and engineering approaches use models, or abstractions, for designing, reasoning, and analysing systems.

It is e.g., seen in the European Artemis Industry Association's research agenda as a key enabler for the development of high-quality complex embedded systems[ARTEMIS SRA WG(????a), ARTEMIS SRA WG(????b)]. During the last decade significant amounts of research have been invested into methods, modeling tools, and powerful automated analysis and synthesis tools. The approaches are also gaining popularity in the embedded systems industry and is applied for specific tasks.

However, most attention has been focused until now on modeling and analyzing functional, qualitative properties. Existing model-based approaches and tools are rather limited in their handling of quantitative constraints. Hence, model-based approaches must be extended to handle quantitative properties, in order to develop embedded systems that are not only functionally correct but that also satisfy their quantitative requirements.

It is the ambition of this book to introduce the reader to recent model-based methods, techniques and tools for the design, analysis, and testing of quantitative properties of embedded systems.More specifically, the book deals with

- *modeling of different functional and quantitative aspects of embedded systems like time, energy, cost, probability;*
- *techniques for analysing models with quantitative information;*
- *synthesis and generation of controllers for systems with quantitative constraints;*
- *testing and test generation for system implementations with respect to quantitative models.*

The book is intended for engineers and technical professionals involved in designed, analyzing, and testing embedded systems in which quantitative aspects play a major role. It is not intended to give a full scientific treatment of the area. Where appropriate references to further publications, such as journal and conference papers, are made.

This chapter introduces the area of model-based, quantitative analysis of embedded system. Moreover, it introduces the topics presented in the different chapters of this book, the relations between the topics, and their dependencies, thus providing a reading guideline for the book.

The remainder of the chapter is organized as follows. Section 2 gives the background for the book by characterizing the challenges of building embedded systems that satisfy the multitude of quantitative design constraints. It describes how model-

[2] We distinguish between model-*based* and model-*driven* development. In model-driven development all development activities are centered around and driven by models with increasing amounts of detail until code/hardware can be easily obtained. This represents a long term vision of a new development approach, but is also a big step from current development practices. In model-based development models are used to support development activities by automating specific tasks like correctness checking, performance analysis, test generation etc.

based development and the use of quantitative models may address these challenges. This book has been written an outcome of a European research project, Quasimodo. This is briefly described in Section 3. Section 4 describes the structure of the book and contains reading guidelines.

2 Model-based, Quantitative Analysis of Embedded Systems

2.1 Embedded Systems

ToDo: ESI research agenda? :
ToDo: Artist roadmap LNCS 3436? :

Embedded systems are systems in which a computer including software is embedded in some technical or physical context or system. They are not always perceived as computers: embedded computers may perform many tasks, but they are not always observed as such. The embedded computer controls the system, it observes and checks its functioning, it makes calculations, de- and encodes data, it processes data, e.g., obtained from the environment via sensors, and it influences the environment by controlling actuators.

There is a great variation of embedded systems; many we use on a daily basis without really noticing it (when they work). Others are more exotic. They rage from consumer products, like electric shavers, and washing machines, to professional equipment like wafer scanners. Some are manufactured in hundred thousand copies like mobile phones and television sets, whereas others are built only once, e.g., a satellite or a storm surge barrier control system. For some embedded systems such as DVD players, watches, or hearing aids, their malfunctioning is annoying and may cost their manufacturers market share. But if a pacemaker or the braking system of a car does not function as required, lives may be at stake, whereas an error in the automatic pilot system of an aircraft or in the control of a nuclear power plant may cause a catastrophy.

Thus, there are many different application domains, but the methods described in this book are general and not targeted towards specific domains. Depending on the risk of a particular product the methods may be more or less extensively applied.

Software in embedded systems shares many characteristics. A first important characteristic of embedded systems is that hardware and software have to interact with their technical and physical context in order to perform some task or provide some service: multi-disciplinarity plays an important role. The development of an embedded system requires the interaction of various technical disciplines such as hardware and software engineering, electrical and electronic engineering, mechanics and mechatronics, communication technology, (nuclear) physics, depending on the application domain. Multi-disciplinarity makes designing embedded systems a challenging task.

A second characteristic of embedded systems is that they are very often time critical. Whereas a 2-second delay in the calculation of an insurance premium calculation may annoy, a 2 second delay in delay in the trigger to close a valve in a

nuclear power plant can cause a catastrophy. Embedded systems are real-time systems.

In addition, in order to deal with different concurrent and interrelated tasks and with different input streams, embedded systems are often realized as a large number of concurrent interacting software components individually tracking parts of the environment state, and possibly running in parallel on multiple execution units. concurrent. This gives rise to intricate communication and failure scenarios.

Embedded systems often have special purpose components and hardware, such as special purpose sensors or actuators, adapted to specific contexts. Embedded systems also must be developed with limited resources and budget to meet selling cost targets.

Therefore, embedded systems tend to quickly grow in their complexity, both in the number of components of a system, and in the complexity of the individual components. It is consequently very difficult to comprehend all the interactions between the components, and in particular avoiding undesired interactions, such as heat production or electromagnetic interference.

ToDo: can we really solve this? :

2.2 *Quantitative Properties*

On the one hand, embedded systems must perform the function they are supposed to do (functional requirements), e.g., open a valve under high temperature conditions, or report the location of a flower-cart.

On the other hand, embedded systems must also satisfy a multitude of quantitative constraints as illustrated by Figure 1, that determines how well it performs its functions. By a quantitative constraints we mean a constraint that can be measured and expressed as a number. Existing development techniques tend to emphasize only functionality and is weak in correct construction of the quantitative aspects [Henzinger and Sifakis(2006)].

ToDo: maybe use [Henzinger and Sifakis instead? :

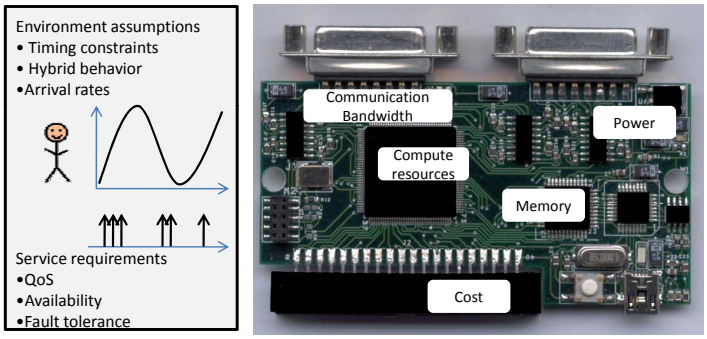


Fig. 1: Quantitative Resources and Constraints in Embedded Systems

A quantitative constraint occurs because the environment including users of the systems has a number of demands on the resources of the system, whereas the system has a limited set of available resources to perform its task.

One important quantitative aspect is the response time or latency of the system. Since an embedded system interacts with a physical environment it needs to keep up with this. In a hard real-time system no violations of deadlines can be tolerated, in a firm real-time system deadline misses are very undesirable, but an occasional miss is not catastrophic. In a soft real-time system most response times should be before the deadline; sometimes the desired response times are further characterized by a probability distribution of its response times with specified means and percentiles.

ToDo: continuous signals :

Another example is throughput (number of tasks completed per time unit, number of packets sent or received per time unit). Sometimes the term "goodput" is used to distinguish successfully completed tasks versus initiated tasks (to differentiate e.g., bandwidth of correctly received messages versus bandwidth used by the sender and wasted due to collisions and message corruption). Again requirements can be formulated in worst-case or probabilistic terms.

Further, reliability parameters (like (mean) time between failures and (mean) time to repair), and availability (the fraction of time the system is correctly operational) are also quantifiable, but probabilistic in nature.

During execution the system consumes various resources. This includes platform resources such as CPU-time, memory, bandwidth, and energy. But it may also be scarce resources of the system's environment such as special lifting cranes, mixing stations, or mechanical wear that vary depending on the reactions of the system. The use of these/such resources can be quantified, and treated generally as different kinds of costs.

It is interesting to note that there often are tensions between the different quantities, and consequently engineering trade-offs to be made. For instance, it is possible to improve response time by using a faster processor (or using dynamic voltage scaling) but at the cost of a superlinear increase in energy consumption. The latency of a wireless sensor network may be improved but at a cost of increasing collision rate and energy consumption.

It is an non-trivial engineering task to predict these quantities, and to optimize and balance them.

To focus on quantitative aspects like performance, timeliness, and efficient resource-usage, which are central to embedded systems, the models must provide quantitative information such as information about timing, cost, data, stochastics and hybrid phenomena.

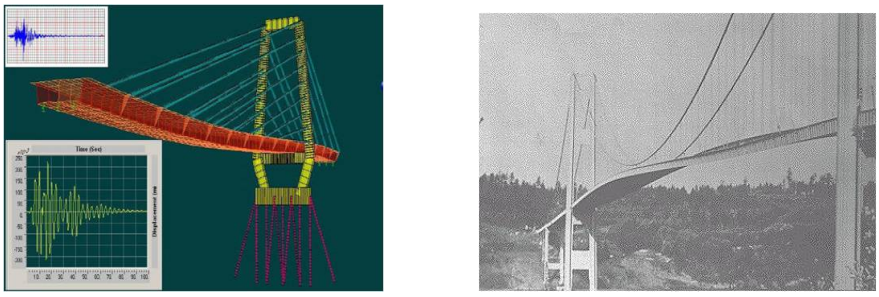
2.3 Models and model-based development

In our basic approach for designing, analyzing, and testing embedded systems is the use of *models*. A model is an abstract view of reality, in which essential proper-

ties are recorded, and other properties and details considered not important for the problem at hand, are removed.

An every-day example of a model is a road map: a road map only contains lines representing roads, and circles representing cities. The map abstracts from many other details of reality, such as buildings, forests, railways, mountains, the width and the kind of pavement of roads, et cetera. Such a map, i.e., a model, may very well help with planning your trip by car from Brussels to Paris, because all relevant details for such a trip are there. For planning a railway trip, however, or for calculating the altitude difference between Brussels and Paris, such a road map is useless. Another map, i.e., another model with other abstractions, is needed such as a railway map, or a geographic map, respectively.

As another example consider civil engineers building large complex structures like bridges. Here it is customary (mandatory) to use and analyze various model before construction. For instance, a static load model is used to predict whether the bridge can sustain itself and a certain number of cars. A dynamic model is used to investigate the dynamic effects of wind and moving cars and trucks.



(a) Enginering model of a bridge. (b) A bridge implementation

Fig. 2: Civil engineering analogy.

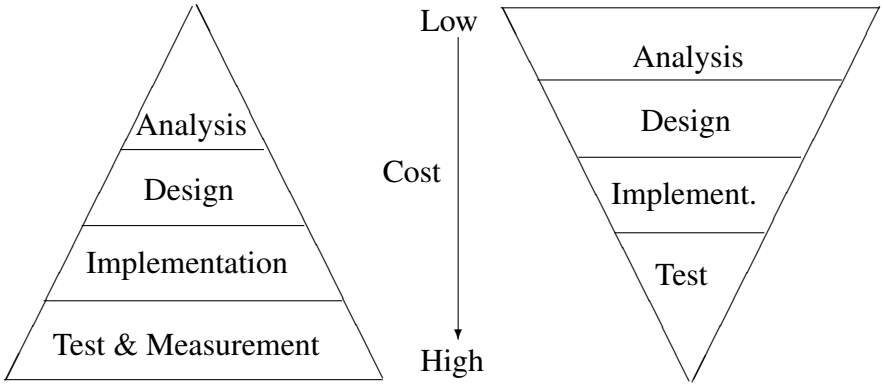
In software engineering models nowadays emphasize the structure of the system (i.e., the relationships between its components), but very little analysis, calculation, and prediction about the behavior and performance of the system is being made. Satirically [3] speaking, if software engineers made bridges they would start immediately by pouring concrete, and the next morning run a heavy truck onto the bridge to test its load carrying capabilities. Most likely it crashes, so they rebuild the bridge slightly stronger in the troublesome area. In contrast, using models for prediction is part of any mature engineering discipline.

Models can be analyzed, they can be the basis for calculations, they can help in understanding a problem, they can form the basis for constructing a system, for

[3] Longman's dictionary of contemporary English: a way of criticizing something such as a group of people or a system, in which you deliberately make them seem funny so that people will see their faults.

testing it, or for diagnosing it, and when they are expressed in an executable language, they can be simulated (simulation models). Models can be made *a priori* to guide and analyze the design, or *a posteriori* to analyze, test, or diagnose an existing system. Different models can be made of the same system, each focusing on a different kind of properties, e.g., a functional model, a performance model, or an energy-consumption model.

ToDo: Relate to OMG's MDA, and AADL? :



(a) Traditional (b) Model-based.

Fig. 3: Effort and Cost in Model Based Development

In current development practice, comparatively more effort is spent in the later development activities, and on testing in particular. Especially the quantitative properties are established by measurements and tests after the system has been built. This is unfortunately, because corrections or pursuing alternatives require costly redesign, re-implementation, and re-testing iterations. Moreover, it is well known that defects are much more expensive to correct the later they are found (most costly during deployment). This is schematically shown in Figure 3a.

It is the thesis that by putting more effort into early analysis and design exploration less emphasis is required late because of less testing and tuning as depicted in Figure 3b. The result is overall less cost.

In most projects, requirements change (or are discovered) during system development. Model based development accommodates this well; obviously models must be updated, but making these changes and (automatically) analyzing their consequences is easier and cheaper at the level of models than on complete designs and implementations.

It is beyond the scope of this book to discuss life cycle models like the V-model, multiple-V, iterative, agile, etc. but we remark that model based development goes well in hand with iterative and agile development methods. Models may be grown and refined iteratively like other design and implementation artifacts.

A main benefit of model-based development is that given a concise model, advanced tools can analyze it and help automate a number of difficult engineering tasks, see Figure 4.

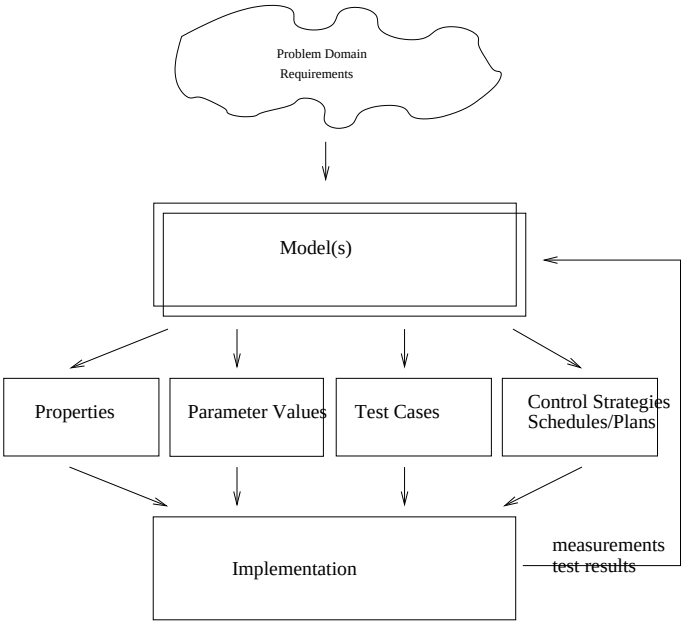


Fig. 4: Establishment of quantitative properties using MDD.

Models are made by the engineer to examine the properties (derived from problem- and application domain requirements) of a design or solution proposal for the system he is building, and to gain confidence in its correctness and fidelity.

Given such a proposed model appropriately represented in a notation with precise syntax and semantics, advanced design tools can interpret this, and automatically check it against the (formalized) high-level properties and requirements the system must satisfy. The engineer will consequently therefore have high confidence in how to build the system correctly.

The properties that are shown to hold on the level of models will then also hold on the implementation if it is carefully constructed from the models. Further, the models can be used to determine values for important system parameters (Like time-out values, slot durations, thresholds etc.) such that important safety and resource constraints are met.

In many cases, these tools can help *synthesize* parts of the implementation such as execution plans describing the order in which tasks must be performed to make

best use of system resources, or control strategies that can be directly implemented in the embedded controller. In some cases direct executable code can be obtained[4].

Moreover, model-based testing can be used to relate the behavior of models to that of real implementations. By generating test cases from the models and by executing these against the implementation, confidence can be gained in the operation of the actual implementation, and that the behavior of the implementation conforms to that prescribed by the models.

Finally, information obtained from the actual implementation can be fed back into the models and the resulting modified models can be used to re-check important system properties.

In all cases the validity of the statements made on the basis of the models requires that they correctly reflect the implementation, and contain the right assumptions.

2.4 Modeling and Analyzing Quantitative Properties

Although the modeling effort itself often spawns many interesting questions to a design or solution strategy, the strongest gain comes when tools can interpret them and compute useful and trustworthy analysis or synthesis results. It is therefore of great importance that the models *precisely* and unambiguously capture functional and quantitative behavior, and are accurately *analyzable* by computers. More or less ad-hoc annotations to existing modeling frameworks, or simulations with fuzzy semantics, does not satisfy these goals.

To this end the book is using automata extended with capabilities for expressing timed, hybrid, priced, and probabilistic behavior as preferred notations. As a small appetizer illustrating the kind of quantitative models used in this book, consider the data transmission system depicted in Figure 5. It consists of a sender component sending an amount of data to a receiver via a simple communication medium. The goal is to transmit N chunks of data.

The behavior of the sender and receiver components is given by the automaton in Figure 6. A transmission is successful when N chunks have been received (counted by integer variables i and j). The medium accepts request for transmitting data chunk from the sender and indicates to the receiver when the data chunk is ready for delivery.

One basic model of the medium that reliably forwards the data chunk is given in Figure 7a. However, this model abstracts away important aspects like transmission time, error rate, and price. Figure 7b adds timing behavior by using a special "clock" variable x and states that it takes between 5 to 10 time units to forward the message.

[4] We note that in general code generation for complete complex embedded systems that meet qualitative as well as quantitative constraints is difficult. Models for code generation typically need to be very low-level and detailed as they must prescribe not only *what* the system is required to do, but *how* it is going to achieve this. E.g., recognizing a car and checking the property that it can go 200 km/h, is simpler than constructing one that will safely go 200 km/h. Yet, code generators exists for a number of special cases.

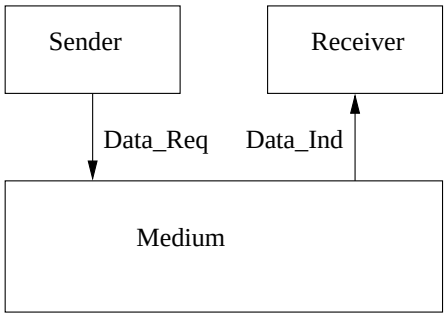


Fig. 5: Structure of a simple transmission system

(a) Sender (b) Receiver

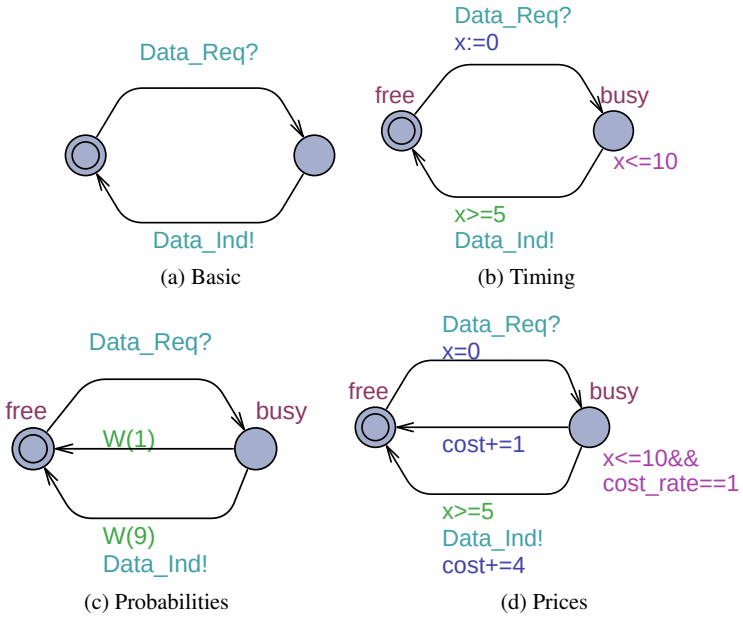
Fig. 6: Sender and Receiver Components

Figure 7a adds message loss probability by stating that with probability 0.1 the medium silently moves back to the starting state. A final aspect of the medium may be price of transmitting a message shown in Figure 7d. Price may represent many different costs, e.g., as monetary cost or, as intended in the example, as energy consumption tracked by the special `cost` variable.

Given precise models of these quantitative aspects, the following examples of system properties can be answered automatically by analysis tools:

- What is the best- or worst-case total (successful) transmission time?
- What is the minumum probability of a successful transmission?
- What is the best- or worst-case energy consumption of a successful transmission?
- What is the average successful transmission time?

It is important to realize that not all interesting problems (esp. quantitative analysis) can be solved efficiently (or at all) by an algorithm realizable on a computer (small or big). The necessary restrictions shows up in the syntax of the models and in the tasks and problem sizes that are manageable by current tools. Therefore, creating analyzable models that reflects the aspects of interest requires insight and also experience. This book therefore also provides advice for how to make analyzable models.



(a) Basic (b) Timing

(c) Probabilities (d) Prices

Fig. 7: Different Quantitative Aspects of a Simple Transmission Medium

3 The Quasimodo Project

This book is one of the results of the Quasimodo project. Quasimodo– *Quantitative System Properties in Model-Driven-Design of Embedded Systems* – was a European research project running from January 2008 until April 2011, and supported by the European Union [**?**].

The principle aim of the project is to provide a coherent and scalable methodology with a supporting collections of tool components that can be used to design reliable embedded systems that meet their requirements in a controlled and resource-efficient way using a model-driven approach.

The main scientific challenges tackled in the project were:

1. How to improve the modeling of diverse quantitative information (real-time, hybrid, stochastic, and resource consumption) of embedded systems in convenient models with theoretically well-founded semantics.
2. How to provide a wide range of efficient algorithmic techniques and symbolic data structures for analyzing models with quantitative information and for establishing abstraction relations between them.
   This is particularly challenging because many quantitative analysis problems (esp. involving multiple quantities) are on border of what is computationally feasible, either because they require algorithms of high complexity, or can right out be proven to be uncomputable by any algorithm.

3. How to materialize the techniques in usable tools for analysis, test and controller/code generation.
4. How to generate predictable code from quantitative models. The theoretical framework of the quantitative models assumes infinitely fast hardware, infinitely precise clocks, unbounded memory etc. In contrast real CPUs are subject to hard limitations in terms of frequency and memory-size. Thus, how to guarantee that properties established by a given model are also valid of its implementation is a major challenge.
5. How to improve the overall quality of testing by using suitable quantitative models as the basis for generating sound and correct test cases.
   Current industrial testing practice is often manual without effective automation and is consequently rather error prone and costly: it is estimated that 30-70% of the total development cost is related to testing.
   Model-based testing is a novel approach to testing with high potential of improving cost and efficiency, but the techniques must be extended to quantitative models allowing generation, selection, execution and provision of coverage-measures to be made.

In order to demonstrate their usefulness of the techniques, a further challenge is to apply them to complex industrial case studies.

Quasimodo has developed new theories, techniques and tool components for accurate and trustworthy analysis of quantitative constraints in model-driven development of embedded systems, covering in particular real-time, hybrid, priced and stochastic aspects. As a further result, the boundary of the quantitative problems (both in terms of size and diversity) that can feasibly be algorithmically handled has been significantly pushed. Moreover, a number of industrial case studies was tackled where the developed techniques and tools were applied.

Quasimodo has thus advanced the state-of-the-art of models, tools and methods for quantitative design, and a selection of the techniques are presented in this book.

The industrial partners of Quasimodo were

Hydac Electonic Gmbh (Germany): HYDAC operates worldwide, offering an extensive product range to cover all areas of fluid technology. The core competence is the design and production of sensors and controllers that are specialized for the extreme rough ambient conditions in hydraulic applications. These applications require substantial know-how in design of safety critical software for industrial and automotive purposes.

Chess B.V. (The Netherlands): Chess develops business critical embedded systems using novel ICT for high-tech customers in the various domains (banking, insurance, transport, trade and industry). Chess believes strongly in advanced research and development, and supports development by an “Innovation Team”.

Terma A/S (Denmark): Terma is a leading Danish company for developing mission critical solutions to the aerospace, defense, and security industry; by nature these depend highly on correctness and robustness.

There were 8 academic partners. Aalborg University (Denmark, coordinator) has particular expertise in techniques and tools for analyzing, testing and synthesizing

real-time systems, and is a main developer of the timed automata based UPPAAL tool suite.

The Embedded Systems Institute (The Netherlands) is a private/public research with significant experiences in research based on industrial case studies, and in industrial collaboration and dissemination. Radboud University (The Netherlands) is an expert in modeling and analyzing complex real-time systems, protocols and components. University of Twente (The Netherlands) is strong in applications of formal methods, especially testing, and the developer of the Torx model-based testing tool.

ENS-Cachan/CNRS (France) is focuses its research on the verification of critical software and systems, as has have very good reputation in the theory of timed automata and priced extensions.

RWTH Aachen University (Germany) has expertise in integrating formal specification and analysis techniques with performance and dependability. Universität des Saarlandes (Germany) has particular strengths in approximate, simulation-based analysis techniques for stochastic timed systems, in combination with rigorous, real-time model checking. Combineded these two groups are leading in analysis of probabilistic and stochastic systems, and is behind several tools for their analysis, including the MRMC and MODEST/MOTOR tool environments.

Université Libre de Bruxelles (Belgium) makes strong contributions to analysis of complex hybrid systems, controller synthesis, and implementability of timed automata on physical hardware.

Combined the partners have leading competences in developing embedded software, a solid theoretical foundation, and a broad spectrum of techniques and tools for quantitative analysis of embedded systems.

4 Overview of the Book

4.1 Contents

This book is divided into **??** chapters that covers 5 topics corresponding to different goals of model-based development:

Modeling and model-checking (of timed systems): Chapter **??** introduces modeling and model-checking from first principles using the notion of timed automata and the UPPAAL tool. Chapter **??** introduces more advanced UPPAAL features and shows how modeling the same problem can result in different models that more or less directly reflect the problem, and in models of very different (state-space) sizes, and thus leading to more or less costly analysis. Chapter **??** presents an advanced industrial case study of analyzing clock synchronization in a medium access protocol for wireless sensor networks. It both identifies a flaw in the clock synchronization algorithm and proposes an alternative. Chapter **??** reports on the experiences that have been obtained by (mainly) industrial engineers from modeling and analyzing a self balancing scooter.

Schedulability analysis: This part proposes a model-based approach to schedulability analysis, i.e., determining whether a set of (periodic) always meet their deadlines even using their worst case execution time. Chapter **??** presents the modeling framework, and Chapter **??** reports on the experiences of applying it to time critical satellite software.

Controller synthesis: This part aims at explaining how model-based techniques can be used to generate parts of the implementation, namely the heart of the controller: the control strategy. Chapter **??** introduces the principles of controller synthesis, and Chapter **??** shows a case study where a safe and energy optimal controller for a hydraulic plastic molding machine has been algorithmically synthesized.

Probabilistic modelling and performance analysis: Chapter **??** introduces the principles of modeling (potentially timed) probabilistic and stochastic aspects of embedded systems using the Modest language. Further it, shows how model-checking and simulation tools can be used for performance analysis. The presentation is accompanied by a small communication system example. Chapter **??** presents a case study of probabilistic analysis of the energy consumption in wireless sensor networks. It studies the trade-offs between latency and energy consumption under different sending strategies.

Model-based testing: Chapter **??** presents the concept of model-based testing, gently introduces the underlying theory and algorithms, and demonstrates its application to a small example. Chapter **??** contains a case study of performing real-time testing of the (MAC LAYER) a wireless sensor node.

System Engineering: Chapter **??** shows how (formal) model-based development, especially emphasizing model-checking and model-based testing, can be used *during* system development. The chapter is based on a case study where the goal was to develop a server component of a software bus for shipping and mailing applications.

Perspectives and outlook:

4.2 Reading guideline

Chapters are independently readable, but have strong connection. For each topic there is a chapter that introduces the principles of the topic, and an application chapter.

Readers inexperienced with modeling and model-checking is highly recommended to start with reading Chapter **??** followed by Chapter **??**.

If the goal is mainly performance modeling and analysis we recommend Chapters **??** and **??**.

If the aim is to gain insight into how model-based techniques can be used to aid advanced engineering tasks, without necessarily fully understanding the underlying techniques, the reader is recommended to go directly to reading the case study chapters: **??**, **??**, **??**, **??**, and **??**.

4.3 Additional Resources

Accompanying this book is a website containing a selection of the models used in this book. Also it provides links to tools, slides, etc.:

`http://www.quasimodo.aau.dk/QuasimodoBook`

4.4 Acknowledgments

The editors would like to thank the contributing chapter authors for their great effort. Also we would like to thank the Quasimodo researchers for an inspiring project, and for the many good technical presentations and discussions.

We would like to thank XXX for reading and commenting on the chapters.

References

[ARTEMIS SRA WG(????a)] ARTEMIS SRA WG (????a) The artemis strategic research agenda. URL http://www.artemis-ia.eu/downloads/SRA_MARS_2006.pdf, checked April 29, 2011

[ARTEMIS SRA WG(????b)] ARTEMIS SRA WG (????b) The artemis strategic research agenda - design methods & tools. URL http://www.artemis-ia.eu/downloads/RAPPORT_DMT.pdf, checked April 29, 2011

[Bouyssounouse and Sifakis(2005)] Bouyssounouse B, Sifakis J (2005) Embedded Systems Design: The ARTIST Roadmap for Research and Development, Lecture Notes in Computer Science, vol 3436. Springer

[Henzinger and Sifakis(2006)] Henzinger TA, Sifakis J (2006) The embedded systems design challenge. In: Misra J, Nipkow T, Sekerinski E (eds) FM, Springer, Lecture Notes in Computer Science, vol 4085, pp 1–15

[Henzinger and Sifakis(2007)] Henzinger TA, Sifakis J (2007) The discipline of embedded systems design. IEEE Computer 40(10):32–40

Chapter 1
A First Introduction to Uppaal

Frits Vaandrager

Abstract This chapter provides a first introduction to the use of the model checking tool Uppaal. Uppaal is an integrated tool environment that allows users to model the behavior of systems in terms of states and transitions between states, and to simulate and analyze the resulting models. Uppaal can also handle real-time issues, that is, the timing of transitions. Using an example of a jobshop, we explain in a step by step manner how one can make a simple Uppaal model, simulate its behavior and analyze properties.

1.1 Introduction

1.1.1 Model Checking

Model checking [5, 3, 1] is a powerful technique for automated debugging of complex reactive systems such as hardware components, embedded controllers and network protocols. In model checking, specifications about a system are expressed as (temporal) logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. In 2007, E.M. Clarke, E.A. Emerson and J. Sifakis were awarded the ACM Turing Award for their roles in developing model checking into a highly effective verification technology, widely adopted in industry.

Model checkers allow one to analyze models that capture the *dynamic behavior* of systems. These can be all sorts of systems: a network of computers or a printer, a Sudoku puzzle or an ant colony, an autonomous robot or train control software, etc. Actually, in principle any system can be analyzed using a model checker, as long

Frits Vaandrager
Institute for Computing and Information Sciences, Radboud University Nijmegen, Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands, e-mail: F.Vaandrager@cs.ru.nl.

as it has *states* and *transitions* between states. In order to illustrate the concept of model checkers, we consider the following puzzle:

> Six girls each know a secret. By means of a series of bilateral conversations (regular phone conversations, say) they want to exchange all secrets. Whenever two girls have a conversation they share all the secrets they know at the time. How many conversations are needed before every girl knows every secret?

Figure 1.1 shows a *state-transition diagram* or *automaton* for the simplified version of the problem in which there are 3 girls. A *state* consists of a 3 by 3 matrix `knows` that records for each girl the gossips she knows. If girl i knows the gossip of girl j then we write a 1 in the entry for row i and column j: $\texttt{knows[i][j]} == 1$. Otherwise we write a 0: $\texttt{knows[i][j]} == 0$. In the *initial state*, at the top of the diagram, each girl only knows her own gossip, and hence there are 1's on the diagonal and 0's elsewhere. *Transitions* between states occur whenever girls have a conversation. In the initial state three transitions are possible: girls 1 and 2 call each other,

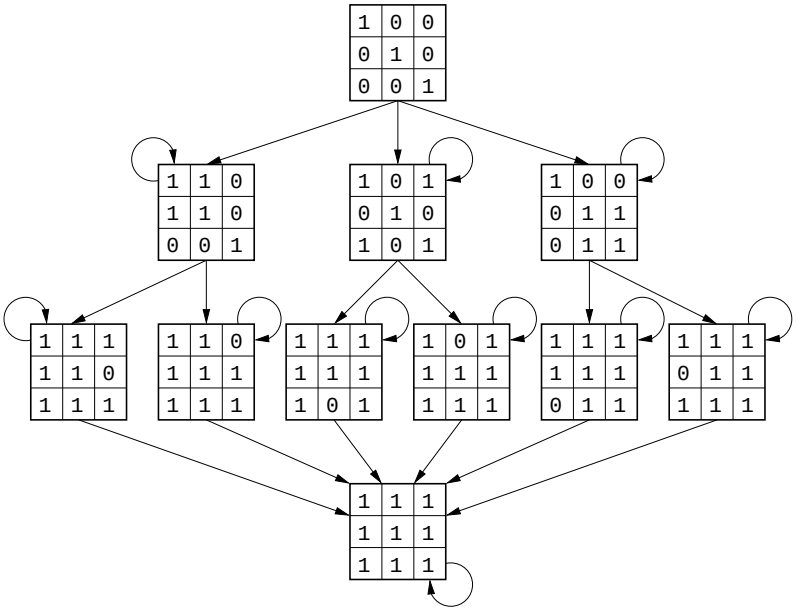


Fig. 1.1 State space for 3 gossiping girls.

girls 1 and 3 call each other, and girls 2 and 3 call each other. In the new states, the rows for the corresponding girls are adjusted and all gossips are exchanged. Not all conversations lead to a new state: sometimes a conversation does not produce any

new information for any of the participants, and there is just a loop from the state to itself. Altogether 11 states can be reached starting from the initial state, and at least 3 conversations are needed before every girl knows every gossip.

Properties of state-transition diagrams can be described in the language of *temporal logic*. For instance, if φ is a property of states, then the temporal logic formula $\mathbf{E}\diamond\ \varphi$ describes the property "there exists a path that leads to a state in which φ holds". Model checkers can compute whether a temporal logic formula holds for (the initial state of) a given state-transition diagram. We can solve the gossiping girls puzzle by asking a model checker to produce the shortest diagnostic trace that shows that the formula $\mathbf{E}\diamond$ "each girl knows each gossip" holds for the state-transition model for 6 girls. In the syntax of the model checker Uppaal:

```
E<> forall (a : girls) forall (b : girls)
                knows[a][b] == 1
```

Figure 1.2 shows the solution for the gossiping girls puzzle that was found by Uppaal: at least 8 conversations are needed before every girl knows every secret. In fact, the message sequence diagram displayed in Figure 1.2 has been directly generated by the Uppaal tool.

Although utterly simple, the gossiping girls example already illustrates several key features of model checkers:

1. No proofs. Mathematicians have established that, in general, n girls need at least $2n-4$ conversations to exchange all gossips [8]. The result of [8] required ingenuity and hard work, whereas Uppaal produces its solution for the special case $n=6$ fully automatically.
2. Fast. Using his of her favourite model checker, an experienced user can construct a model for the gossip puzzle within half an hour. Uppaal needs a few minutes to finding the optimal solution for $n=6$, and just a few seconds to find an optimal solution for the cases $n<6$.
3. Diagnostic counterexamples. Model checkers are very good at finding unexpected scenarios. When trying to solve the gossip puzzle, most people quickly come up with a solution that requires 9 conversations. The optimal solution found by Uppaal, which only requires 8 conversations, is tricky and much harder to find for humans. In industrial applications, model checkers often produce unexpected event orders that lead to an error state, fast solutions for scheduling problems, etc. The diagnostic counterexamples produced by model checkers frequently provide key insights in a design.
4. State space explosion. Since states of the model for n girls are $n\times n$ Boolean matrices, the number of states of the model will be in the order of 2^{n^2} and hence grows exponentially in n. Using some special verification features (symmetry reduction and the sweepline method), Uppaal can handle up to 7 girls (see Chapter 2 **Add ref**). Despite these limitations, in practice we often see that analysis of small instances of a parametrized model already produces important insights. It is trivial to generalize the solution with 8 conversations for $n=6$ of Figure 1.2 to a general solution with $2n-4$ conversations for $n>6$. Similarly, if a design

contains a flaw then usually this flaw can already be revealed by model checking a small instance of the design.

1.1.2 Uppaal

This chapter provides a first introduction to the use of the model checking tool Uppaal. Uppaal is an integrated tool environment that allows users to model the behavior of systems in terms of states and transitions between states, and to simulate and analyze the resulting models. Uppaal can also handle real-time issues, that is, the timing of transitions. Using an example of a jobshop, we will explain in a step by step manner how one can make a simple Uppaal model, simulate its behavior and analyze properties.

Uppaal is available for free for non-commercial applications in academia and for private persons via `www.uppaal.org`. For commercial applications a commercial license is required, see `www.uppaal.com`. The software runs under Windows,

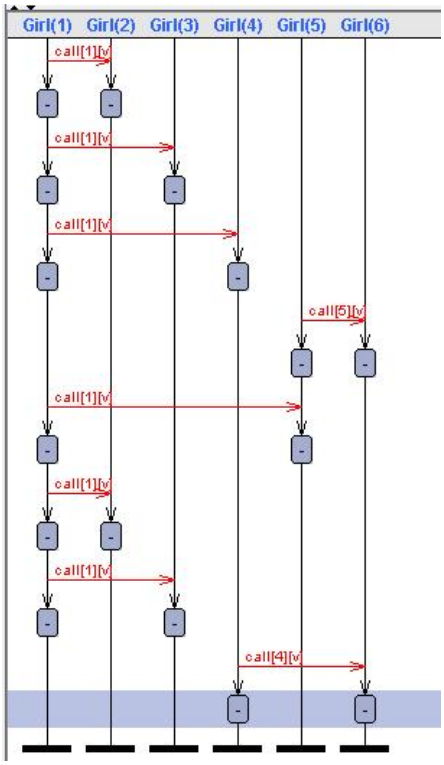


Fig. 1.2 Solution for gossiping girls puzzle found by model checker Uppaal.

Linux and Mac OS X. Installation is usually trivial. The only requirement is that Java version 6 (e.g. J2SE Java Runtime Environment) or newer has been installed and properly configured on the system. Uppaal is developed in collaboration between the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark, with input from several other universities around the world (including the author's group from the Radboud University Nijmegen).

Uppaal is a toolkit with a wealth of possibilities to model and analyze systems. In this chapter, we will restrict attention to the absolute basics. The next chapter of this handbook will give an overview of some of the more advanced features of Uppaal. More details and documentation can also be found on the Uppaal website and in the tutorial paper [2] (the last article requires some background in formal methods). The help menu within Uppaal also provides an excellent explanation of the various features and possibilities of the tool. Actually, some text in this chapter has been directly taken from the help menu.

There are numerous other model checkers that one can freely download from the internet: Spin, Blast, MCRL2, Java Pathfinder, etc. We refer to [11] for an overview. Advantages of Uppaal are the graphical user interface and the short learning curve. After you have spent half a day on reading this tutorial and following the detailled instructions for building and analyzing some simple models, you can already start using the tool yourself.

1.2 A jobshop

We will now explain step by step how a simple production line can be modeled in Uppaal. This example is due to Robin Milner; we have taken the following description and illustration of Figure 1.3 from [10]:

> We suppose that two people are sharing the use of two tools — a hammer and a mallet — to manufacture objects from simple components. Each *object* is made by driving a peg into a block. We call a pair consisting of a peg and a block a *job*; the jobs arrive sequentially on a conveyor belt, and completed objects depart on a conveyor belt. The jobshop could involve any number of people, whom we shall call *jobbers*, sharing more or fewer tools. In this chapter, we assume a system with two jobbers and a hammer and a mallet. To make the example more specific, we shall assume that the nature of the job influences the jobber's actions in a particular way. We suppose that he may use two predicates *easy* and *hard* over jobs, to determine whether a job is easy or hard or neither. He will do easy jobs with his hands, hard jobs with the hammer, and other jobs with either hammer or mallet.

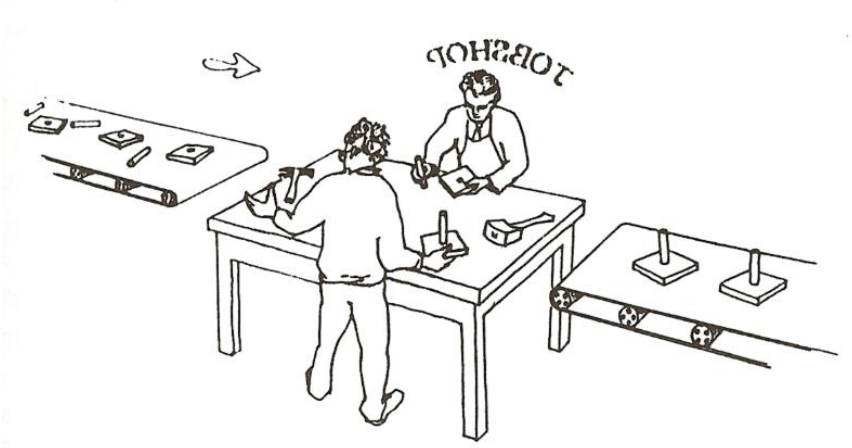


Fig. 1.3 A jobshop (picture taken from [10]).

1.3 The system editor

After starting Uppaal, we see the window displayed in Figure 1.4. The Uppaal graphical user interface consists of three main parts, accessible via three tabs in the main window: the *system editor*, which can be used to construct models, the *simulator*, in which the behavior of models can be simulated, and the *verifier*, in which the behavior of models can be analyzed. In this subsection we discuss the system editor, subsection 1.4 will present the simulator, and subsection 1.6 will present the verifier. Upon starting Uppaal, at first the system editor is displayed.

A Uppaal model (called *system*) is defined as a composition of a number of basic components (called *automata* or *processes*). Automata are diagrams with states (called *locations*) and transitions between states (called *edges*).

The system editor has four drawing tools for building automata, see Figure 1.5, named *Select*, *Location*, *Edge* and *Nail*.

- The *Select tool* is used to select, move, modify and delete elements. Elements can be selected by clicking on them or by dragging a rubber band around one or more elements. Elements can be added or removed from a selection by holding down the control key while clicking on the element. The current selection can be moved by dragging them with the mouse. Double clicking an element brings up a menu in which properties for that element can be specified. Right clicking an element brings up a pop-up menu from which properties of the element can be changed.
- The *Location tool* is used to add new locations. Simply click with the left mouse button in order to add a new location. Be careful: if one clicks several times new locations will be stacked on top of each other (a common mistake, leading to

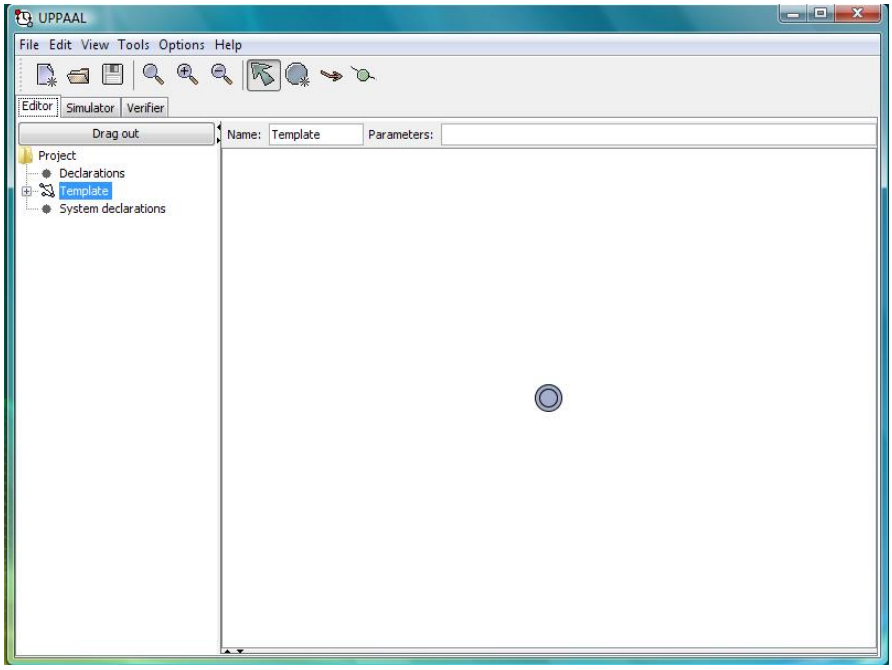


Fig. 1.4 Uppaal after starting the toolkit.

Fig. 1.5 The four drawing tools *Select*, *Location*, *Edge* and *Nail*.

models with strange behavior). In order to move a location to another position or to edit its properties, one first has to return to the *Select tool*.

- The *Edge tool* is used to add new edges between locations. Start the edge by clicking on the source location, then click in order to place nails and finally click the target location. The operation can be cancelled by pressing the middle or right mouse button. It is possible to change the source and target of an edge by moving the mouse to the beginning or end of an edge until a small circle appears (the "nail"). Drag this circle to a new location in order to change the source or target of the edge.
- The *Nail tool* is used to add new nails to an edge, that is, places where an edge may change direction. Simply click and drag anywhere on an edge to add and place a new nail.

We will now construct our first Uppaal model. In the field *Name* at the top of the drawing window we enter the name of the first component ("template") of our

model: `Jobber`. Next we right click (with the *Select tool*) on the location which Uppaal has already placed in the drawing window. We can then give this location a name, for instance `begin`. Each automaton has at most one *initial location*, marked by a double circle. During simulation or in verifications the automaton will always start in this location. By checking the box *Initial* in the menu for a location, we specify that this location is the initial location of the automaton. The menu contains a couple of other fields and options (*Invariant*, *Urgent* and *Committed*), but for the moment we will not use these.

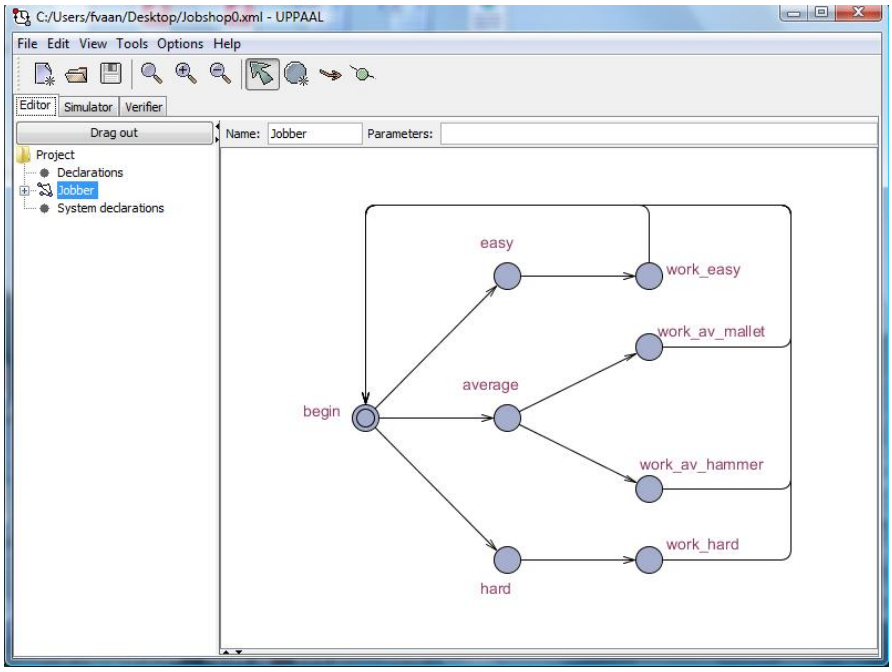


Fig. 1.6 A first version of the model.

We can continue drawing and construct the automaton that is depicted in Figure 1.6. The first transition from the initial location corresponds to the moment when a jobber picks a new job from the conveyor belt. There are three transitions possible since according to the informal specification there are three types of jobs: easy jobs, hard jobs and jobs with average difficulty (neither easy nor hard). The next transition corresponds to the moment at which the jobber grabs a tool (if needed) and starts working on the job. In the case of a job with average complexity, there are two possible transitions, depending on the tool that is selected. The third transition corresponds to the moment that the jobs is done: the automaton returns to its initial state and the jobber is ready for the next job.

At this point, we have not specified in our model how the choice between the transitions from location `begin` to location `easy`, `average` or `hard` is made. Also, we have not specified how the choice between the transitions from location `average` to location `work_av_mallet` and `work_av_hammer` is made. Choices for which the model does not specify how they are resolved are called *nondeterministic*. Nondeterminism is very useful for maintaining a high level of abstraction in descriptions of the behavior of physical systems and machines [6].

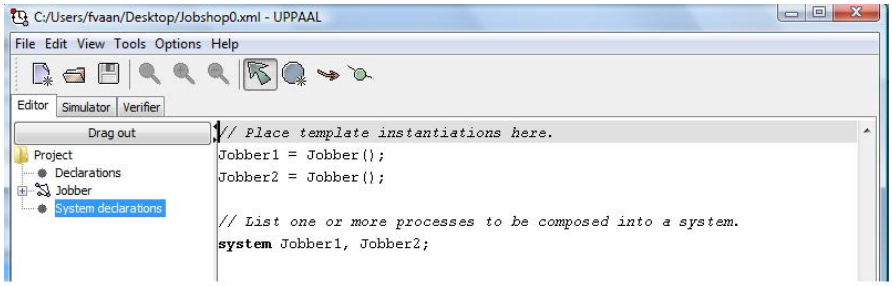


Fig. 1.7 Declaration of system components.

Once we have constructed the automaton of Figure 1.6, our first model is almost ready. Click on *System declarations* in the left window. We now see a screen in which one can list all the processes (automata) in a model. As shown in Figure 1.7, we specify that our first model consists of two instances of the template `Jobber`, called `Jobber1` and `Jobber2`. So our model consists of two automata that, intuitively, run in parallel. As we will see, a global state of the full model is fully determined by the locations (states) of its components, and the transitions of the full model are in direct correspondence with the edges (transitions) of its components. By clicking on *Tools* in the menu bar at the top of the screen, and then on *Check Syntax*, we may check whether a model is syntactically correct. If a model contains mistakes, these will be underlined in red. A more detailed description of the errors is provided in a window at the bottom of the screen (normally not visible, but one can make it larger using the mouse).

1.4 The simulator

Once a model is syntactically correct, we can simulate it, that is, explore the state space of the model in a step-by-step fashion, by selecting the tab *Simulator*. The resulting screen is displayed in Figure 1.8. Uppaal makes two copies of the template

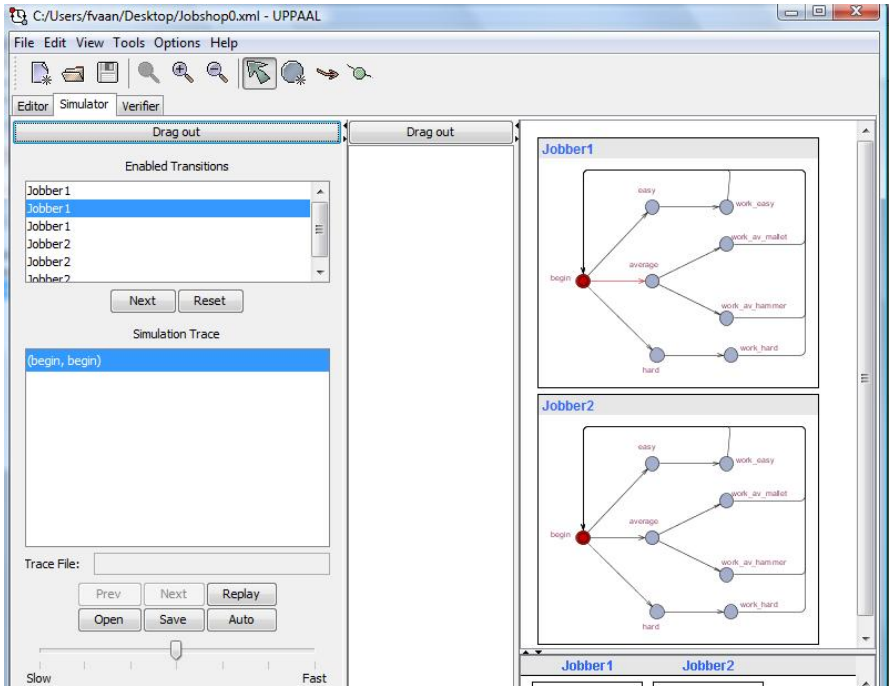


Fig. 1.8 Screenshot of the simulator.

`Jobber`, one for each automaton instance. Using red dots the current location of each automaton is highlighted. Initially, the current location of an automaton is its initial location. In the simulation control panel on the left, we see that (due to nondeterminism) six transitions are possible from the initial state (three for each jobber). When we select one of these transitions, the corresponding step in the automaton diagram for `Jobber1` or `Jobber2` is colored red. Pressing the *Next* button causes the simulated system to take the selected transition, and to update the current location. Using the *Prev* button one can go to the previous step in a simulation trace, with the *Reset* button one can bring the system back to its initial state, and with the *Replay* button one can instruct Uppaal to automatically replay the current trace. If we press the button *Auto*, then Uppaal starts executing randomly selected transitions, one after the other. The speed of the simulation can be changed by moving the arrow in between *Slow* and *Fast*. We can stop the random simulation by pressing the *Auto* button again.

1.5 Channels

The simulator is very useful for obtaining insight in the behavior of a model and finding mistakes. By playing with our first model in the simulator, we quickly discover that something is wrong: both jobbers can be in location `work_hard` simultaneously. This should not be possible, since in this location both jobbers are using the hammer, and there is only one hammer. Hence we need to refine/correct our model.

In order to fix the model, we introduce separate templates for both the hammer and the mallet. For each tool there are 2 locations: `free` or `taken`. The automaton for a tool moves from location `free` to location `taken` when it is grabbed by one of the jobbers. In order to model the synchronization between tools and jobbers, we use the notion of *(synchronization) channels* from Uppaal. Once "`a`" has been declared as a channel, transitions can be labeled with either `a!` or `a?`. This can be done by double clicking (within the *Editor*) transitions with the *Select tool*, and then writing `a!` or `a?` in the *Sync* field. When two automata synchronize on channel "`a`", this means that an `a!` transition of one automaton occurs simultaneously with an `a?` transition of another automaton. An `a!` or `a?` transition can never occur on its own: `a!` always has to synchronize with `a?`, and vice versa. If there is one automaton `S` that can do an `a!` but two automata `R1` and `R2` that can do an `a?`, then there is a nondeterministic choice and `S` can synchronize with either `R1` or `R2`. The other automaton has do something else or has to wait until the next `a!` synchronization will be offered. In our jobshop example it does not matter which transition is labeled with `a!` and which transition is labeled with `a?`. Often, we place the `a!` on a transition of the component that takes the "initiative" for the synchronization. In the case of the jobshop, the jobbers take the initiative to grab a tool, whereas the tools are passive and just "wait" until someone is using them. Hence, we place the !'s in the template for the jobbers, and the ?'s in the templates for the tools. Figure 1.9 shows the adjusted model of the jobber, in which synchronization channels have been added. Synchronization channels must also be declared. This can be done by clicking on the (global) project *Declarations* in the window on the left, and inserting the following text:

```
// Place global declarations here.
chan get_hammer, put_hammer, get_mallet, put_mallet;
```

If we simulate the extended model, we quickly see that in the new model "deadlocks" are possible, states from which no transition is enabled. This occurs, for instance, when both jobbers are in location `hard` and both want to perform a `get_hammer!` transition. But since there is no automaton that can perform a matching `get_hammer?`, the system comes to a crunching halt. In order to rule out these deadlocks, we add new templates `Hammer` and `Mallet` (this can be done by selecting the option *Insert Template* in the *Edit* menu). Figure 1.10 shows the def-

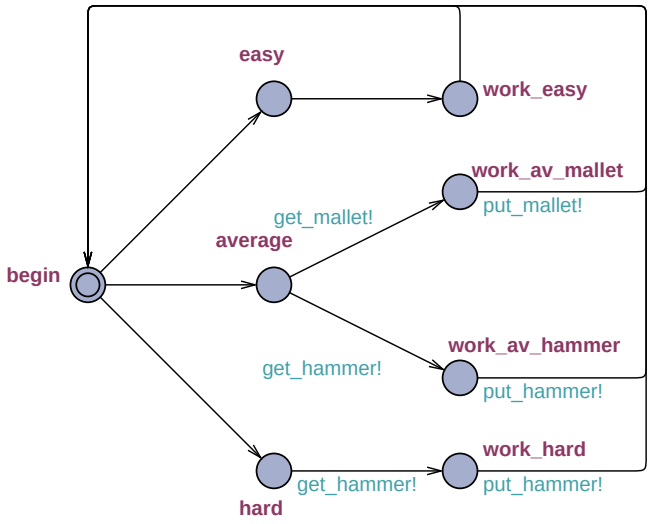


Fig. 1.9 Model of jobber, extended with synchronization labels.

initions of these templates.[1] We add the new automata to the *System declarations* via the text

```
system Jobber1, Jobber2, Hammer, Mallet;
```

We have now completed our first Uppaal model! We can open the model in the simulator and convince ourselves that it indeed behaves as specified in the informal description. Once we are satisfied with the model, we can save it by selecting in the *File* menu the option *Save System As...*. Uppaal models are stored as `.xml` files.

Fig. 1.10 Models of hammer and mallet.

[1] Actually, it would be better to have just one template `Tool`, with two instances `Hammer` and `Mallet`. It is possible to define this in Uppaal, but this involves adding channel names as parameters to a template. Since we prefer to explain the notion of template parameters in Section 1.9 of this chapter, we introduce separate templates for hammer and mallet.

1.6 The verifier

The *Simulator* is extremely useful for playing with a model and obtaining insight, but it does not answer questions like "Is it possible to reach a state in which (some given) property *Bad* holds"? Even if we have not seen a *Bad* state after hours of simulation, this does not guarantee that no such state exists! Fortunately, Uppaal's *Verifier* allows us to rigorously answer this type of questions.

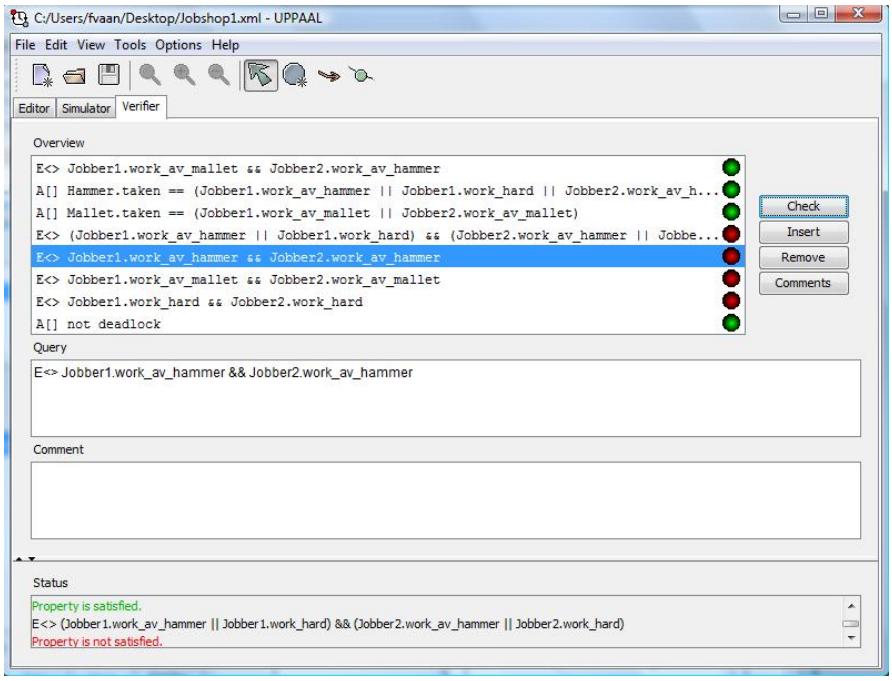


Fig. 1.11 Screenshot of the *Verifier*.

1.6.1 Queries

Within the *Verifier*, we can specify a so-called *Query*, a property that may or may not hold for a given model. By exhaustive exploration of the set of reachable states (the "state space"), the *Verifier* can establish whether a *Query* is satisfied or not. Figure 1.11 shows a screenshot of the *Verifier*. Queries often start with the symbols "`A[]`". This notation, which is taken from the field of temporal logic, means "In all reachable states it is the case that". Thus, for example, the query

```
A[] not deadlock
```

states that in all reachable states of the system there is no "deadlock". Recall that a state has a "deadlock" if it has no outgoing transitions. Stated differently, the above query asserts that in all reachable states at least one transition is possible. If we type the query in the *Query* window of the *Verifier* and then press the *Check* button, Uppaal explores all the reachable states to see if they have an outgoing transition. In the case of our model this is indeed the case, and therefore Uppaal returns *Property is satisfied*. This means that in each of the reachable states always either `Jobber1` or `Jobber2` can proceed.

A new query can be entered by pressing the *Insert* button, and typing the query in the window *Query*. We may for instance enter the following text:

```
E<> (Jobber1.work_hard && Jobber2.work_hard)
```

Here the notation "`E<>`", again taken from temporal logic, means "There exists a reachable state such that". The above query asserts that there exists a reachable state in which that `Jobber1` and `Jobber2` are in location `work_hard`, that is, both jobbers are working on a hard job. Much of the syntax of Uppaal is similar to that of common programming languages such as C, C++, Perl and Java. For instance, `&&` denotes logical "and": it combines two boolean values and returns true if and only if both of its operands are true. When we ask Uppaal to check the above query, the result is *Property not satisfied*. This is what we expect: if a jobber is working on a hard job he is using the hammer, and since there is only one hammer, at most one jobber can work on a hard job at a time. Note that Uppaal does not use any clever form of reasoning to arrive at this conclusion. The tool just uses brute force to explore all the reachable global states of the model and to check for each of these states whether both jobbers are working on a hard job.

In the Uppaal help menu the full syntax for queries and expressions is described. In this chapter, we only consider queries of the form `A[]e` and `E<>e`, where `e` is an expression. An expression `e` consists of a boolean combination of atomic propositions. Atomic propositions can be of the form `A.l`, for `A` an automaton and `l` a location. Such a proposition is true in a global state of the model if in this state automaton `A` is in location `l`. Table 1.1 gives some examples of boolean operators that can be used in Uppaal. Further on in this chapter we will encounter other types of properties.

Symbol	Operator name	Meaning
`&&`	and	`e && f` is true if both `e` and `f` evaluate to true
`||`	or	`e || f` is true if `e` evaluates to true or `f` evaluates to true
`==`	equality	`e == f` is true if `e` and `f` evaluate to same value
`imply`	implication	`e imply f` is true if `e` evaluates to false or `f` evaluates to true
`not`	negation	`not e` is true if `e` evaluates to false

Table 1.1 Some logical operators in Uppaal

1.6.2 Diagnostic traces

We can ask Uppaal whether there exists a reachable state in which one jobber is working on an average job with a mallet, and the other jobber is working on a average job with the hammer:

```
E<> (Jobber1.work_av_mallet && Jobber2.work_av_hammer)
```

Uppaal then answers *Property is satisfied*. In this case, Uppaal can also provide a concrete example that illustrates why the property holds, that is, a trace leading to a state in which the first jobber is working on an average job using the mallet, and the second jobber is working on an average job using the hammer. In order to let Uppaal compute such an example, we choose under *Options* the entry *Diagnostic Trace* and then select the option *Shortest*. If we now let Uppaal check the above property again, it will again produce the answer *Property is satisfied* but in addition it will compute the shortest path (or trace) leading to a state in which both `Jobber1.work_av_mallet` and `Jobber2.work_av_hammer` hold. The tool asks whether it may store this path in the simulator. If we give Uppaal permission to do this, we can replay the trace in the simulator. If we open the simulator, then we see the final state of the trace in which `Jobber1` is in location `work_av_mallet` and `Jobber2` is in location `work_av_hammer`. By pressing the *Reset* button we go to the initial state of the trace, and by pressing the *Replay* button we tell Uppaal to replay the trace within the simulator. We can also replay the trace step-by-step by repeatedly pressing the lowest *Next* button.

In general, Uppaal can provide a diagnostic trace for `E<>` properties that hold, and for `A[]` properties that do not hold. In the case of `E<>` properties that do not hold, or `A[]` properties that hold, Uppaal can only report that it exhaustively checked all the reachable states of the model and didn't find anything. In the above example of an `E<>` property, the diagnostic trace is rather trivial and consists of only four transitions. However, in realistic models of industrial applications, diagnostic traces may describe tricky scenarios involving thousands of transitions. In such cases, Uppaal's ability to provide diagnostic traces is extremely useful. For instance, if Uppaal tells us that a certain correctness property does not hold, an engineer typically wants to know why this is the case.

1.6.3 Saving queries and traces

We can save queries by selecting in the *File* menu the option *Save Queries As...*. The queries are then saved as a text file with extension `.q`. If one opens a Uppaal model `model.xml`, then automatically also the query file `model.q` is opened (if it exists). Alternatively, one can open a query file by selecting in the *File* menu the option *Open Queries...*. It is also possible to save traces by pressing the *Save* button in the simulator. Traces are saved as (unreadable) text files with extension `.xtr`. A trace file can be opened by pressing the *Open* button in the simulator.

1.6.4 How many states are there?

Let us try to compute the total number of reachable states of our jobshop model. Since each jobber is modeled by an automaton with 8 locations, and each tool is modeled by an automaton with 2 locations, there are at most $8 \times 8 \times 2 \times 2 = 256$ global states. However, many of these states can not be reached from the initial state: they will never occur in any run of the system. In order to see this, observe that once we know the state of the two jobbers, we also know the state of the two tools:

```
A[] Mallet.taken == (Jobber1.work_av_mallet
                  || Jobber2.work_av_mallet)

A[] Hammer.taken == (Jobber1.work_av_hammer
                  || Jobber1.work_hard
                  || Jobber2.work_av_hammer
                  || Jobber2.work_hard)
```

The first query states that the mallet is taken exactly when either the first or the second jobber uses it for an average job. The second query states that the hammer is taken exactly when either the first or the second jobber is using it for either an average or a hard job. Since the locations of both mallet and hammer are fully determined by the locations of the jobbers, this means that our model has at most $8 \times 8 = 64$ global states that are reachable from the initial state.

Five of these remaining 64 states can not be reached since the mallet and hammer can only be used by one jobber at a time:

```
A[]
not (Jobber1.work_av_mallet && Jobber2.work_av_mallet)
 &&
not (Jobber1.work_av_hammer && Jobber2.work_av_hammer)
 &&
not (Jobber1.work_av_hammer && Jobber2.work_hard)
 &&
not (Jobber1.work_hard      && Jobber2.work_av_hammer)
 &&
not (Jobber1.work_hard      && Jobber2.work_hard)
```

All the other combinations of locations of `Jobber1` and `Jobber2` can be reached, and so the total number of reachable states of our model is $64 - 5 = 59$.[2] For Uppaal our model is really small: the program can easily handle models with thousands or even millions of states. It is trivial to construct models that are so big that Uppaal cannot handle them and, for instance, runs out of memory. For instance, if we modify our jobshop model and increase the number of jobbers to 100, then the size of the state space becomes in the order of 8^{100} and too big for Uppaal.

[2] Unfortunately, the regular version of Uppaal has no option to count the number of reachable states in a model. Such an option is present however in the command line version of the verifier: `verifyta -u`.

1.7 Variables

We will now describe how one can add integer variables to Uppaal models. The values of these variables can be tested and updated in transitions, and thus influence the behavior. State variables are indispensable for modeling nontrivial systems and give the Uppaal modeling language an expressive power that is comparable to simple programming languages. To illustrate the use of state variables, we discuss a small modification of the jobshop model:

> We suppose that the jobbers stop with their work as soon as they have completed 10 jobs together.

In order to capture this additional requirement in our model, we add an integer state variable that records how many jobs have been taken from the belt. This is achieved by adding the following lines in the global project *Declarations* in the window on the left in the *Editor*:

```
const int J = 10;
int[0,J] jobs;
```

In the first line, we declare an integer constant with value 10. As suggested by their name, the value of constants always remains the same. In the second line an integer variable `jobs` is declared with minimum value 0 and maximum value J. The value of variables can be modified when transitions occur. By default, the initial value of a variable is 0. The domain of integer variables in Uppaal is always bounded. If we specify no bounds and simply declare

```
int jobs;
```

then implicitly the minimum value is -32768 and the maximum value is 32768.[3] Whenever during exploration of the state space — either in the simulator or in the verifier — a variable gets assigned a value outside its domain, this is referred to as a "run time error" and an error message is generated.

Figure 1.12 shows an extension of the jobber model in which the variable `jobs` is used. In the location `begin`, a jobber only accepts a new job when `jobs` $<$ `J`. In this case, the value of `jobs` is incremented by 1. We can implement this change of the model by double clicking the transition from `begin` to `easy`, and then write `jobs` $<$ `J` in the field *Guard* of the resulting menu. In Uppaal, a transition can only be taken if its guard evaluates to true (if no guard is specified then we assume it equals `true`). In the field *Update* of the transition, we specify that the value of `jobs` must be incremented by 1 whenever the transition is taken. This is done by writing `jobs++`, using syntax that has been taken from the programming language C. Alternatively, we can also write `jobs` $=$ `jobs` $+$ `1` or `jobs` $:=$ `jobs` $+$ `1` (this all means the same). We also add an extra location `done` to the model, and a transition

[3] Like in C, Uppaal uses 16-bit `int`'s, including 1 bit to represent the sign.

from `begin` to `done`, which is taken whenever `jobs` has reached the value `J` and all jobs are done.

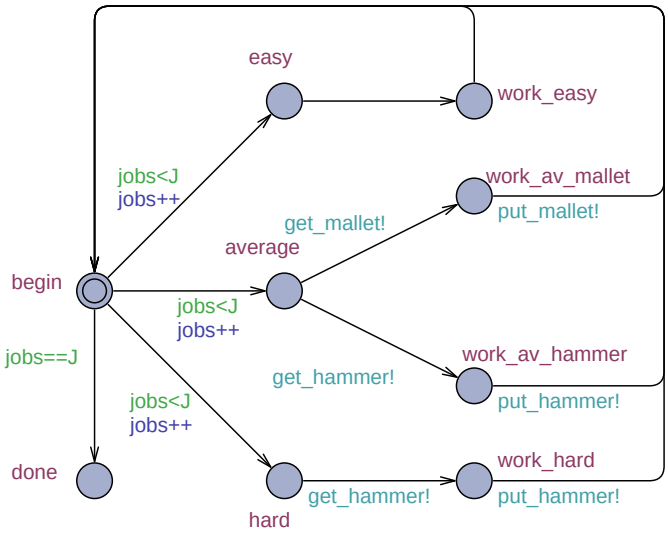


Fig. 1.12 Model of jobber in which exactly `J` jobs are carried out.

When we now start the simulator, we see a separate window in which, for each state, the value of the state variable `jobs` is listed. Using the verifier, we can establish that the model of Figure 1.12 satisfies exactly the same correctness properties as the model of Figure 1.9, except for the property

```
A[] not deadlock
```

which is no longer satisfied. When the jobbers have finished `J` jobs and have moved to location `done`, no further transitions are possible and therefore a deadlock state has been reached.

Observe that in the modified model, both jobbers *together* perform `J` jobs. This is because `jobs` is a *global* variable that can be tested and updated by both jobbers. In Uppaal, we can also declare *local* variables, which can only be used by one automaton. When in the *Editor* we click on the "+" symbol at the left of the template `Jobber`, a new line *Declarations* appears below `Jobber`. When we select this line, we can declare local variables for the template. For instance, by moving the line

```
int[0,J] jobs;
```

from the global declarations section to the local declarations section of template `Jobber`, we give `Jobber1` and `Jobber2` each their own, local copy of variable `jobs`. The result is that both `Jobber1` and `Jobber2` have to perform `J` jobs, instead of `J` jobs together.

Uppaal has a rather extensive syntax for the expressions in guards and updates. It is possible to declare arrays and Boolean variables, and a user can even define new "record" types and new functions. The syntax for doing this is very similar to the syntax of programming languages such as C and Java. For an overview of the syntax we refer to Chapter 2 of this handbook **Add ref** and to the help menu of Uppaal: click on *Language Reference* and then on *Expressions*.

1.8 Time and clocks

In the design and analysis of real systems, timing aspects often play a role. Sometimes we may abstract from quantitative timing, and only consider the ordering of events, but often timing information has to be included in the model in order to be able to answer certain questions. For instance, we may need to establish not only that a certain location can be reached, but also how fast. In the case of our jobshop, the following question could arise. Uppaal has been more or less designed to answer this type of questions.

> We suppose that a jobber needs (at least) 5 seconds for an easy job, 10 seconds for an average job using the hammer, 15 seconds for an average job using the mallet, and 20 seconds for a hard job. We suppose that the jobs arrive in the following order: `H`, `A`, `H`, `H`, `H`, `E`, `E`, `A`, `A`, `A`, where `E` denotes an easy job, `A` an average job, and `H` a hard job. How much time do the two jobbers need (at least) to complete the 10 jobs?

1.8.1 Modeling the conveyor belt

Before we add timing to our model, we first add an automaton that describes the behavior of the conveyor belt. The belt was not included in our first model (there was no reason for adding it) but in order to answer the question about timing, the order in which jobs arrive appears to be relevant. In order to model incoming jobs, we declare three new channels `jobE`, `jobA` en `jobH`, which correspond to the arrival of easy, average, and hard jobs, respectively. The automaton `Belt`, which is depicted in Figure 1.13, describes the behavior of the conveyor belt that delivers the 10 jobs in the specified order. By adding automaton `Belt` to the model in *System declarations* and by labeling the outgoing transitions of location `begin` of the jobber automaton of Figure 1.9 in the obvious way with the synchronization channels `jobE`?, `jobA`? en `jobH`?, we model that the jobbers have to deal with the specified sequence of 10 jobs.

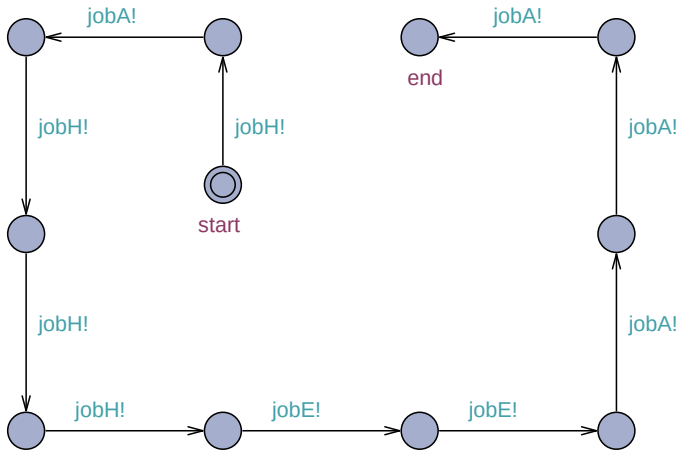


Fig. 1.13 Model of conveyor belt that delivers 10 jobs.

1.8.2 Clocks and lower bounds on timing

In the models that we have constructed thus far, time is not modeled explicitly. In Uppaal we assume that transitions occur instantaneously and do not take time. Time may only elapse when all automata in the model are waiting in a location. Whenever we want to model an activity that takes time, we can do this by introducing two consecutive transitions, one corresponding to the start of the activity, and another corresponding to the end of it. Often, models only contain transitions that correspond to either the beginning or the end of an activity that has a duration. If we want to be really precise then, for instance, we could say that the `jobH!` transition corresponds to the moment when a jobber starts to pick a hard job from the conveyor belt, the `get_mallet` transition corresponds to the moment when a jobber starts grabbing the mallet and the `put_mallet` transition corresponds to the moment when a jobber ends the activity of putting the mallet back on the table again.

If we want to specify lower and upper bounds on the time that an automaton may stay in a certain location, we can do this in Uppaal using so-called *clocks*. A clock is a special type of variable, whose domain consists of the set of nonnegative real numbers. Just like other variables, clocks can be declared either as a global variable (which can be tested and updated by all automata) or as a local variable (which can only be used by one automaton). In the initial state, all clocks have value 0. When an automaton is waiting in a location and time elapses then the values of its clocks increase. More precisely, when t time units pass then the values of all clocks in the model increase with t. Thus, all clocks are "perfect" and increase at exactly the same rate as real-time. In reality, of course, no clock is 100% perfect, but in our modeling language it is convenient to use the idealization of perfect clocks to specify upper and lower bounds on the timing of transitions.

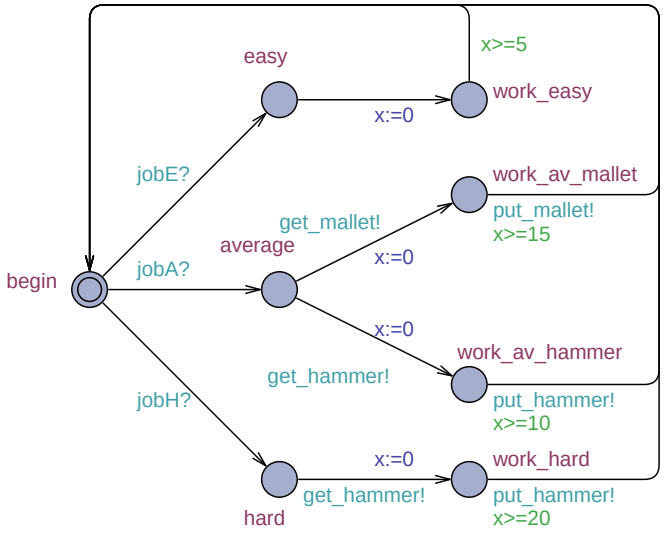


Fig. 1.14 Model of jobber extended with a clock.

Figure 1.14 shows a model of the jobber that has been extended with the new synchronization channels `jobE`?, `jobA`? and `jobH`?, and also with a clock variable `x`. Clock `x` has been declared by adding the following line to the local *Declarations* section of template `Jobber`:

```
clock x;
```

When a jobber moves from location `easy` to location `work_easy`, that is, starts to work on an easy job, clock `x` is reset to 0 via an update `x := 0`. Subsequently, the guard of the outgoing transition of `work_easy` tests whether `x >= 5`. In this way, we enforce that this transition may only occur once the automaton has been in location `work_easy` for at least 5 time units. This corresponds to our assumption that a jobber needs at least 5 time units to complete a simple job. In a similar way we model that the automaton spends at least 10, 15 and 20 time units in locations `work_av_hammer`, `work_av_mallet` and `work_hard`, respectively.

In the resulting model, `Jobber1` and `Jobber2` both have a local clock variable `x` that records how long they have been working on a certain job. Each time when a jobber starts with a new job its local clock is reset to 0. In order to record the total amount of time that has elapsed, we also introduce a global clock `now`, which is never reset. As a result, the global declarations look as follows:

```
chan jobE, jobA, jobH,
     get_mallet, get_hammer, put_mallet, put_hammer;
clock now;
```

We can ask Uppaal whether there exists a state in which all the 10 jobs have been delivered and both jobbers have returned to their initial state (that is, all the jobs have been completed):

```
E<> (Belt.end && Jobber1.begin && Jobber2.begin)
```

When Uppaal computes a diagnostic trace which demonstrates that this property is satisfied, this will, in general, not be the shortest trace. After all, we have only specified lower bounds for the timing in our model and no upper bounds. Thus the belt may wait indefinitely before delivering a job, a jobber may dawdle for days before he starts with the job, and for years before completing it. However, if we select in the menu *Options* under *Diagnostic Trace* the option *Fastest*, then Uppaal will produce the fastest execution that leads to the specified state. In the simulator we can see that in the final state of this execution `now` $>=$ `100`. This means that the jobbers need at least 100 time units to complete the 10 jobs. The easiest way to understand the schedule that has been computed by Uppaal is via the so-called "message sequence chart" in the lower right window of the simulator: here we see which jobber handles which job. Figure 1.15 visualizes this fastest schedule in an

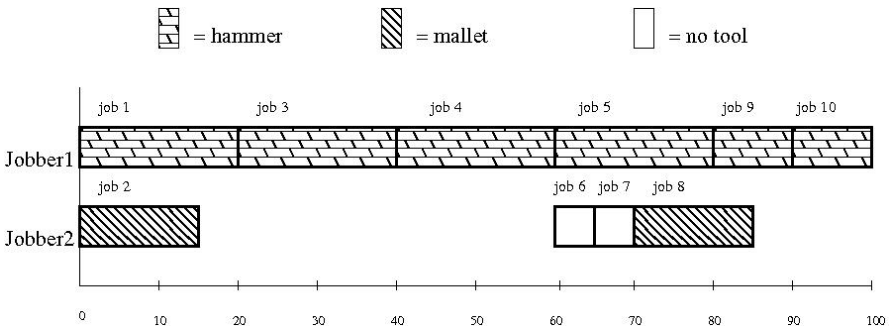


Fig. 1.15 Fastest schedule for completing the 10 jobs.

even more compact way as a "Gantt chart".[4] We see that one jobber is permanently busy with the hammer, whereas the other jobber has a relaxed schedule in which he is idling most of the time but also completes some jobs with his hands or with the mallet. It is easy to come up with an equally fast schedule in which the work load is more evenly distributed. We may decide, for instance, to give the third job to `Jobber2`.

[4] Figure 1.15 was constructed manually. Effort is underway to extend Uppaal with a feature for automatic visualization of traces using Gantt charts.

1.8.3 Upper bounds on timing

We have seen that *lower bounds* on timing can be specified with the help of clock constraints in guards. For example, the constraint `x >= 5` in the guard of the transition from `work_easy` to `begin` indicates that this transition can only be taken once the jobber has spent at least 5 time units in location `work_easy`. In practice, we often need to prove upper bounds on timing: an airbag needs to inflate within a few microseconds after a collision has been detected, a soccer playing robot must quickly react when the ball is nearby, etc. In our jobshop example, the following question may arise:

> We suppose that a jobber needs at most 7 seconds for an easy job, 12 seconds for an average job with the hammer, 17 seconds for an average job with the mallet, and 22 seconds for a hard job.
>
> How much time do the two jobbers need at most to complete the 10 jobs, assuming that a jobber picks a new job from the belt as soon as he is ready to work on it, and that he grabs a tool that allows him to do a job as soon as it becomes available?

In Uppaal we can specify *upper bounds* on timing, using so-called "invariants". When we double click the location `work_easy`, a window appears with a field *Invariant*. By entering `x <= 7` in this field, we specify that in this location the value of `x` will always be at most 7. In other words, a jobber needs at most 7 time units to complete a simple job. If more than 7 time units have elapsed then the jobber has left location `work_easy`. In Figure 1.16 upper bounds of 7, 12, 17 and 22 have been added for locations `work_easy`, `work_av_hammer`, `work_av_mallet` and `work_hard`, respectively. In addition, an upper bound of 0 has been added for location `easy`. This models the assumption that when a jobber has picked an easy job from the belt he starts working on it right away. We could have enforced this upper bound by resetting clock `x` upon entering location `easy`, in combination with an invariant `x <= 0` for this location. But Uppaal supports a simpler way of specifying the same requirement: if we double clock the location `easy` we can check the field *Urgent*. If a location is *Urgent* then this means that time can not elapse within this location, and hence a transition to another location will occur immediately.

After all these modifications of the model, it is still not possible to infer an upper bound on the time need to complete all jobs. The reason is that workers may wait indefinitely before picking the next job from the belt, and they may wait indefinitely before grabbing a tool. We have not yet modelled the requirement that jobbers grab jobs and tools as soon as they can. A convenient way to eliminate idling is by making the synchronizations for grabbing a job or tool "urgent". When a synchronization channel is urgent, this means that whenever a synchronization with this channel is enabled, time can not advance and a transition has to be taken immediately. We can specify this in Uppaal by changing the declarations of the channels as follows:

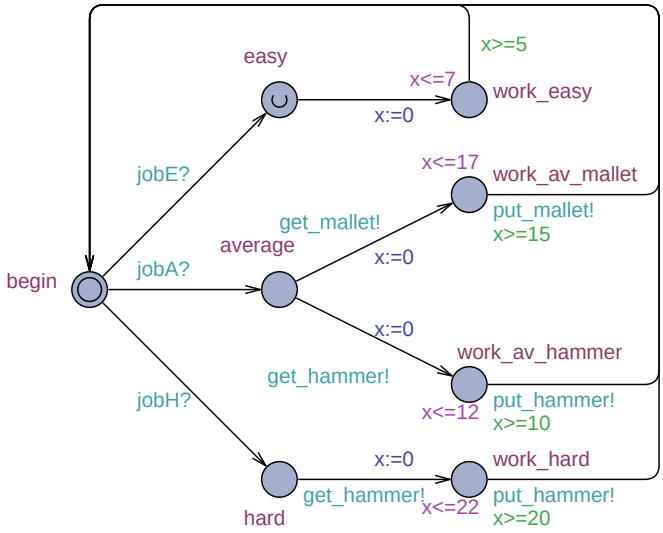


Fig. 1.16 Model of jobber with upper bounds on timing.

```
urgent chan jobE, jobA, jobH, get_mallet, get_hammer;
chan   put_mallet, put_hammer;
```

Note that there exists a subtle difference between the use of urgent locations and urgent synchronizations. If we would make location `begin` of the jobber template urgent rather than channels `jobE`, `jobA` and `jobH`, a problem would arise in a state where a jobber is in location `begin` but no further jobs are on the belt and hence no `jobE`, `jobA` or `jobH` synchronizations are offered: in such a state there would be no possibility for time to progress and there would be a "time deadlock". Clearly, such a model is not realistic. Likewise, a time deadlock would arise if we would make location `hard` urgent rather than channel `get_hammer`.

Observe that in the model of Figure 1.16, a jobber may spend in between 5 and 7 time units in location `work_easy`. We have not specified in our model any further information about the time needed to complete an easy job: this choice is fully nondeterministic. Maybe a jobber usually completes an easy job within 6 time units but sometimes needs more time when he is tired. Maybe one jobber always complete a job within 5.731 time units, whereas another jobber always needs at least 6.194 time units. Maybe jobbers will always complete easy jobs within 5.5 and 5.8 time units, but we have not carried out the exact measurements and want to be on the safe side. The ability to have nondeterminism in the timing of transitions is an extremely useful feature of timed automata, which makes it possible to describe systems and reason about them at a high level of abstraction.

Whereas the *Verifier* has an option to compute the fastest execution leading to a certain state, there is no corresponding option to compute the slowest execution. Nevertheless, we can compute this slowest execution via a few successive approx-

imations. Uppaal can conform our guess that after 200 time units all jobs will be completed, that is, that the following property is satisfied:

```
A[] now>=200 imply
          (Belt.end && Jobber1.begin && Jobber2.begin)
```

We even have

```
A[] now>=150 imply
          (Belt.end && Jobber1.begin && Jobber2.begin)
```

but not

```
A[] now>=110 imply
          (Belt.end && Jobber1.begin && Jobber2.begin)
```

So the jobbers need at most between 110 and 150 time units to complete all the jobs. By doing a "binary search" and reducing the size of the interval step by step, we may infer that the following property is satisfied:

```
A[] now>=127 imply
          (Belt.end && Jobber1.begin && Jobber2.begin)
```

but the next property is not:

```
A[] now>=126 imply
          (Belt.end && Jobber1.begin && Jobber2.begin)
```

Hence, the diagnostic trace for the last property gives the slowest possible schedule, which takes exactly 126 time units. In this schedule, the final transitions from the jobbers back to the begin state are missing, but these transitions take no time. So even when the jobbers always start working on jobs as soon as they can, and immediately grab tools whenever they become available, the worst case schedule is still 26 time units slower than the fastest schedule from Figure 1.15. Figure 1.17 visualizes the slowest schedule as a Gannt chart. At time 0, `Jobber1` grabs the first

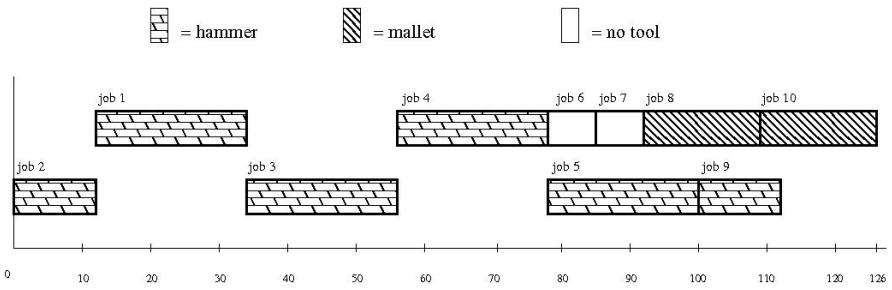


Fig. 1.17 Slowest schedule for completing the 10 jobs.

(hard) job, and `Jobber2` grabs the second (average) job. Of course, in any reasonable scenario, `Jobber1` would use the hammer and `Jobber2` the mallet. But

in the worst case scenario of Figure 1.17, `Jobber2` gets the hammer, and hence `Jobber1` has to wait. Until time 78 one jobber is using the hammer while the other jobber is idling. At the end of the schedule, after `Jobber2` has finished job 9, `Jobber2` is still working for some time on job 10 using the mallet.

1.9 Parameters and arrays

In this section, we will discuss two useful features of Uppaal, that allow us to describe our jobshop model more compactly: template parameters and arrays.

1.9.1 Parameters

In the jobshop model, we have defined a single template `Jobber` with two instances `Jobber1` and `Jobber2`. Likewise, we would like to have a single template `Tool` with two instances `Hammer` and `Mallet`. It is possible to define this in Uppaal using the notion of *Parameters*. Parameters can be declared to have either call-by-value or call-by-reference semantics, that is, a template may have access to either a local copy of the argument or to the original. The syntax is taken from C++, where the identifier of a call-by-reference parameter is prefixed with an ampersend in the parameter declaration. Clocks and channels must always be call-by-reference parameters.

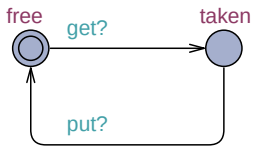


Fig. 1.18 Generic model of template `Tool`.

Figure 1.18 shows the definition of the generic template `Tool`. In the field *Parameters* at the top of the *Editor* tab, we declare the two channel names that are used as parameters in this template:

```
urgent chan &get, chan &put
```

In the *System declarations* section, we can now define `Hammer` and `Mallet` as instances of `Tool`:

```
Hammer = Tool(get_hammer,put_hammer);
Mallet = Tool(get_mallet,put_mallet);
```

The old templates `Mallet` and `Hammer` can be deleted via the *Remove template* option in the *Edit* menu. Our new model has exactly the same behavior (in terms of global states and transitions) as the old model, but due to the use of parameters the definition has become shorter.

1.9.2 Arrays

The model of the belt of Figure 1.13 is not easy to extend or reuse: each time we want to study a new arrival pattern of jobs, we have to redraw the automaton. This is cumbersome, especially if we want to schedule a batch with a large number (say hundreds) of jobs. It is also more natural to define the arrival pattern as a *data structure* rather than as a *control structure* (automaton). We can describe the job arrival pattern as a (constant) integer array as follows:

```
const int E=0;
const int A=1;
const int H=2;

const int J=10;
const int[0,2] jobs[J] = {H,A,H,H,H,E,E,A,A,A};
```

Uppaal does not support enumerated types and therefore we use 0 to encode easy jobs (`E`), 1 to encode average jobs (`A`), and 2 to encode hard jobs (`H`). As before, the integer constant `J` denotes the total number of jobs in the batch. The one dimensional constant array `jobs` specifies the arrival pattern of jobs (which is identical to the pattern in Figure 1.13). The size of the array equals `J` (counting from 0 to `J` $-$ 1) and the range is the set $\{0,1,2\}$ or equivalently $\{\texttt{E},\texttt{A},\texttt{H}\}$. Figure 1.19 shows a generic model for the `Belt` template that uses the information from the array `jobs` to generate the appropriate sequence of `jobE!`, `jobA!` and `jobH!` transitions. The automaton uses an auxiliary variable

```
int[0,J] i;
```

that records the number of jobs that has been delivered thus far. Again, the new model has exactly the same behavior (in terms of global states and transitions) as the old model, but due to the use of arrays the definition has become shorter and easier to reuse.

1.10 What is a good model?

After having presented the basic functionality of the Uppaal tool, we want to conclude this chapter with some general recommendations for constructing models. To some extent, building good models is an art. Dijkstra's motto "Beauty is our business" [4] applies to models as well as to programs. Nevertheless, we can state seven

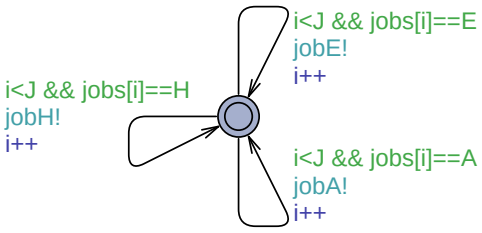


Fig. 1.19 Generic model of template `Belt`.

criteria for good models.[5] These criteria are in some sense obvious, and any person with experience in modelling will often try to adhere to them. Often some criteria are hard to meet and typically several of them are conflicting. In practice, a good model is often one which constitutes the best possible compromise, given the current state-of-the-art of tools for modelling and analysis.

1. A good model has a clearly specified **object of modelling**, that is, it is clear what thing the model describes. The object of modelling can be (a part of) an existing artefact or physical system, but the object may also be a document that informally specifies a system or class of systems (for instance a protocol standard), and it may even be a collection of ideas of a design team about a system they construct, expressed orally and/or by some drawings on a whiteboard.
   In the case of our jobshop example, the object of modelling is the informal specification that is contained in the grey boxes throughout this chapter.
2. A good model has a clearly specified **purpose** and (ideally) contributes to the realization of that purpose. Possible purposes include: communication between stakeholders about a design, a specification of a system, verification of specific properties (safety, liveness, timing,..), analysis and design space exploration, code generation, and test generation. A model can be descriptive or prescriptive. If a model has to serve several distinct purposes then often it is better to construct multiple models rather than one.
   The only purpose of the jobshop models constructed in Sections 1.3 up to 1.7 is to explain the use of the Uppaal tool. The models in Section 1.8 serve the additional purpose that they help us to answer the timing related questions stated in the grey boxes.
3. A good model is **traceable**: each structural element of a model either (1) corresponds to an aspect of the object of modelling, or (2) encodes some implicit domain knowledge, or (3) encodes some explicit additional assumption. Additional assumptions are for instance required when a protocol standard is incomplete (e.g., it does not specify how to handle certain events in certain cases). Links between the structural elements of the model and the aspects of the object of modelling should be clearly documented. A distinction must always be

[5] Most of these criteria are described by Mader, Wupper and Boon [9]. We refer to [9] for further links to related work in the areas of software engineering, requirements analysis, and design.

made between properties of (a component of) a model and assumptions about the behavior of its environment.
Our jobshop models are traceable: in the text we explain for each element in a model how it relates to the informal specification.

4. A good model is **truthful** (or **valid**): relevant properties of the model should also carry over to (hold for) the object of modelling. Typically, for each (relevant) behavior of the object of modelling there should be a corresponding behavior of the model. In the construction of models often idealizations or simplifications are necessary in order to allow for the use of a certain modeling formalism or in order to be able to analyze the model. In these cases, the model may not be entirely truthful. The modeller should always be explicit about such idealizations/simplifications, and have an argument why the properties of the idealized model still say something about the artefact. In the case of quantitative models this argument will typically involve some error margin. In the case of timed automata models it frequently occurs that a model "overapproximates" reality and that, due to nondeterminism, certain behaviors that are possible in the model are not possible for the artefact.
   The untimed model of Section 1.7 is certainly truthful to the informal specification of Milner listed in the grey box in Section 1.2. However, in the timed model of Section 1.8 there is at least one idealization that is not entirely realistic. Our model assumes that it takes no time to grab a job from the conveyor belt. Or more precisely: that no time elapses between starting to pick a job from the belt and starting to pick a tool. Since a job consists of both a peg and a block, it is reasonable to assume that a jobber needs both hands to grab a job from the conveyor belt. Hence it is not possible to grab a job and a tool simultaneously, and if we really want our model to be thruthful, we should add extra transitions that correspond to the end of the activity to grab a job, and we should give lower and upper bounds for the duration of this activity. In the jobshop example the criterion of thruthfulness collides with our next criterion of simplicity: in order to keep our model simple we compromised a bit on thruthfulness.
5. A good model is **simple** (but not too simple). Occam's razor is a principle particularly relevant to modelling: among models with roughly equal predictive power, the simplest one is the most desirable. Hence, the number of states and state variables should be as small as possible, and the level of atomicity of transitions should be as coarse grained as possible (but not coarser), i.e., the number of transitions should be minimal given the intended use of the model. Preferably, things should be written only once, and one should avoid ugly encodings. Preferably, the model uses stable, well-defined and well-understood concepts and semantics.
   The model of Section 1.9 is almost maximally simple. We would have liked to simplify the automaton of Figure 1.19 even further so that it only has one transition instead of three. But this is not possible since Uppaal does not allow data parameters for synchronization channels.
6. A good model is **extensible and reusable**, that is, it has been designed to evolve and be used beyond its original purpose. Typically, if one defines models in a modular and parametric way this allows for dimensioning, future extensions

and modifications, especially if modules have well-defined interfaces. Ideally, a model should not just describe the specific system at hand: by appropriate instantiation and dimensioning it should be possible to model a whole class of similar systems.

Our jobshop model can be extended or reused in several ways: we can easily increase the number of jobbers and tools, modify the sequence of jobs, and modify the timing parameters. What can not be changed so easily are the assumptions on which tools can be used for which jobs. The next chapter presents a more generic version of the jobshop model in which variations of these assumptions can be trivially modified. However, this requires the use of some advanced modelling features, which are explained in the next chapter.

7. A good model has been designed and encoded for **interoperability and sharing** of semantics. Model-driven development of an embedded system typically leads to a plethora of models, all presenting different views on and abstractions of the system. If a model is not somehow linked to other models, its usefulness will be limited. Ideally therefore, the relationships between all models should be properly defined, for instance via formal refinement relations.

   In this chapter, we have not addressed issues of interoperability and sharing. We refer to chapters XX and YY **add refs** of this handbook for a description of various links between Uppaal and other model based development tools. In a more realistic version of the jobshop example, in which for instance the jobbers are replaced by robots, one of the things one could do is to take the schedules computed by Uppaal and translate these to control programs for the robots. Such an approach (using Uppaal) is described for instance in [7]. **(and Chapter XX?)**

Clearly, there are many relationships and dependencies between the criteria. If a model is traceable, that is, links between the structural elements of the model and the aspects of the object of modelling are clearly documented, then chances increase that the model will be truthful. Also, if a model has been set up in a modular way, then one may apply a divide-and-conquer strategy both for establishing truthfulness of the model and for analysis.

biosection??

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, Massachusetts, 2008.
2. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
3. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

4. W.H.J. Feijen, Gasteren A.J.M. van, D. Gries, and J. Misra, editors. *Beauty is our business — A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science. Springer-Verlag, 1990.
5. O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
6. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.
7. T. Hune, K.G. Larsen, and P. Pettersson. Guided synthesis of control programs using Uppaal. *Nord. J. Comput.*, 8(1):43–64, 2001.
8. C.A.J. Hurkens. Spreading gossip efficiently. *Nieuw Archief voor Wiskunde*, 5/1(2):208–210, June 2000.
9. A. Mader, H. Wupper, and M. Boon. The construction of verification models for embedded systems. Technical Report TR-CTIT-07-02, Centre for Telematics and Information Technology, University of Twente, The Netherlands, 2007.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
11. Wikipedia. List of model checking tools, February 2011. `http://en.wikipedia.org/wiki/List_of_model_checking_tools`.

Chapter 1
More Features in UPPAAL

Alexandre David, Kim G. Larsen

Abstract Following the introduction to the model checking tool UPPAAL of the previous chapter, this chapter presents a number of additional modeling and verification features offered by the tool. These features include in particular a C-like imperative language with user-defined types and functions, allowing for readable and compact models with reusable updates of discrete variables. Using an example of a Train Gate, we demonstrate the use(fulness) of these features. Also, the chapter presents the full query language of UPPAAL covering both safety, liveness and time-bounded liveness properties, again illustrated using the Train Gate example. Finally, directions are given on modelling choices and use of verification options that may improve time- and/or space-performance of the UPPAAL verifier

1.1 The Train Gate

In the previous chapter the basic modelling formalism of UPPAAL was presented: automata interacting over channels and extended with (integer) variables and clocks. For the ease of modeling, the full formalism of UPPAAL allows for structured variables (arrays and records) together with a C-like imperative language for their updates. To present and illustrate the use(fulness) of these features we use an example of a Train Gate. This example was originally presented in [27] as a number of trains running on seperate tracks, but – for economical reasons – having to cross a commong bridge. The challenge is to model the timing behaviour of the trains, as well as to design (and verify) a controller that will stop and (re)start trains in an appropriate manner, e.g. to avoid trains colliding on the common bridge.

Alexandre David and Kim G. Larsen
Department of Computer Science, Aalborg University, Selma Lagerlöfsvej 300, 9220 Aalborg, Denmark e-mail: adavid,kgl@cs.aau.dk

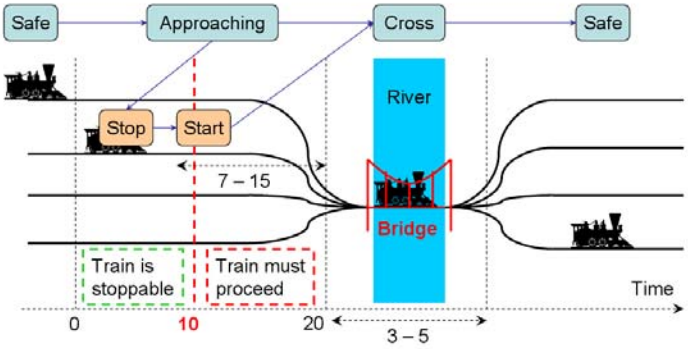


Fig. 1.1 The train gate problem. Trains run on their own tracks except on the bridge. Trains may be stopped before 10 time units, after which they must proceed to the bridge. If stopped a train will take some time to reach the bridge (7–15 time units). Crossing takes some time (3–5 time units).

A Simple Railway Control System [27]: We consider a railway control system to automatically control trains passing a critical point such as a bridge. The idea is to use a computer to guide trains from several tracks crossing a single bridge instead of building many bridges. Obviously, a safety property of such a system is to avoid the situation where more than one train are crossing the bridge at the same time.

Figure 1.1 depicts the problem of trains (here only 4) crossing a bridge. Initially trains are far enough from the bridge and are in a *Safe* state. At some point a train is approaching the bridge (state *Approaching*). The gate controller has then 10 time units to stop it. After this time the train has too much inertia to be stopped safely and must proceed to the bridge. It will take 20 time units to reach the bridge. If the train is stopped (state *Stop*) then it will be restarted again eventually (state *Start*) and it will take between 7 and 15 time units to reach the bridge. A train can be stopped at any time before 10 time units and so we model this non-determinism. When a train crosses the bridge it takes between 3 and 5 time units and we want as a safety property that only one train at a time has access to the bridge. After crossing, a train will go to its safe state again.

From the modelling point-of-view, this cyclic behaviour models different trains arriving on the same track. In addition, the behaviour of all the trains is the same and we only need a way to distinguish them, typically with a unique identifier. The model will then consist of a number of *train* instances derived from the same template and a *gate controller*.

1.2 User-Defined Types

In programming languages types allow for static checks to be performed, for ensuring that variables and expressions are used in a manner that will not lead to domain incompatibility in the sense that an operation is applied to a value that is *not* in its domain of arguments. The ability to define new types allow the user to identify and name value domains that are *not* primitive of the language being used, say "stacks" of characters equipped with operations for clearing, pushing, and popping a stack, selecting the top component, and testing for emptiness. For the modelling formalisms of UPPAAL a similar design decision has been taken.

The gate controller will typically need a queue to keep track of stopped trains and restart them. Trains are distinguished with a unique identifier whose range is defined by the total number of trains. It is natural and safe from a modelling point-of-view to declare a type for this identifier being a bounded integer. Furthermore we can structure the queue into one type that contains an array and a length. User-defined types are declared with the syntax

typedef type name;

where the type can be a bounded integer (*int*[min,max]) or a structure declared as

struct { type1; type2; ... }

In our example, the global declaration contains the following:

```
const   int N = 6;
typedef int[0,N-1] id_t;

chan        appr[N], stop[N], leave[N];
urgent chan go[N];
```

Here `id_t` is the type for identifiers that is used as argument for the template of trains. In addition, arrays of channels are declared for the communication between the gate and every train. Furthermore, the template of the gate has its own local declarations:

```
typedef struct {
        id_t list[N];
        int[0,N] len;
} queue_t;

queue_t q;
```

The type `queue_t` defines the domain of queues as records consisting of an array of identifiers and an associated length. The variable `q` over this type constitutes the actual queue to be used by the gate. Being declared locally, both `queue_t` and `q` are only visible within the gate template.

Figure 1.2 shows the templates used for the trains (a) and the gate (b). The train template has the argument `const id_t id` that defines its identifier. Its states

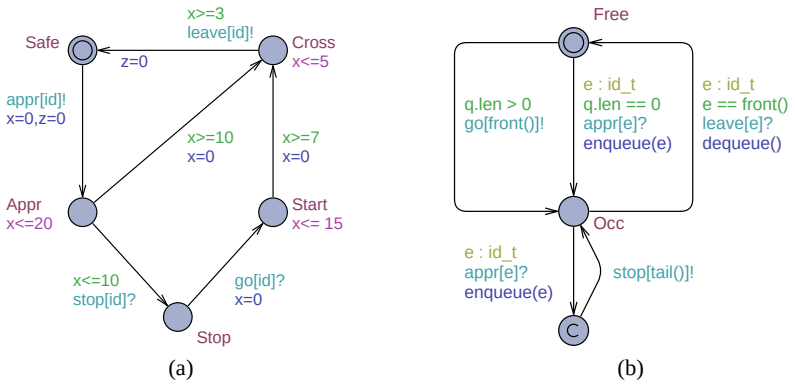


Fig. 1.2 Template for the trains (a) and the gate (b).

correspond to the states in Fig. 1.1. The communication with the gate is done with the channels with this identifier. When trains are approaching, the gate controller is notified with `appr[id]!` and when they leave the bridge they notify with `leave[id]!`. The gate controller can stop a train and then restart it. Trains listen on `stop[id]?` and `go[id]?` for this purpose. The different timing constraints of Fig. 1.1 are modelled with invariants on states and guards. Trains have a local clock `x` for this purpose (the purpose of the additional clock `z` will be explained later).

The gate controller uses and manipulates its queue structure in the automaton. When trains are approaching the controller enqueues their identifiers and dequeues them when they leave the bridge. If trains are approaching when the bridge is occupied (state *Occ*) they are stopped and their identifiers enqueued. The model is made more compact by using the so-called *select* statement `e: id_t` to unfold the corresponding edge with `e` ranging over the type `id_t`. More details on this useful feature will be given in section 1.4. To keep the template readable, queueing and dequeuing are performed with the help of user-defined functions, as will be detailed in the next section 1.3.

We note that one location of the gate is marked with "C" indicating that it is *Committed*. If a location is *Committed* then this means that time can not elapse within this location, similar to the condition for *Urgent* locations (see section **??**). However, for *Committed* locations transitions are in addition restricted only to those leaving committed locations. This removes interleaving between processes and allows the user to model atomic sequences of actions to do, e.g., a multicast[1].

[1] Broadcast channels are supported and are declared by prefixing the channel declaration by *broadcast*.

1.3 User-defined functions

As a recommendation to the programmer, in its formulation by Benjamin C. Pierce [24], the *Abstraction Principle* reads: “Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts”.

With the availability of discrete variables (integer, boolean and even structured variables as well as variables over user-defined types), their updates quickly become more involved, e.g., inserting an element into a sorted array. Updates of discrete variables take place on transitions as sequences of simple assignments. Instead of using complex automata to encode an update (even using committed states for that purpose), it is far more convenient, compact, and efficient to use a function that can pack complex control flow constructs such as nested conditional statements and loops. Thus, concerning the imperative part of a model UPPAAL provides support for the principle of abstraction.

For managing the queue of train identifiers of the gate controller we use functions to queue, dequeue, and access the tail and front of the queue. The declaration is shown in figure 1.3. C-like syntax is used with the extension of references (like in C++) and without pointers. Enqueuing adds an element at the end of the queue and increases the length of the queue. Dequeuing removes the front element and shifts all the elements. Here we point out the final reset of the last element to zero. From a programming point of view this reset seems completely superfluos as this element is no longer part of the queue and will have no effect on the subsequent behaviour. In fact an optimizing compiler would most likely remove the reset. However, in a model checker this reset to a default value is key to limit the state-space explosion problem. If we were not reseting here, the queue would remember the last element that was there, even though it is no longer in the queue and has no relevance for the future behaviour of the system. Thus states that are behaviourally equivalent, would be different, thus impacting the performance of the model-checker (you may want to check the validity of this claim yourself!). The functions for reading the front and tail elements of the queue are straight-forward.

While writing these functions in C is simple, implementing the same functionality in “pure” timed automata with simple updates is complex and error prone (but possible).

```
// Put an element at the end of the queue
void enqueue(id_t element)
{
        q.list[q.len++] = element;
}

// Remove the front element of the queue
void dequeue()
{
        int i = 0;
        q.len -= 1;
        while (i < q.len)
        {
                q.list[i] = q.list[i + 1];
                i++;
        }
        q.list[i] = 0;
}

// Returns the front element of the queue
id_t front()
{
   return q.list[0];
}

// Returns the last element of the queue
id_t tail()
{
   return q.list[q.len - 1];
}
```

Fig. 1.3 Declaration of the functions used locally by the gate controller.

1.4 Select label

The gate controller of figure 1.2.(b) is using a *select* statement. This statement has the effect of duplicating the edge into several instances with the variable(s) used in the select taking values over the specified range(s), which in fact could be any type (here `id_t`). This construct is useful for models with a parameterised number of processes having to synchronise. Arrays of channels can be used for that purpose, e.g. allowing in our example the controller to know with which train it is synchronising and subsequently store its identifier in its queue. This can also be interpreted as message passing using (here) the `appr` channel that carries the identifier of the approaching train. Fig. 1.4 shows the window used to edit that edge. The variable `e` declared in the select label has its scope on all the labels of that edge, here it is used in both the guard and synchronisation labels.

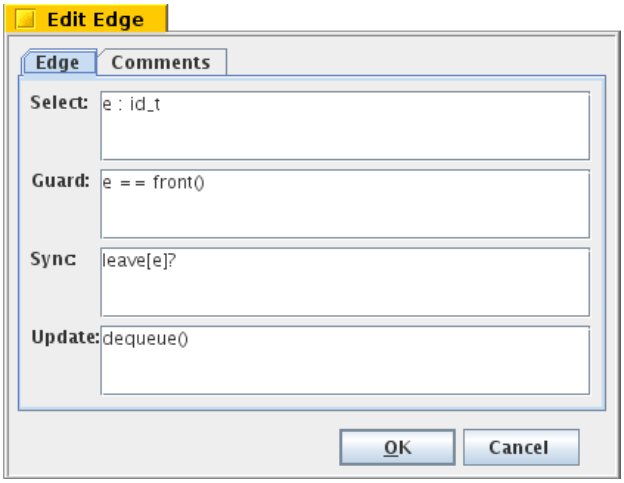


Fig. 1.4 Window for editing edges showing the different editable labels.

1.5 The Simulator Revisited

Having now completed the modeling of the Train Gate example, the first thing to examine the behaviour using the simulator of UPPAAL. Figure 1.5 shows a screenshot of the simulator, while simulating an instance of the Train Gate with six trains. From the simulation of the Jobshop example of the previous chapter, we recognize various parts of the simulator tab. The left part allows the user to control the simulation by choosing transitions and playing traces. The right part shows the automata with the active locations and the transitions that are taken, and below them a message sequence chart that shows the synchronisations between the automata.

However, whereas the center part of the simulator tab was completely empty for the Jobshop example, it now contains quite some information. In fact the center part provides information about the current value of variables and clocks. The user may select which information (s)he wants to pay attention to during a given simulation, by hiding (or viewing) processes and variables using the *View* menu.

Turning to the information offered for clocks, we see that the simulator does not exhibit concrete (real) values for the clocks but rather a collection of constraints on individual clocks, e.g. $x \leq 4$, or constraints on clock differences, e.g. $x - y < 10$. This reflects the fact, the model checking engine of UPPAAL does *not* perform state-space exploration based on concrete states with concrete values of clocks (which would be impossible due to the uncountably many such values) but rather based on *sets of clock values* described by such simple constraints. Sets of clock valuations described by constraints on clocks and clock differences are called *zones*, and may be represented in a canonical manner by so-called *difference bound matrices (DBM)*, where entries $b_{i,j}$ describe the upper bound on a clock difference $x_i - x_j$. In the *View* menu, the amount of information presented for clocks in the center part may be

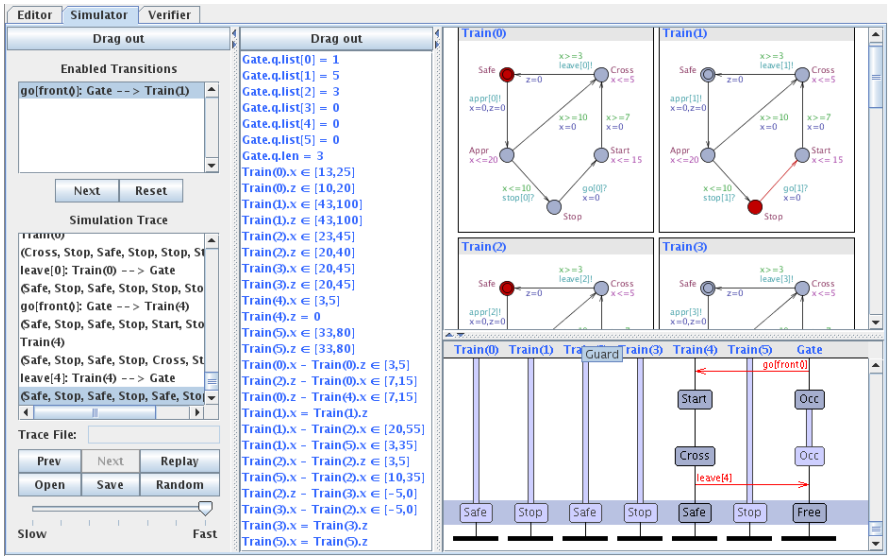


Fig. 1.5 Screenshot of the simulator running of the train-gate example.

affected. When the option *Full DBM* is selected, the all constraints present in the DBM will be shown. When the option is not selected, a reduced (but semantically equivalent) list of constraint is shown.

Thus, the model checking engine as well as the simulator of UPPAAL operates on *symbolic states* being tuples of the form (L,Z,V), where L is the location vector (active locations for all automata), Z is a zone, and V is the variable vector (values of all the integers). Figure 1.6 illustrates the sequence of symbolic states encountered during simulation of a simple timed automaton with two clocks x and y. As indicated in (a) simulation starts from the initial state where both x and y has the value 0.

- (a) From the intial state it is possible to delay and reach all states that respect the invariant of the initial location (i.e. $x \leq 4$). As the two clocks x and y increase in perfect synchrony, the resulting zone may be describe by the constraints: $x = y$ and $x \in [0,4]$.
- (b) Taking the transition that resets the clock y results in the zone described by $y = 0$ and $x \in [0,4]$
- (c) From each of these clock values (or points) delaying is again possible (except for $x = 4, y = 0$), which gives the zone described by the constraints $y \geq 0$ and $y \leq x$ and $x \in [0,4]$.
- (d) Finally, taking the transition guarded by $y >= 2$ adds this constraint resulting in the zone described by $y \geq 0$ and $y \leq x$ and $x \in [2,4]$.

Internally in the tool, these logical constraints are represented as a matrix with one extra special reference clock used for lower and upper bounds on individual clocks, e.g., $x \in [0,4]$. This representation also explains the limitation on the syntax for guards, namely they must be a conjunction of constraints of the form $x_i - x_j \leq b_{ij}$

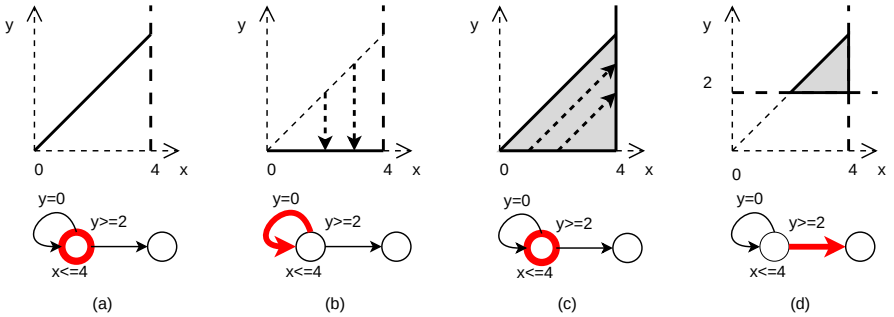


Fig. 1.6 Exploration of symbolic states. Starting from the origin $(0,0)$, (a) shows all the states reachable by delaying, (b) depicts the reset on the clock y, (c) shows a subsequent delay, and (d) shows the states that can take a transition guarded by $y \geq 2$.

(or a strict inequality). Within the simulator of UPPAAL, when selecting a transition, the variable view is updated to the symbolic state that *can* take that transition and the constraints change. When a transition is taken, the constraints are updated to reflect the resets and the delay that follows.

1.6 Queries Revisited

In the previous chapter, we considered basic UPPAAL queries of the types `A[]` ϕ and `E<>` ϕ for specifying safety and reachability properties of the job-shop example. As stated, these notations come from the field of temporal logic:

> In a temporal logic we can then express statements like ”I am always hungry”, ”I will eventually be hungry”, or ”I will be hungry until I eat something”.
>
> Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems. For instance, one may wish to say that whenever a request is made, access to a resource is eventually granted, but it is never granted to two requestors simultaneously.

Two early contenders in formal verifications were Linear Temporal Logic, LTL (Amir Pnueli and Zohar Manna) and Computation Tree Logic, CTL (Edmund Clarke and E. Allen Emerson). In this section we will detail the full query language of UPPAAL, which is in fact a subset of CTL. Figure 1.7 illustrates the five formula-types supported by UPPAAL: `A[]` ϕ, `E<>`ϕ, `A<>`ϕ, `E[]` ϕ and $(\phi \rightsquigarrow \psi)$. The main restriction compared to full CTL is that UPPAAL does not allow nesting of formula, i.e. in the above ψ or ϕ must be state predicates refering only to locations, clocks and variables.

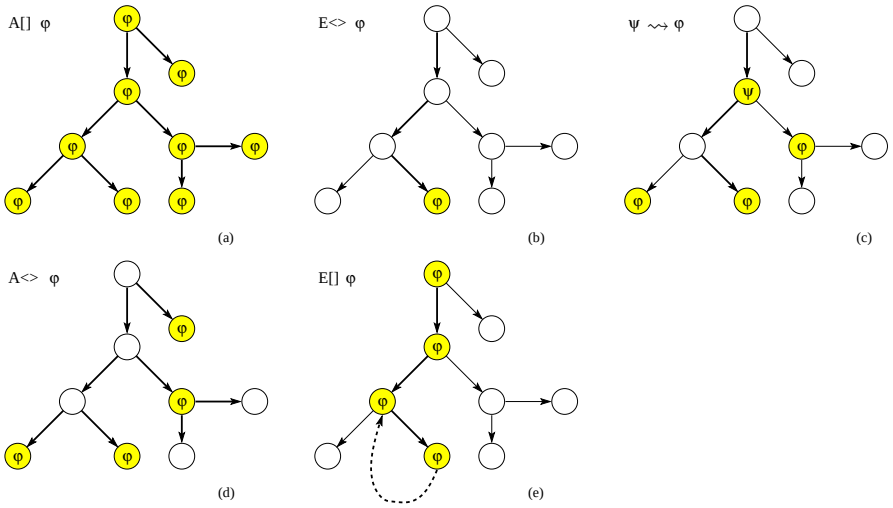


Fig. 1.7 The different types of logic formulas supported by UPPAAL.

1.6.1 Reachability

Reachability properties are of the form `E<>` ϕ and mean *there exists some path on which* ϕ *holds at some state* (Fig. 1.7.(b)). Reachability properties are useful for checking that models proposed at early design stages possess expected basic behaviours and to ask for diagnostic traces to confirm and study this more closely. For the Train Gate example such sanity properties could be:

```
E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall(i:id_t) i!=0 imply Train(i).Stop)
```

serving to check that the gate can be occupied, that trains 0 or 1 can cross, that train 0 can cross while train 1 is stopped, and that train 0 can cross while all *other* trains are stopped. In the last property – expected but maybe difficult to exhibit using manual or random simulation – *forall* is used over a range of indices.

1.6.2 Safety

Safety properties are of the form `A[]` ϕ and mean that *for all paths and for all states on those paths* ϕ *holds* (Fig. 1.7.(a)). We note that `E<>` $\neg\phi = \neg$ `A[]` ϕ, which means that `E<>` $\neg\phi$ gives a counter example in terms of a trace to a state that does not satisfy ϕ. For the Train Gate example expected safety properties are:

```
A[] forall (i:id_t) forall (j:id_t) \
    Train(i).Cross && Train(j).Cross imply i==j
```

```
A[] not deadlock
```

Here the first safety property expresses that the gate controller correctly implements mutual exclusion of the bridge, in that no two different trains can be in the crossing simultaneously. The nested usage of the *forall* construct ranging over `id_t`, ensures that the formula correctly (and conviniently) expresses mutual exclusion regardless of the number of trains.

Other properties that also fall within the category of safety properties are of the form `E[]` ϕ, that means *there exists a path on which* ϕ *always holds* (Fig. 1.7.(e)).

1.6.3 Liveness

Whereas safety properties are usefull for expressing "that something bad will never happen", they are not sufficient for ensuring that a designed system is adequate. Given the Train Gate example it is utterly simple to obtain a safe system guaranteeing no crashes on the bridge: simply use a gate controller that will stop all trains! Clearly, this is not satisfactory.

What is needed is the additional ability to express liveness properties of a system in the sense "that something good is guaranteed to eventually happen". The first liveness property has the form `A<>` ϕ expressing that *for all paths* ϕ *eventually holds* (Fig. 1.7.(d)). We note that `E[]` $\neg\phi = \neg$ `A<>` ϕ, which means that `E[]` $\neg\phi$ gives a counter example to the liveness property `A<>` ϕ in the form of an infinite path (witnessed as a loop) or a path that ends on a deadlock on which ϕ does not hold.

The second, and particularly useful, liveness property has the form ϕ `-->` ψ and should be read as ϕ *leads to* ψ. In fact this property is equivalent to (and a shorthand for) the formula `A[](`ϕ `imply A<>` ψ`)`, and means that *whenever* ϕ *holds for a state, then* ψ *will always hold eventually for all paths starting from that state* (Fig. 1.7.(c)). More interestingly is its usage as a *time bounded liveness* property with the help of an observer as shown in Fig. 1.16.(a) of Section 1.9.

In our Train Gate example, we may want to ensure that whenever a train is approaching it eventually will be at crossing. This will clearly rule out the inadequate solution of a controller which (purposely) stops all train, or wrongly implemented controllers under which some trains might get stuck in the queue.

```
Train(0).App --> Train(0).Cross
Train(1).App --> Train(1).Cross
Train(2).App --> Train(2).Cross
...
```

1.6.4 Bounded Liveness and Performance Evaluation

Having studied the correctness of the model, we may be interested in its performance. Though it is essential to know that trains approaching will eventually reach

the crossing, we may additionally want to obtain lower and upper bounds on the time between trains being in the *Appr* and *Cross* locations. For this, we add a clock `z` to the model as shown in Fig. 1.2. The clock is reset whenever a train is approaching[2]. To determine the relevant time-bounds we perform a binary search using properties of the type

```
A[] Train(0).Cross imply Train(0).z <= UP
A[] Train(0).Cross imply Train(0).z >= LOW
```

with `UP` and `LOW` being constants used in the binary search, e.g., 1000, 500, 250, 125, ... Adding an extra clock to the model that is reset when we want to measure some time and asking a safety or reachability property on a specific state is a technique used for *bounded liveness*. We note that this is cheaper than liveness.

As an attractive alternative to performing the (manual) binary search, we may use the queries `inf{Pr}:exp` and `sup{Pr}:exp`, which returns the infimum (supremum) of the expression `exp` over all reachable states satisfying the state predicate `Pr`. For the Train Gate example the queries

```
inf{Train(0).Cross}: Train(0).z
sup{Train(0).Cross}: Train(0).z
```

will give us the desired time-bounds directly for this clock in the state *Train(0).Cross*. The state predicate is optional and not putting the brackets at all is the same as having *true* as the predicate. The bounds are here $7 \leq z \leq 125$ for 6 trains.

We could also have used a *stop-watch*, which is sometimes simpler, i.e., by adding the invariant `z'==0` to the state *Safe* to stop the clock and reseting it when entering this state. Then we would ask `sup: Train(0).z`, which gives the same upper bound. We note that using stop-watches makes the reachability problem undecidable but UPPAAL uses an over-approximation technique so the result is reliable. In this example, the bound is exact but it could have been looser.

1.7 Verification Options

The model checking technology comes with the curse of state space explosion, i.e. the number of states to be explored tend to grow exponentially with the number of components (automata, clocks, variables, etc.) of the model. Development of techniques for state-space representation and exploration for making model checking efficient in practice is an extremely active area. Benefitting from (and contributing to) this research, UPPAAL offers a number of verification options to affect the representation and exploration of the veririfcation engine. These are available in the GUI under the *Options* menu.

[2] The reset to *Safe* is to reduce the state-space and has an impact because as the clock is used in the property, it is always active.

1.7.1 Search Order

The classical search ordering depth-first and breadth-first search are supported, as well as random depth-first (Fig. 1.8). In scheduling models where one solution is wanted, depth-first (or random depth-first) will typically be the most efficient option. There is another option available in the 4.1.x versions, closest to target first. This is an experimental heuristic used to guide the search.

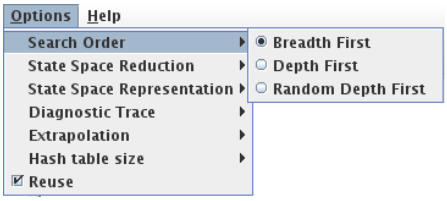


Fig. 1.8 Search order options.

1.7.2 State Space Reduction

These options are used to reduce the size of the *stored* state-space (Fig. 1.9). No optimisation can be chosen (none), committed states may not be stored unless they start a loop (conservative), only states starting loops will be stored (aggressive), or no state at all will be stored (extreme). The last option should be used with caution and is useful only when the model guarantees progress or is acyclic.

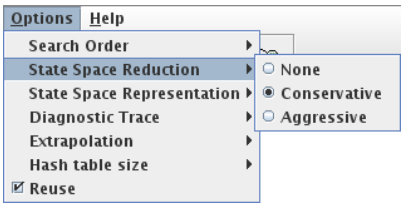


Fig. 1.9 State space reduction options.

1.7.3 State Space Representation

These options specify how to store individual state (Fig. 1.10.(a)). The option *DBM* uses the canonical matrix representation of constraints known as *difference bound matrix*. The option *Compact Data Structure* computes a reduced set of necessary constraints to store, which costs time but reduces memory footprint. The option *Under approximation* activate the *bit-state hashing* technique where every state is stored as one bit in a big hash table whose size is specified in the *Hash table size* option (Fig. 1.10.(b)). Finally the option *Over approximation* merges states together internally using an over-approximation technique (known as *convex-hull*), which reduces the number of states.

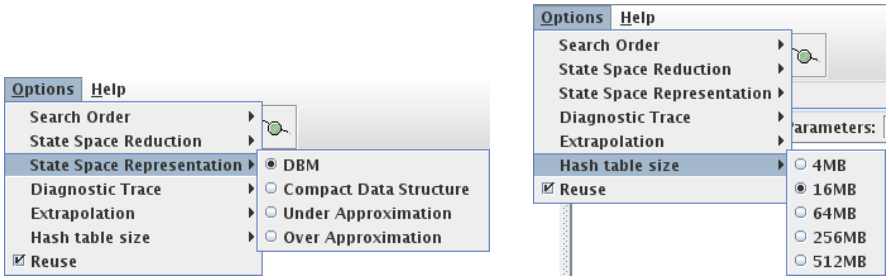


Fig. 1.10 State space representation (a) and hash table size (b) options.

1.7.4 Diagnostic Trace

To obtain a trace, a different option than "none" should be selected. When this is done, only one property at a time may be checked. Some trace may be obtained, or the shortest possible trace w.r.t. the number of steps, or the fastest w.r.t. time.

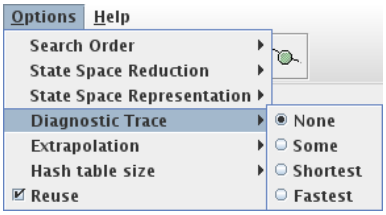


Fig. 1.11 Diagnostic trace options.

1.7.5 Extrapolation

Our verifier is using a symbolic technique to explore the state-space: rther than operating on concrete states with concrete values of clocks, the symbolic technique operates on *sets of clock values* (so-called zones) represented by constraints on clocks, $x \leq c$, and constraints on clock differences ,$x - y \leq c$. When analysing a particular timed automaton, static analysis will reveal that for each clock x there is a threshold value c_x above which the exact value of x is irrelevant for the behaviour of the timed automaton. This observation is crucial for abstractions (widening) of zones to obtain a finite symbolic state-space. This is in essence what the extrapolation does. It turns out there are different proposals for exptrapolation, that will be either exact or an over-approximation depending on the type of constraints used in the model. It is recommended to leave that option to *automatic*. The other settings are *none* (may prevent termination), *difference* (a more expensive operation useful when constraints of

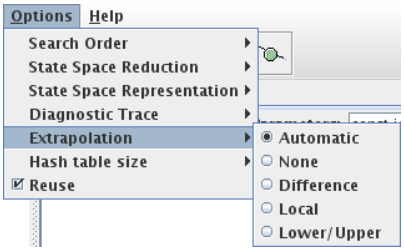


Fig. 1.12 Extrapolation options.

the form $x - y \leq c$ are used), *local* (the default), and *lower/upper* (an optimisation applicable for certain models).

1.7.6 Reuse

When checking consecutive reachability or safety properties, the model-checker may reuse the generated states if that option is selected. We note that alternating reachability and liveness property will cancel the benefit of that option.

1.7.7 Impact

Table 1.1 shows the impact of some of these options on a few examples that can be found on *http://www.uppaal.org* under examples/benchmarks. The example csma is a collision detection protocol, here with 10 nodes. Then we experiment with Fischer's protocol, a mutual exclusion protocol with 10 nodes here. Finally we use a token ring FDDI (fiber distributed data interface) protocol with 25 nodes. The properties checked here are safety properties and the depth-first search option makes the search a lot slower. This is explained by inclusion of symbolic states that will not be effective with that order. However, in other cases where a simple schedule is wanted, this order will work well. Deactivating compact data-structures will incur a small speed improvement and a large loss in memory compared to the default setting. The aggressive state-space representation stores fewer states and can sometimes (in these cases) give speed improvements. This is explained by the inclusion check that is done on fewer states.

	def	*dfs*	*S2*	*-C*
csma-10	5.8s	115s	5.6s	5.1s
	15.4M	16.5M	15.4M	24.6M
fischer-10	24.7s	111s	21.2s	20.6s
	23.9M	21.2M	15M	30.6M
fddi-25	4.4s	*	2.5s	3.8s
	13.7M	*	10.9M	29M

Table 1.1 Performance comparison with different options. The option *def* corresponds to the default setting, which is breadth-first search, compact data structure, and conservative state-space representation. The option *dfs* only changes the search order to depth-first. The option *S2* is the default setting changed with aggressive state-space representation. Finally the option *-C* is the default setting without the compact data-structure. We show results in seconds and MBytes obtained on a PentiumD running at 2.8GHz. The entries '*' mark an experiment that was stopped because it was taking more than 3 minutes.

1.8 Gossipping Girls: A Case-Study for Efficient Modelling

In this section we iterate over different versions of models to solve the gossipping girls problem (mentioned in the previous chapter), a notoriously difficult combinatorial problem. The goal is to expose the inherent limits of the model-checking technique, known as state-space explosion, and see how to change a model to improve performance.

The gossiping girls problem. Let n girls have each a private secret they wish to share with each other. Every girl can call another girl and after a conversation, both girls know mutually all their secrets. The problem is to find out how many calls are necessary so that all the girls know all the secrets. A variant of the problem is to add time to conversations and ask how much time is necessary to exchange all the secrets, allowing concurrent calls.

The basic formulation of the problem is not timed and is typically a combinatorial problem with a string of n bits that may take (at most) 2^n values for every girl. That means we have in total a string of n^2 bits taking 2^{n^2} values (in product with other states of the system).

1.8.1 Modelling in UPPAAL

We face choices regarding the representation of the secrets and where to store them. Every girl keeps track of her known secrets. The natural encoding would be to use an array of booleans. One could think of using a more compact encoding by choosing to use one integer to do so and to manage the bits manually as booleans. This limits the model to the number of bits available but as we have seen from the complexity, the state-space explodes too quickly for this to be a limiting factor. We will explore both encodings to see that the "optimized" version using integers is in fact not convenient at all for further refinements of the model. The second choice is where to store the messages, in one shared table or locally with every girl. The models described here can be found at *http://www.cs.aau.dk/~adavid/GossippingGirls/*.

In the following sub-sections we present different versions of the model. They will all use these common global declarations:

```
const int GIRLS = 4;
typedef int[0,GIRLS-1] girl_t;
chan phone[girl_t], reply[girl_t];
```

The declaration of the constant GIRLS allows us to scale the model easily. Notice that it is possible to declare that arrays of channels are indexed by a given type, which implicitly gives them the right size. This is useful for an optimization seen later.

The girl template is named Girl and has girl_t id as parameter. A first version of the the template is shown in figure 1.13. Every girl has a different ID. The *system* declaration is simply: system Girl;. This makes use of the *auto-instantiation* feature of UPPAAL. All instances of the template Girl ranging over its parameters are generated. The number of instances is controlled by the constant GIRLS.

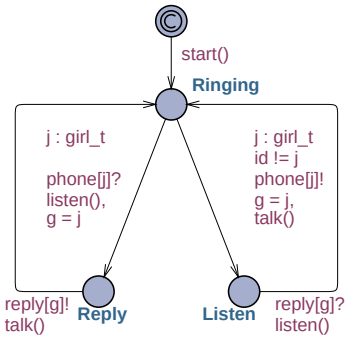


Fig. 1.13 First attempt for modelling the gossipping girls. The model consists in different instances of the same template with different identifiers to differentiate the girls. This is the template of a girl, taking as argument an identifier of type `girl_t`.

The template of a girl uses functions on its transitions to handle communication. It is good practice to use such function to improve readability of the model and to make it more flexible. We will change the internal data-structures and implementation of these functions but still keep the same automaton. Since the identifier parameter is a template argument, its scope is the whole template, which means it can be used directly in any local function. Here the start() function will initialize the girl with her unique secret (which depends on her identifier). The functions talk() and listen() aure used to send and receive secrets to and from other girls. The synchronization is done with the channels corresponding to other girls. We note that for replying, a girls needs to remember who she talked to so the model keeps track of that with a local variable `girl_t g`.

1.8.2 Representing Secrets With Boolean Arrays

We need to encode message passing between different processes, which is not directly supported by UPPAAL. To do so, the standard way is to declare a temporary shared variable. In addition, this variable is prefixed with the keyword *meta*, which means that it is a special temporary variable that will *not* be part of the states. This means that users should never refer to it between two states. Its value is only reliable on one given transition (possibly involving several edges in case of a synchronization).

We add to the global declarations meta bool tmp[girl_t]; to encode message passing. The functions mentioned previously are implemented as follows:

```
bool secrets[girl_t];
void start()  { secrets[id] = true; }
void talk()   { tmp = secrets; }
void listen() { for(i:girl_t) secrets[i] |= tmp[i]; }
```

In this version we use assignment between arrays for talk(). The function listen() uses an iterator. The template automaton is given in figure 1.13 and is common for both versions gossip1 (boolean encoding) and gossip0 (integer encoding). This first attempt captures the fact that we want the model to be symmetric with respect to sending and receiving and is quite natural with symmetric uses of talk() and listen().

Initialisation is done by setting secret *id* to true. The initial committed location ensures all girls are initialised before they start to exchange secrets. Then we have a standard message passing using a shared variable with the receiver merging the secrets sent with her own (logical or).

1.8.3 Representing Secretes With Integers

This time we add `meta int tmp;` to pass integer messages between the girls. The functions are now implemented as follows to manage the integers' bits:

```
int secrets;
void start()  { secrets = 1 << id; }
void talk()   { tmp = secrets; }
void listen() { secrets |= tmp; }
```

Initialisation is done here by setting bit *id* to one. The other functions are similar to the boolean encoding but manipulating all bits at once this time.

1.8.4 Basic Improvements

Basic optimisations of a model.

1. Avoid useless interleavings by using committed locations.
2. Make sure to model exactly what you need and not more.
3. Use *active variable reduction*, which is to reset a variable to a fixed known value, whenever its value is not relevant to the current state.

Let us now apply and illustrate the above basic optimization rules using the Gossiping Girls example:

1. The intermediate state `Listen` should be made committed otherwise all interleaving of half-started and complete calls will occur.
2. One select statement is enough because we are modelling something else here, namely girl `id` selects a channel indexed by `j` and *any other* girl that selects the same channel index can communicate with girl `id`.
3. The local variable `g` contributes badly to the state-space when its value is not relevant, i.e., the previous communication does not need to be kept. We can set it in a symmetric manner upon the start and reset it after communication to `id`.

The updated model is shown in Fig. 1.14. This update is for both boolean (gossip3) and integer (gossip2) encodings. The template keeps as an invariant that the variable `g` is always equal to `id` whenever it is not sending. In addition, when a channel `j` is selected, then it corresponds to exactly girl `j`. Only one committed location is enough but it is a good practice to mark them both. It is more explicit when we read the model. The previous versions could only be checked up to 4 girls, now we can check 5 within roughly the same time. This is a very good improvement considering the exponential complexity of the problem.

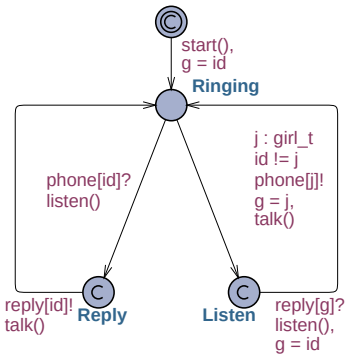


Fig. 1.14 Improved model of the gossipping girls.

1.8.5 Abstracting The Communication Protocol

We can abstract which communication line is used by declaring only one channel `chan call`. Since the semantics says that any pair of enabled edges (call!,call?) can be taken, we do not need to make an extra select. In addition, processes cannot synchronise with themselves so we do not need this check either. The downside is that we lose the information on the receiver from the sender point of view. We do not need this in our case. We can get rid of the local variable `g` as well. We can also simplify the communication protocol by merging the sequence `listen()`-talk() into one function and simplify `listen()` to a simple assignment since we know that the

message already contains the sent secrets. The global declaration is updated with only chan call; for the channel. The updated automaton is depicted in Fig. 1.15.

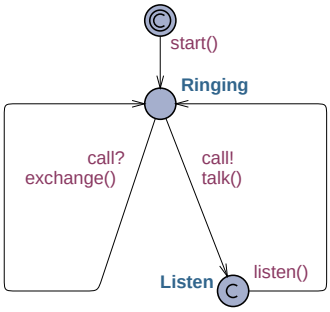


Fig. 1.15 Abstract model of the gossipping girls.

The *integer* version of the model (gossip4.xml) has the following local functions:

```
int secrets;
void start()    { secrets = 1 << id; }
void talk()     { tmp = secrets; }
void exchange() { secrets = (tmp |= secrets); }
void listen()   { secrets = tmp; }
```

The *boolean* version of the model (gossip5.xml) is changed with the functions:

```
bool secrets[girl_t];
void start()    { secrets[id] = true; }
void talk()     { tmp = secrets; }
void exchange() { for(i:girl_t) tmp[i] |= secrets[i];
                  secrets = tmp; }
void listen()   { secrets = tmp; }
```

The exchange function could have been written as follows:

```
void exchange() {
    for(i:girl_t) secrets[i] = (tmp[i] |= secrets[i]);
}
```

which is almost the same. The difference is that the number of interpreted instructions is lower in the first case. It is possible to further optimise the model by having one parameterised shared table and avoid message passing all-together. We leave this as an exercise for the reader but we notice that this change destroys the nice design with the local secrets to each process.

1.8.6 Verification

We check the reachability property that all girls will eventually know all secrets. For the integer version of the model, the property is:

```
E<> forall(i:girl_t) forall(j:girl_t)
    (Girl(i).secrets & (1 << j))
```

We can write a shorter but less obvious equivalent formula that takes advantage of the fact that $2^{GIRLS} - 1$ generates a bit mask with the first GIRLS bits set to one:

```
E<> forall(i:girl_t)
    Girl(i).secrets == ((1 << GIRLS)-1)
```

The formula for the boolean version is:

```
E<> forall(i:girl_t) forall(j:girl_t)
    Girl(i).secrets[j]
```

The formulas use the *for-all* construct, which gives compact formulas that automatically scale with the number of girls in the model. The version with the integers checks with a bit mask that the bits are set.

Table 1.2 shows the resource consumption for the different models with different number of girls. Experiments are run on an AMD Opteron 2.2GHz with UPPAAL rev. 2842. The results show how important it is to be careful with the model and to optimise the model to reduce the state-space whenever possible. We notice that the model is not even using clocks. The model with integers is faster due to its simplicity but consumes marginally less memory. The two last models (gossip6 and gossip7) are discussed in the next paragraph.

Girls	4	5	6	7
gossip0	0.6s/24M	498s/3071M	-	-
gossip1	1.0s/24M	809s/3153M	-	-
gossip2	0.1s/1.3M	0.3s/22M	71s/591M	-
gossip3	0.1s/1.3M	0.5s/22M	106s/607M	-
gossip4	0.1s/1.3M	0.2s/22M	37s/364M	-
gossip5	0.1s/1.3M	0.3s/22M	63s/381M	-
gossip6	0.1s/1.3M	0.1s/1.3M	3.4s/29M	399s/1115M
gossip7	0.1s/1.3M	0.1s/1.3M	0.3s/21M	29s/108M

Table 1.2 Resource consumption for the different models with different number of girls. Results are in seconds/M-bytes.

1.8.7 Improving Verification with Symmetry and Progress Measures

UPPAAL features two major techniques to improve verification. These techniques concern directly verification and are orthogonal to model optimisation. The first is *symmetry reduction*. Since we designed our model to be symmetric from the start, taking advantage of this feature is done by using a *scalar set* for the type girl $\perp$. The second feature is the generalised *sweep-line* method. We need to define a progress measure to help the search. Furthermore, only the model with booleans is eligible

for symmetry reduction since we cannot access individual bits in an integers in a symmetric manner (using scalars).

Symmetry reduction. This technique exploits the structure of states in order to identify symmetries that occur during verification, in order to minimize the states-space that needs to be considered. The intuition behind symmetry reduction is that the order in which state components (automata, variables, clocks, etc) are stored in a state-vector does not influence the future behaviour of the system. Ideally, the reduced state-space will have only one state representing each symmetry equivalence class. As an example consider a protocol with *n* functionally identical nodes to implement a mutual exclusion. The only difference between the nodes is their respective identifier. It is not relevant to distinguish configurations where node 1 is in state A, node 2 in state B, and node 3 in state C from configurations where node 2 is in state A, node 3 in state B, and node 1 in state C. What matter is the number of nodes being in state A, state B and state C (respectively). This technique has the potential of giving an exponential gain in both time and memory.

In UPPAAL [17] symmtry reduction is activated whenever a *scalar* set is declared. A scalar set is a set of different scalar values that can only be compared for equality. A variable of a given scalar set type can only be set to another variable of the same scalar set type. Arithmetic operations that would break symmetry are not supported.

Sweep line method [11]. In models where it is possible to define some progress in the exploration, UPPAAL can save memory by "forgetting" past states if it knows it is progressing forward in the exploration. The technique works by declaring some progress measures that are used by the model-checker to delete states and save memory when it knows that it is making progress.

The only change required to take advantage of symmetry reduction is for the definition of the type `girl_t`. We use a *scalar* set for the new model (gossip6):

```
typedef scalar[GIRLS] girl_t;
```

To activate the sweep-line optimisation, we need to define a progress measure that is cheap to compute and relevant to help the search. It is important that it is cheap to compute since it will be evaluated for every state. To do so, we add `int m;` to the global declarations, we add the progress measure definition *after* the system declaration:

```
progress { m; }
```

Finally, we compute `m` in the exchange function as follows:

```
void exchange() {
    m = 0;
    for(i:girl_t) {
```

```
            m += tmp[i] ^ secrets[i];
            tmp[i] |= secrets[i];
        }
    }
```

This measure counts the number of new messages exchanged per communication. The two last experiments in table 1.2 show that these features give gains with another order of magnitude both in speed and memory. The model still explodes exponentially, which we cannot avoid given its nature.

1.9 Modelling Tips

In this last section we summarize some of the modelling patterns that we have been using in our examples as well as present some additional ones.

1.9.1 Active variables

When the value of a variable is not important for the model, one should alwaysreset it to a default value, e.g., 0. This is what `list[i] = 0;` does in the `dequeue` function of the Train Gate example. If this statement is not here the model still works but states will keep memory of the last train that was in the queue, thus increasing the state-space needlessly. The same principle should be applied to all integers.

1.9.2 Value passing

Sometimes it is useful to have one process send a value to another. There are two ways to do that. The first is to use a *meta* variable. Such a variable is not part of the state and can be used only as a temporary place-holder on one transition. Its value is not reliable between two states. Typically the sender process synchronises with a `c!` and writes on a meta variable. Then the receiver process reads it when synchronising with a `c?`. Here `c` is a channel. In practice the instructions are executed first on the `c!` side, which explains why this trick works. A variant of this is to write directly the value into the destination variable on the sender or receiver side if the variables are shared. The drawback is that this makes for less modular modelling. The second way to encode value passing is to use arrays of channels. This is recommended for small ranges. One would declare `chan c[5];` and then send `i` with `c[i]!`. The trick is on the receiver where *select* is used with `i : int[0,4]`. The receiver can then receive with `c[i]?`.

1.9.3 Multi-cast

In cases when we want to synchronise from one process to several processes either from one central sender or in a chain, the following pattern should be used. Every step of the synchronisation should use a different channel and every intermediate location should be *committed*. Committed locations forbid interleaving with other non-committed locations. In a given committed state (i.e., a location vector, a variable vector, and a zone), only the transitions from its committed locations are allowed. In case of synchronisation, leaving one committed location (as part of the synchronisation) is enough.

1.9.4 Urgent transitions

The only way to model urgent transitions in UPPAAL is to use urgent channels. The question remains how to model a simple transition that we want to be urgent by itself? Simply add a dummy process with a self loop synchronising with `go!` on an urgent channel. The transition we want to be urgent synchronises with `go?`. We note that if the guard on that transition evaluates to false then delay is allowed (unless the state itself is urgent or committed).

1.9.5 Model Decoration and Monitors

Sometime the temporal formulas supported by the query language of UPPAAL are too limited. In such cases, an automaton acting as a monitor with an accepting or error state can help checking more complex properties involving causality between values and variables and time.

Using model decoration and monitor is a general technique that consists in adding to the original model variables or a whole automaton to monitor the state of the system. This can be used in different ways to measure delays or to detect error states with some complex causality relationship.

As an example, let us suppose that we want to check a bounded liveness property of the form $\phi \rightsquigarrow_{\leq t} \psi$, i.e., whenever ϕ holds for a state then ψ will eventually hold on all paths starting from that state within t time units. It is not possible to check this property directly using only the formulas supported. Instead we use a monitor automaton following the pattern shown in figure 1.16.(a). The states marked ϕ are those for which ϕ should hold, similarly for ψ. The boolean `b` is set to true or false in the monitor. Not shown in the automaton are the guards to go between the states that should monitor the conditions ϕ and ψ. Those transitions should be made urgent.

The bottom (dark) states are the states for which the check is active (`b` is true). The property to check is then `A[](b imply z<= t)`.

A model can also be checked for non-*zenoness* with the help of an observer. In timed automata, it is allowed to have behaviours that will let an infinite amount of transitions to be fired (take actions) within a finite amount of time. This can be done by looping or blocking time with invariants and not using resets. Unless proper guards are used, this may happen but it is seldom a desirable property of the model. By using the observer of figure 1.16.(b) (let us call it `Obs`) in parallel with the reset of the model, one can check the property `Obs.A --> Obs.B`. In the automaton `C` is a constant set to a good value w.r.t. the rest of the model. The value itself is not very important as long as it is strictly positive.

A model is *zeno* if it allows an infinite number of discrete transitions to take place in a finite amount of time. In other words, it contains a loop where time does not diverge. This is an undesirable behaviour for a real system but it is very easy to obtain with timed automata. It is sometimes useful to check that a model does not allow such behaviours.

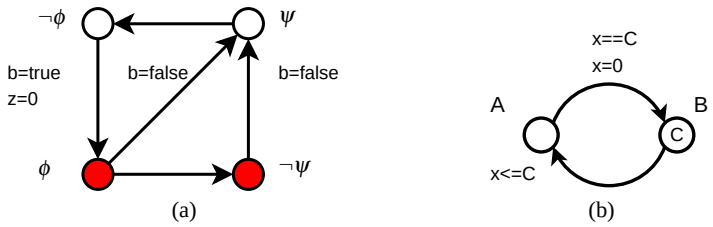


Fig. 1.16 Patterns for checking bounded liveness (a) and non-zeno behaviours (b).

1.10 Extensions of the Formalism

The UPPAAL tool suite has been specialised to different application domains. Here we mention its different variants.

CORA is a specialised version of UPPAAL that implements guided and minimal cost reachability algorithms [4, 5, 21]. It is suitable in particular for solving cost-optimal schedulability problems [2, 6]. The extension consists in adding a special `cost` variable to the model. The variable is put on location in conjunction with existing invariants in a *cost rate* expression of the form `cost' == expr` where `expr` is an expression that evaluates to a non-negative integer. In addition transition updates may have discrete cost increases with expressions of the form `cost += expr` with the same kind of expression.

TRON [20, 22] is a testing tool suited for black-box conformance testing [25, 19] of timed systems. It is mainly targeted for embedded software commonly found in various controllers. Testing is done online in the sense that that tests are derived, executed, and checked while maintaining the connection to the system in real-time.

TIGA [3] is a specialisation of UPPAAL designed to verify systems modelled as timed game automata where a controller plays against an environment. The tool synthesises code represented as a strategy to meet control objectives [14, 1, 23, 26]. The control objectives are specified as *until* or *weak-until* properties that are the more general forms of reachability and safety. The tool is based on an on-the-fly algorithm [9] and has be applied to industrial case studies [18, 10]. The tool can also handle timed games with partial observability [8] and has been extended [7] to check for simulation of timed automata and timed game automata.

PORT [15] is a version targeted to component-based modelling and verification. Its interface is developed as an Eclipse plug-in. The tool supports graphical modelling of internal component behaviour as timed automata and hierarchical composition of components. It is able to exploit the structure of such systems and apply partial order reduction techniques successfully [16].

ECDAR [13, 12] is a specialisation of TIGA that implements a recent specification theory that combines notions of specifications with a satisfaction relation, a refinement relation and a set of operators supporting stepwise design. The operators supported are composition, conjunction, and quotient. Specifications and implementations are given as timed I/O automata.

References

1. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller Synthesis for Timed Automata. In *Proc. IFAC Symp. on System Structure & Control*, pages 469–474. Elsevier Science, 1998.
2. G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader. Scheduling lacquer production by reachability analysis – a case study. In *Workshop on Parallel and Distributed Real-Time Systems 2005*, pages 140–. IEEE Computer Society, 2005.
3. Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In *Proceedings of the 19th International Conference on Computer Aided Verification*, number 4590 in LNCS, pages 121–125. Springer, 2007.
4. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer, 2001.
5. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer, 2001.
6. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

7. Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, number 5813 in LNCS, pages 73–87. Springer, 2009.
8. F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.
9. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.
10. Franck Cassez, Jan J. Jessen, Kim G. Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Automatic synthesis of robust and optimal controllers. In *Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 90–104. Springer, 2009.
11. Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2001.
12. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In *Proceedings of ATVA 2010*, page to appear, 2010.
13. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM, 2010.
14. L. De Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic Algorithms for Infinite-State Games. In *Proc. 12th Conf. on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
15. John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In *ATVA*, pages 252–257, 2008.
16. John Håkansson and Paul Pettersson. Partial Order Reduction for Verification of Real-Time Components. In *Proc. of the 5th Int. Conf. on FORMATS*, LNCS, pages 211–226. Springer-Verlag, 2007.
17. Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.
18. Jan Jakob Jessen, Jacob Illum Rasmussen, Kim G. Larsen, and Alexandre David. Guided controller synthesis for climate controller using UPPAAL-TIGA. In *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, number 4763 in LNCS, pages 227–240. Springer, 2007.
19. Moez Krichen and Stavros Tripakis. *Model Checking Software*, volume 2989 of *LNCS*, chapter Black-Box Conformance Testing for Real-Time Systems, pages 109–126. Springer-Verlag, 2004.
20. K.G. Larsen, M. Mikučionis, and B. Nielsen. Online Testing of Real-time Systems Using UPPAAL. In *FATES'04*, LNCS, pages 79–94, Linz, Austria, September 2004.
21. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer, 2001.
22. Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.

23. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. 12th Symp. on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 229–242. Springer, 1995.
24. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
25. Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.
26. S. Tripakis and K. Altisen. Controller synthesis for discrete and dense-time systems. In *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *LNCS*, pages 233–252. Springer, 1999.
27. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

Chapter 1
An Industrial Application of Uppaal: The Chess gMAC WSN Protocol*

Mathijs Schuts, Feng Zhu, Faranak Heidarian†, and Frits Vaandrager

Abstract We report on an industrial application of the timed automaton model checking tool UPPAAL in the area of wireless sensor networks (WSN). We constructed a detailled UPPAAL model of the gMAC clock synchronization algorithm for a WSN architecture that has been developed by the Dutch company Chess. Using the UPPAAL model checker, we established that in certain cases a static, fully synchronized network may eventually become unsynchronized if the current algorithm is used, even in a setting with infinitesimal clock drifts.

1.1 Introduction

Wireless sensor networks consist of autonomous devices that communicate via radio and use sensors to cooperatively monitor physical or environmental conditions. The Dutch company Chess has developed a wireless sensor network architecture in the context of the MyriaNed project. To experiment with its designs, Chess currently builds prototypes and uses advanced simulation tools. However, due to the huge number of possible network topologies and clock speeds of nodes, it is difficult to discover flaws in the clock synchronization algorithm via these methods.

Timed automata model checking has been succesfully used for the analysis of worst case scenarios for protocols that involve clock synchronization, see for instance [3, 7, 19]. To enable model checking, models need to be more abstract than for simulation, and also the size of networks that can be tackled is much smaller.

Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands, e-mail: M.Schuts@student.ru.nl, FengZu@student.ru.nl, faranak@cs.ru.nl, F.Vaandrager@cs.ru.nl.

* This is an adaptation of a paper that appeared earlier as [15].

† Research supported by NWO/EW project 612.064.610 Abstraction Refinement for Timed Systems (ARTS).

However, a big advantage is that the full state space of the model can be explored and worst case scenarios can be found. Within the context of the Quasimodo project, Chess therefore proposed the analysis of gMAC, the clock synchronization algorithm for its wireless sensor networks, as a case study for timed automata technology. An informal description of the case study was made available by Chess as a deliverable of the project [14]. Figure 1.1 displays a sensor node developed by Chess on which the gMAC algorithm runs.

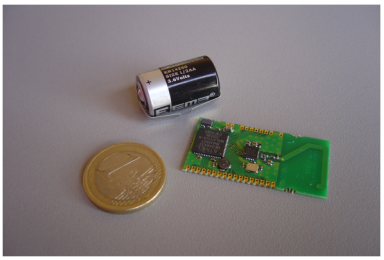


Fig. 1.1 Chess MyriaNode 2.4 Ghz wireless sensor node

Model checking projects are often carried out a posteriori: the artefact exists and has been documented in some manual of protocol standard. An advantage of such projects is that the object of modelling is clear and stable. A disadvantage may be that the potential impact of the work is limited. The Chess case study is an example of a situation in which timed automata technology has been applied during the design of a new system. In such case studies, the object of modelling is a moving target, which changes every week or sometimes every day. Also, there is not a single document describing the whole design in a consistent manner. In fact, certain aspects of the design only exist in the developers mind. The goal of our research was to find out whether state-of-the-art model checking technology is able to contribute during the design if this type of embedded real-time systems.

We constructed a detailed model of the gMAC algorithm using the input language of the timed automata model checking tool UPPAAL [2]. Our objective was to build a faithful model of the clock synchronization algorithm as presented in [14]. Nevertheless, our model does not incorporate some features of the full algorithm and network, such as dynamic slot allocation, synchronization messages, uncertain communication delays, and unreliable radio communication. At places where the informal specification of [14] was incomplete or ambiguous, the engineers from Chess kindly provided us with additional information on the way these issues are resolved in the current implementation of the network [20].

The clock synchronization algorithm that is used in the current implementation of Chess is an extension of the Median algorithm of [17]. This algorithm works reasonably well in practice, but by means of simulation experiments, Assegei [1] already exposed some flaws in the algorithm: in some test cases where new nodes join or networks merge, the algorithm fails to converge or nodes may stay out of

sync for a certain period of time. Our analysis with UPPAAL confirms these results. In fact, we demonstrated that the situation is even worse: in certain cases a static, fully synchronized network may eventually become unsynchronized if the Median algorithm is used, even in a setting with infinitesimal clock drifts.

In Section 1.2, we explain the gMAC algorithm in more detail. Section 1.3 describes our UPPAAL model of gMAC. In Section 1.4, the analysis results are described. Finally, in Section 1.5, we draw some conclusions. In this paper, we assume that the reader has a basic knowledge of the timed automaton tool UPPAAL. For a detailed account of UPPAAL, we refer to the first two chapters of this handbook. The UPPAAL model described in this paper is available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/chess09/`.

1.2 The gMAC Protocol

In this section we introduce the gMAC protocol as it has currently been implemented by Chess.

1.2.1 Background

The algorithm that we consider is part of the *Medium Access Control (MAC) layer*, which is responsible for the access to the wireless shared channel. Within its so-called gMAC protocol, Chess uses a Time Division Multiple Access (TDMA) protocol. Time is divided in fixed length *frames*, and each frame is subdivided into *slots* (see Figure 1.2). Slots can be either *active* or *sleeping* (*idle*). During active slots, a

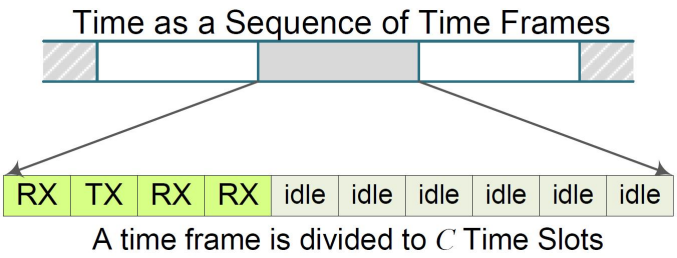


Fig. 1.2 The structure of a time frame

node is either listening for incoming messages from neighboring nodes (*RX*) or it is sending a message (*TX*). During sleeping slots a node is switched to energy saving mode. Since energy efficiency is a major concern in the design of wireless sensor networks, the number of active slots is typically much smaller than the total number

of slots (less than 1% in the current implementation). The active slots are placed in one contiguous sequence which currently is placed at the beginning of the frame. A node can only transmit a single message per time frame, during its TX slot. The protocol takes care that neighboring nodes have different TX slots.

One of the greatest challenges in the design of the MAC layer is to find suitable mechanisms for clock synchronization: we must ensure that whenever some node is sending all its immediate neighbors are awake and listening. Sensor nodes come equipped with a crystal clock, which may drift. This may cause the TDMA time slot boundaries to drift and thus lead to situations in which nodes get out of sync. To overcome this problem nodes will have to adjust their clocks now and then. Also, the notion of *guard time* is introduced: at the beginning of its TX slot, a sender waits a certain amount of time to ensure that all its neighbors are ready to receive messages. Similarly, a sender does not transmit for a certain amount of time at the end of its TX slot. In order to save energy it is important to reduce these guard times to a minimum. Many clock synchronization protocols have been proposed for wireless sensor networks, see e.g. [16, 4, 17, 11, 1, 10, 13]. However, these protocols (with the exception of [17, 1] and possibly [13]) involve a computation and/or communication overhead that is unacceptable given the extremely limited resources (energy, memory, clock cycles) available within the Chess nodes.

1.2.2 The Synchronization Algorithm

In each frame, each node broadcasts a single message to its neighbors. The timing of this message is used for synchronization purposes: a receiver may estimate the clock value of a sender based on the time when the message is received. Thus there is no need to send around (logical) clock values. In the current implementation of Chess, clock synchronization is performed once per frame using the following algorithm [1, 20]:

1. In its sending slot, a node broadcasts a packet which contains its transmission slot number.
2. Whenever a node receives a message it computes the `phase error`, that is the difference (number of clock cycles) between the expected receiving time and the actual receiving time of the incoming message. Note that the difference between the current slot number of the sender and the current slot number of the receiving node must also be taken into account when calculating the phase errors.
3. After the last active slot of each frame, a node calculates the `offset` from the phase errors of all incoming messages in this frame with the following algorithm:

```
if (number of received messages == 0)
        offset = 0;
else if (number of received messages <= 2)
  offset = phase error of first received message * gain;
else
  offset = median of all phase errors * gain
```

Here `gain` is a coefficient with value 0.5, used to prevent oscillation of the clock adjustment.

4. During the sleeping period, the local clock of each node is adjusted by the computed `offset` obtained from step 3.

In situations when two networks join, it is possible that the phases of these networks differ so much that the nodes in one network are in active slots whereas the nodes in the other network are in sleeping slots and vice versa. In this case, no messages can be exchanged between two networks. Therefore in the Chess design, a node will send an extra message in one (randomly selected) sleeping slot to increase the chance that networks can communicate and synchronize with each other. This slot is called the synchronization slot and the message is in the same format as in the transmission slot. The extreme value of `offset` can be obtained when two networks join: it may occur that the `offset` is larger than half the total number of clock cycles of sleeping slots in a frame. Chess uses another algorithm called `join` to handle this extreme case. We decided not to model joining of networks and synchronization messages because currently we do not have enough information about the `join` algorithm.

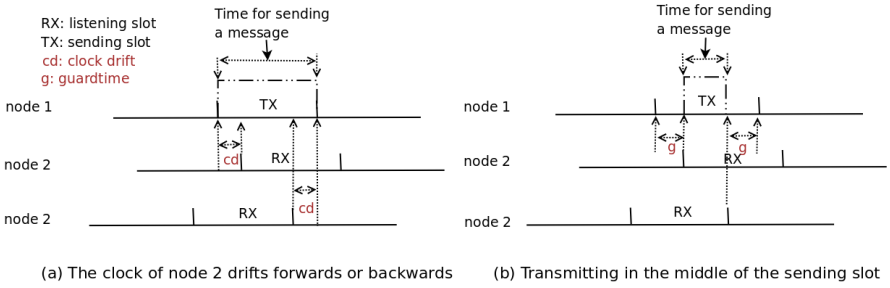
1.2.3 Guard Time

The correctness condition for gMAC that we would like to establish is that whenever a node is sending all its neighbors are in receiving mode. However, at the moment when a node enters its TX slot we cannot guarantee, due to the phase errors, that its neighbors have entered the corresponding RX slot. This problem is illustrated in Figure 1.3 (a). Given two nodes 1 and 2, if a message is transmitted during the entire sending slot of node 1 then this message may not be successfully received by node 2 because of the imperfect slot alignment. Taking the clock of node 1 as a reference, the clock of node 2 may drift backwards or forwards. In this situation, node 1 and node 2 may have a different view of the current slot number within the time interval where node 1 is sending a message.

To cope with this problem, messages are not transmitted during the entire sending slot but only in the middle, as illustrated in Figure 1.3 (b). Both at the beginning and at the end of its sending slot, node 1 does not transmit for a preset period of g clock ticks, in order to accomodate the forwards and backwards clock drift of node 2. Therefore, the time available for transmission equals the total length of the slot minus 2g clock ticks.

1.2.4 Radio Switching Time

The radio of a wireless sensor node can either be in sending mode, or in receiving mode, or in idle mode. Switching from one mode to another takes time. In the



(a) The clock of node 2 drifts forwards or backwards (b) Transmitting in the middle of the sending slot

Fig. 1.3 The need for introducing guard times

current implementation of the Chess gMAC protocol, the radio switching time is around 130μsec. The time between clock ticks is around 30μsec and the guard time g is 9 clock ticks. Hence, in the current implementation the radio switching time is smaller than the guard time, but this may change in future implementations. If the first slot in a frame is an RX slot, then the radio is switched to receiving mode some time before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, in order to keep the implementation simple, the radio is switched to the receiving mode only at the start of the RX slot. Therefore messages arriving in such receiving slots may not be fully received. This issue may also affect the performance of the synchronization algorithm.

1.3 Uppaal Model

In this section, we describe the UPPAAL model that we constructed of the gMAC protocol.

We assume a finite, fixed set of wireless nodes $\mathsf{Nodes} = \{0,\ldots,\mathsf{N}-1\}$. The behavior of an individual node $\mathsf{id} \in \mathsf{Nodes}$ is described by five timed automata $\mathsf{Clock}(\mathsf{id})$, $\mathsf{Receiver}(\mathsf{id})$, $\mathsf{Sender}(\mathsf{id})$, $\mathsf{Synchronizer}(\mathsf{id})$ and $\mathsf{Controller}(\mathsf{id})$. Figure 1.4 shows how these automata are interrelated. All components interact with the clock, although this is not shown in Figure 1.4. Automaton $\mathsf{Clock}(\mathsf{id})$ models the hardware clock of node id, automaton $\mathsf{Sender}(\mathsf{id})$ the sending of messages by the radio, automaton $\mathsf{Receiver}(\mathsf{id})$ the receiving part of the radio, automaton $\mathsf{Synchronizer}(\mathsf{id})$ the synchronization of the hardware clock, and automaton $\mathsf{Controller}(\mathsf{id})$ the control of the radio and the clock synchronization.

Table 1.1 lists the parameters that are used in the model (constants in UPPAAL terminology), together with some basic constraints. The domain of all parameters is the set of natural numbers. We will now describe the five automaton templates used in our model.

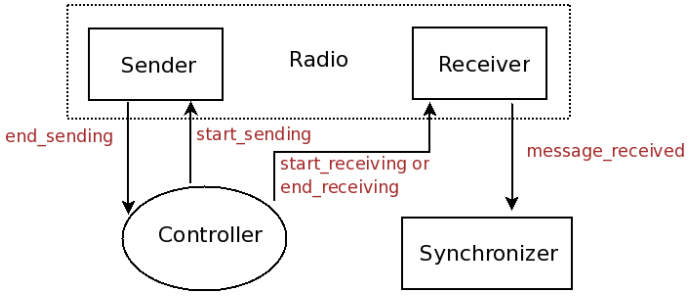


Fig. 1.4 Message flow in the model

Parameter	Description	Constraints
N	number of nodes	$0 < N$
C	number of slots in a time frame	$0 < C$
n	number of active slots in a time frame	$0 < n \leq C$
tsn[id]	TX slot number for node id	$0 \leq tsn[id] < n$
k_0	number of clock ticks in a time slot	$0 < k_0$
g	guard time	$0 < g$
r	radio switch time	$0 \leq r$
min[id]	minimal time between two clock ticks of node id	$0 < min[id]$
max[id]	maximal time between two clock ticks of node id	$min[id] \leq max[id]$

Table 1.1 Protocol parameters

Clock

Timed automaton Clock(id) models the behavior of the hardware clock of node id. The automaton is shown in Figure 1.5. At the start of the system state variable csn[id], that records the current slot number, is initialized to $C-1$, that is, to the last sleeping slot. Hardware clocks are not perfect and so we assume a minimal time min[id] and a maximal time max[id] between successive clock ticks. Integer variable clk[id] records the current value of the hardware clock. For convenience (and to reduce the size of the state space), we assume that the hardware clock is reset at the end of each slot, that is after k_0 clock ticks. Also, a state variable csn[id], which records the current slot number of node id, is updated each time at the start of a new slot.

Sender

The sending behavior of the radio is described by the automaton Sender[id] shown in Figure 1.6. The behavior is rather simple. When the controller asks the sender to transmit a message (via a start_sending[id] signal), the radio first switches to sending mode (this takes r clock ticks) and then transmits the message (this takes $k_0 - 2 \cdot g$ ticks). Immediately after the message transmission has been completed, an

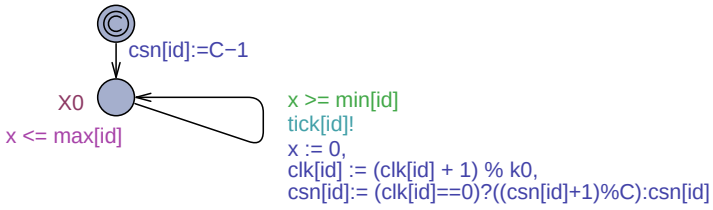


Fig. 1.5 Automaton Clock[id]

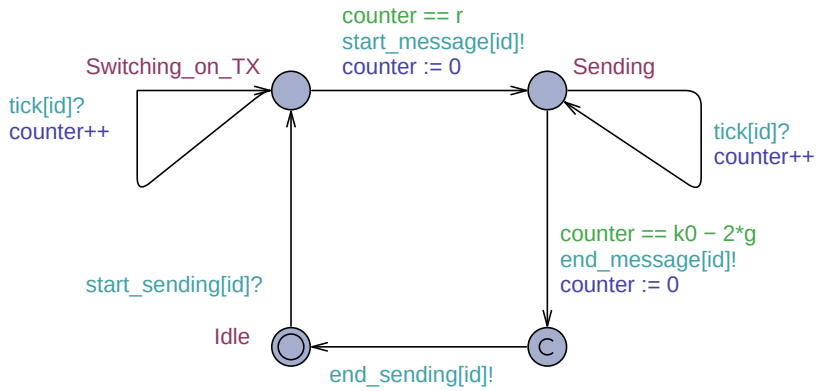


Fig. 1.6 Automaton Sender[id]

end_sending[id] signal is sent to the controller to indicate that the message has been sent.

Receiver

The automaton Receiver[id] models the receiving behavior of the radio. The automaton is shown in Figure 1.7. Again the behavior is rather simple. When the controller asks the receiver to start receiving, the receiver first switches to receiving mode (this takes r ticks). After that, the receiver may receive messages from all its neighbors. A function neighbor is used to encode the topology of the network: neighbor(j, id) holds if messages sent by j can be received by id. Whenever the receiver detects the end of a message transmission by one of its neighbors, it immediately informs the synchronizer via a message_received[id] signal. At any moment, the controller can switch off the receiver via an end_receiving[id] signal.

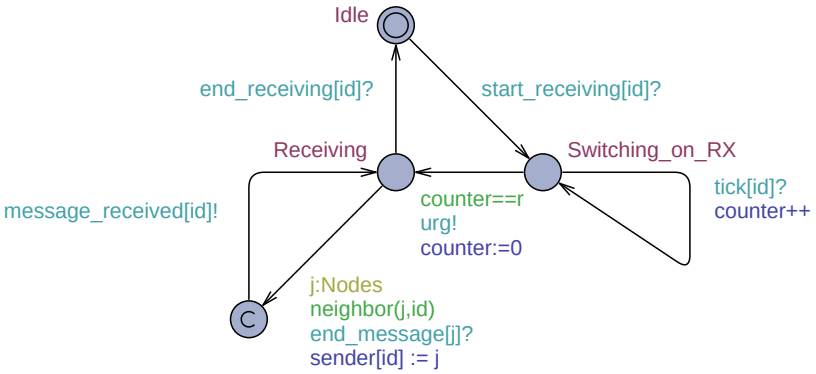


Fig. 1.7 Automaton Receiver[id]

Controller

The task of the Controller[id] automaton, displayed in Figure 1.8, is to put the radio in sending and receiving mode at the appropriate moments. Figure 1.9 shows the definition of the predicates used in this automaton. The radio should be put in sending mode r ticks before message transmission starts (at time g in the transmission slot of id). If $r > g$ then the sender needs to be activated $r - g$ ticks before the end of the slot that precedes the transmission slot. Otherwise, the sender must be activated at tick $g - r$ of the transmission slot. If the first slot in a frame is an RX slot, then the radio is switched to receiving mode r time units before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, as described in Section 1.2.4, the radio is switched to the receiving mode only at the start of the RX slot. The controller stops the radio receiver whenever either the last active slot has passed or the sender needs to be switched on.

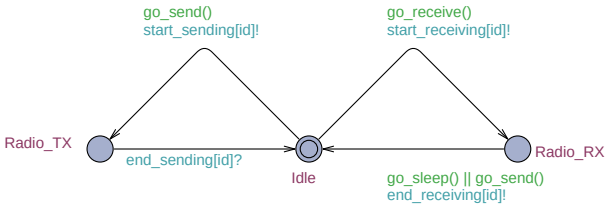


Fig. 1.8 Automaton Controller[id]

All the channels used in the Controller[id] automaton (start_sending, end_sending, start_receiving, end_receiving and synchronize) are urgent, which means that these signals are sent at the moment when the transitions are enabled.

```
bool go_send(){return (r>g)
 ?((csn[id]+1)%C==tsn[id] && clk[id]==k0-(r-g))
 :(csn[id]==tsn[id] && clk[id]==g-r);}

bool go_receive(){return
(r>0 && 0!=tsn[id] && csn[id]==C-1 && clk[id]==k0-r)
|| (r==0 && 0!=tsn[id] && csn[id]==0)
|| (0<csn[id] && csn[id]<n && csn[id]-1==tsn[id]);}

bool go_sleep(){return csn[id]==n;}
```

Fig. 1.9 Predicates used in Controller[id]

Synchronizer

Finally, automaton Synchronizer[id] is shown in Figure 1.10. The automaton maintains a list of phase differences of all messages received in the current frame, using a local array phase_errors. Local variable msg_counter records the number of messages received.

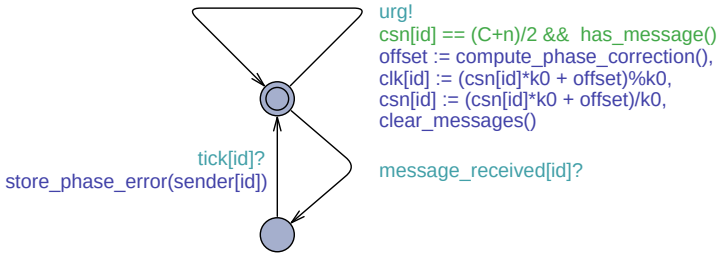


Fig. 1.10 Automaton Synchronizer[id]

```
void store_phase_error(int sender)
{
phase_errors[msg_counter] =
   (tsn[sender] * k0 + k0 - g)
        - (csn[id] * k0 + clk[id]);
msg_counter++
}
```

Fig. 1.11 Function used in Synchronizer[id]

Whenever the receiver gets a message from a neighboring node (message_received[id]), the synchronizer computes and stores the phase difference using the function store_phase_error at the next clock tick. Here the phase difference is defined as the

expected time at which the message transmission ends (`tsn[sender] * k0 + k0 - g`) minus the actual time at which the message transmission ends (`csn[id] * k0 + clk[id]`), counting from the start of the frame. The complete definition is listed in Figure 1.11. Recall that in our model we abstract from transmission delays.

As explained in Section 1.2.2, the synchronizer computes the value of the phase correction (offset) and adjusts the clock during the sleeping period of a frame.[3] Hence, in order to decide in which slot we may perform the synchronization, we need to know the maximal phase difference between two nodes. In our model, we assume no joining of networks. When a node receives a message from another node, the phase difference computed using this message will not exceed the length of an active period. Otherwise one of these two nodes will be in sleeping period while the other is sending, hence no message can be received at all. In practice, the number of sleeping slots is much larger than the number of active slots. Therefore it is safe to perform the adjustment in the middle of sleeping period because the desired property described above holds. When the value of `gain` is smaller than 1 the maximal phase difference will be even smaller.

The function of compute_phase_correction implements exactly the algorithm listed in Section 1.2.2.

1.4 Analysis Results

In this section, we present some verification results that we obtained for simple instances of the model that we described in Section 1.3. We checked the following invariant properties using the UPPAAL model checker:

```
INV1 : A[] forall (i: Nodes) forall (j : Nodes)
  SENDER(i).Sending && neighbor(i,j)imply RECEIVER(j).Receiving

INV2 : A[] forall (i:Nodes) forall (j:Nodes) forall (k:Nodes)
    (SENDER(i).Sending && neighbor(i,k) && SENDER(j).Sending
      && neighbor(j,k)) imply i == j

INV3 : A[] not deadlock
```

The first property asserts that always when some node is sending, all its neighbors are listening. The second property states that never two different neighbors of a given node are sending simultaneously. The third property states that the model contains no deadlock, in the sense that in each reachable state at least one component can make progress. The three invariants are basic sanity properties of the gMAC protocol, at least in a setting with a static topology and no transmission failures.

We used UPPAAL on a Sun Fire X4440 machine (with 4 Opteron 8356 2.3 Ghz quad-core processors and 128 Gb DDR2-667 memory) to verify instances of our

[3] Actually, in the implementation the offset is used to compute the corrected *wakeup time*, that is the moment when the next frame will start [20]. In our model we reset the clock, but this should be equivalent.

model with different number of nodes, different network topologies and different parameter values. Table 1.2 lists some of our verification results, including the resources UPPAAL needed to verify if the network is synchronized or not. In all experiments, $\mathsf{C} = 10$ and $\mathsf{k}_0 = 29$.

Clearly, the values of network parameters, in particular clock parameters min and max, affect the result of the verification. Table 1.2 shows several instances where the protocol is correct for perfect clocks ($\mathsf{min} = \mathsf{max}$) but fails when we decrease the ratio $\frac{\mathsf{min}}{\mathsf{max}}$. It is easy to see that the protocol will always fail when $\mathsf{r} \geq \mathsf{g}$. Consider any node i that is not the last one to transmit within a frame. Right after its sending slot, node i needs r ticks to get its radio into receiving mode. This means that — even with perfect clocks — after g ticks another node already has started sending even though the radio of node i is not yet receiving. Even when $\mathsf{r} < \mathsf{g}$, the radio switching time has a clear impact on correctness: the larger the radio switching time is, the larger the guard time has to be in order to ensure correctness. Using UPPAAL, we can fully analyze line topologies with at most seven nodes if all clocks are perfect. For larger networks UPPAAL runs out of memory. A full parametric analysis of this protocol will be challenging, also due to the impact of the network topology and the selected slot allocation. Using UPPAAL, we discovered that for certain topologies and slot allocations the Median algorithm may always violate the above correctness assertions, irrespective of the choice of the guard time. For example, in a 4 node-network with clique topology and min and max of 100.000 and 100.001, respectively, if the median of the clock drifts of a node becomes -1, the median algorithm divides it by 2 and generates 0 for clock correction value and indeed no synchronization happens. If this scenario repeats in three consecutive time frames for the same node, that node runs $\mathsf{g} = 3$ clock cycles behind and gets out of sync.

Another example in which the algorithm may fail is displayed in Figure 1.12. This network has 4 nodes, connected by a line topology, that send in slots 1, 2, 3, 1, respectively. Since all nodes have at most two neighbors, the Median algorithm

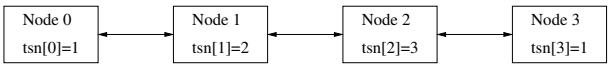


Fig. 1.12 A problematic network configuration

prescribes that nodes will correct their clocks based on the first phase error that they see in each frame. For the specific topology and slot allocation of Figure 1.12, this means that node 0 adjusts its clock based on phase errors of messages it gets from node 1, node 1 adjusts its clock based on messages from node 0, node 2 adjusts its clock based on messages from node 3, and node 3 adjusts its clock based on messages from node 2. Hence, for the purpose of clock synchronization, we have two disconnected networks! Thus, if the clock rates of nodes 0 and 1 are lower than the clock rates of nodes 2 and 3 by just an arbitrary small margin, then two subnetworks will eventually get out of sync. These observations are consistent with

N/n	Topology	g	r	$\frac{\min}{\max}$	CPU Time	Peak Memory Usage	Sync
3/3	clique	2	0	1	1.944 s	24,180 KB	YES
3/3	clique	2	0	$\frac{100,000}{100,001}$	492.533 s	158,064 KB	NO
3/3	clique	2	1	1	1.976 s	68.144 KB	YES
3/3	clique	2	0	$\frac{100,000}{100,001}$	116.68 s	68,144 KB	NO
3/3	line	2	0	1	1.068 s	68,144 KB	YES
3/3	line	2	0	$\frac{100,000}{100,000}$	441.308 s	68,144 KB	NO
3/3	line	2	1	1	1.041 s	68,144 KB	YES
3/3	line	2	1	$\frac{100,000}{100,000}$	99.274 s	68,144 KB	NO
3/3	clique	3	0	1	1.851 s	28,040 KB	YES
3/3	clique	3	0	$\frac{100,000}{100,001}$	575.085 s	272,312 KB	NO
3/3	clique	4	0	$\frac{350}{351}$	115.166 s	516,636 KB	NO
3/3	clique	4	0	$\frac{351}{352}$	147.864 s	630,044 KB	YES
3/3	clique	3	2	1	1.827 s	24,184 KB	YES
3/3	clique	3	2	$\frac{100,000}{100,001}$	109.633 s	26,056 KB	NO
3/3	clique	4	2	$\frac{100,000}{100,001}$	533.345 s	350,504 KB	NO
3/3	clique	5	2	$\frac{587}{588}$	72.473 s	332,552 KB	NO
3/3	clique	5	2	$\frac{588}{589}$	99.101 s	407,884 KB	YES
3/3	clique	3	5	1	0.076 s	21,884 KB	NO
3/3	line	3	0	1	1.05 s	23,348 KB	YES
3/3	line	3	0	$\frac{451}{452}$	29.545 s	148,012 KB	NO
3/3	line	3	0	$\frac{452}{453}$	35.257 s	148,012 KB	YES
3/3	line	3	2	1	1.052 s	22,916 KB	YES
3/3	line	3	2	$\frac{100,000}{100,001}$	82.383 s	78,360 KB	NO
3/3	line	4	2	$\frac{100,000}{100,001}$	414.201 s	53,752 KB	NO
3/3	line	5	2	$\frac{453}{454}$	33.16 s	147,796 KB	NO
3/3	line	5	2	$\frac{454}{455}$	38.811 s	162,184 KB	YES
3/3	line	3	5	1	0.048 s	78,360 KB	NO
4/4	clique	3	0	1	231.297 s	1,437,643 KB	YES
4/4	clique	3	0	$\frac{450}{451}$	Memory Exhausted		
4/4	clique	3	2	1	229.469 s	1,438,368 KB	YES
4/4	clique	3	2	$\frac{100,000}{100,001}$	14,604.531 s	2,317,040 KB	NO
4/3	line	3	0	1	4.749*s* s	94,748 KB	YES
4/3	line	3	0	$\frac{450}{451}$	Memory Exhausted		
4/3	line	3	2	1	4.738 s	94,748 KB	YES
4/3	line	3	2	$\frac{100,000}{100,001}$	1,923.655 s	1,264,844 KB	YES
5/5	clique	3	0	1	Memory Exhausted		
5/5	clique	3	2	1	Memory Exhausted		
5/3	line	3	0	1	46.54 s	249,976 KB	YES
5/3	line	3	2	1	46.489 s	250,880 KB	YES
6/3	line	3	0	1	508.19 s	2,316,416 KB	YES
6/3	line	3	2	1	502.871 s	2,317,040 KB	YES
7/3	line	3	0	1	Memory Exhausted		
7/3	line	3	2	1	Memory Exhausted		

Table 1.2 Model checking experiments

results that we obtained using UPPAAL. If, for instance, we set $\mathsf{min}[\mathsf{id}] = 99$ and $\mathsf{max}[\mathsf{id}] = 100$, for all nodes id then neither `INV1` nor `INV2` holds. In practice, it is unlikely that the above scenario will occur due to the fact that in the implementation slot allocation is random and dynamic. Due to regular changes of the slot allocation, with high probability node 1 and node 2 will now and then adjusts their clocks based on messages they receive from each other.

However, variations of the above scenario may occur in practice, even in a setting with dynamic slot allocation. In fact, the above synchronization problem is also not restricted to line topologies. We call a subset C of nodes in a network a *community* if each node in C has more neighbors within C than outside C [12]. For *any* network in which two disjoint communities can be identified, the Median algorithm allows for scenarios in which these two parts become unsynchronized. Due to the median voting mechanism, the phase errors of nodes outside a community will not affect the nodes within this community, independent of the slot allocation. Therefore, if nodes in one community A run slow and nodes in another community B run fast then the network will become unsynchronized eventually, even in a setting with infinitesimal clock drifts. Figure 1.13 gives an example of a network with two communities.

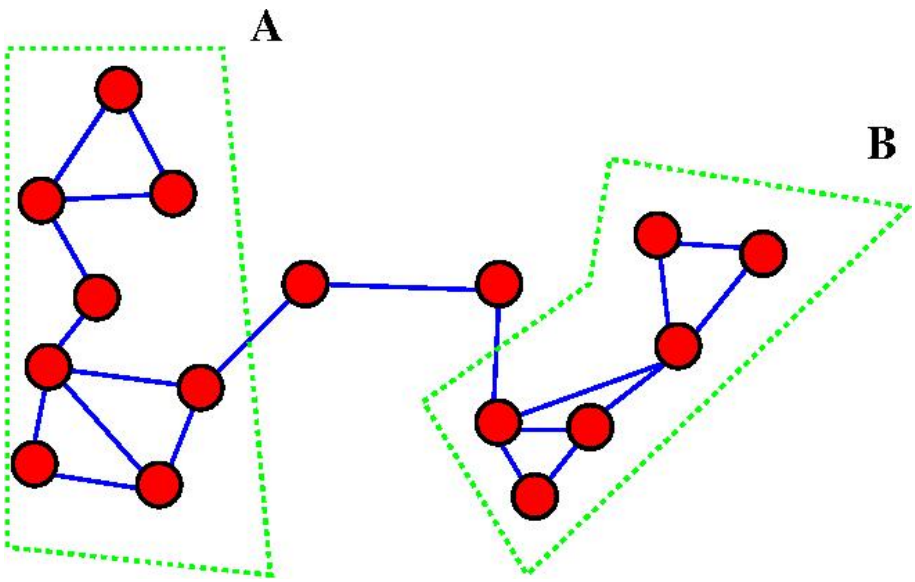


Fig. 1.13 Another problematic network configuration with two communities

Using UPPAAL, we succeeded to analyze instances of the simple network with two communities displayed in Figure 1.14. The numbers on the vertices are the node identifiers and the transmission slot numbers, respectively. Table 1.3 summarizes the results of our model checking experiments.

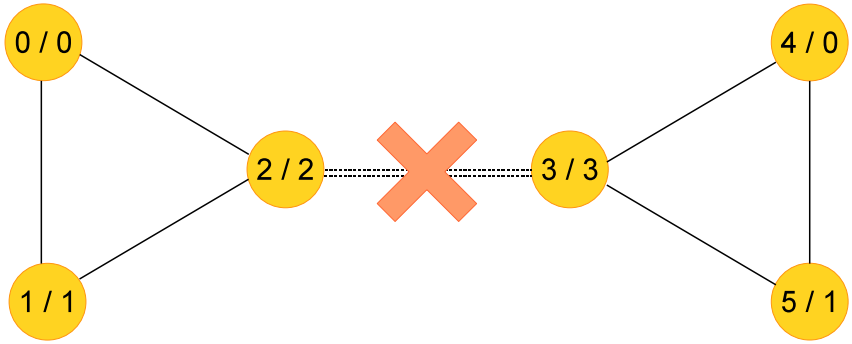


Fig. 1.14 A network with two communities that we analyzed using UPPAAL

g	r	Fast Clock Cycle Length	Slow Clock Cycle Length	CPU Time	Peak Memory Usage
2	0	1	1	Memory Exhausted	
2	0	99	100	457.917 s	2,404,956 KB
2	1	99	100	445.148 s	2,418,032 KB
3	0	99	100	416.796 s	2,302,548 KB
3	2	1	1	Memory Exhausted	
3	2	99	100	22.105 s	83,476 KB
3	2	451	452	798.121 s	3,859,104 KB
3	2	452	453	Memory Exhausted	
4	0	99	100	424.935 s	2,323,004 KB
4	1	99	100	464.503 s	2,462,176 KB
4	2	99	100	420.742 s	2,323,952 KB

Table 1.3 Model checking experiments of a network with two communities

We still need to explore how realistic our counterexamples are. We believe that network topologies with multiple communities occur in many WSN applications. Nevertheless, in practice the gMAC protocol appears to perform quite well for static networks. It might be that problems do not occur so often in practice due to the probabilistic distributions of clock drift and jitter.

1.5 Conclusions

We presented a detailled UPPAAL model of relevant parts of the clock synchronization algorithm that is currently being used in a wireless sensor network that has been developed by Chess [14, 20]. The final model that we presented here may look simple, but the road towards this model was long and we passed through numerous intermediate versions on the way. Using UPPAAL, we established that in certain cases a static, fully synchronized network may eventually become unsynchronized

if the current Median algorithm is used, even in a setting with infinitesimal clock drifts.

In [8], we proposed a slight variation of the gMAC algorithm that does not have the correctness problems of the Median algorithm. However, our algorithm may be of practical interest (although certainly some changes are required). Assegei [1] proposed and simulated three alternative algorithms, to be used instead of the Median algorithm, in order to achieve decentralized, stable and energy-efficient synchronization of the Chess gMAC protocol. It should be easy to construct UPPAAL models for Assegei's algorithms: basically, we only have to modify the definition of the `compute_phase_correction` function. Recently, Pussente & Barbosa [13], also proposed a very interesting new clock synchronization algorithm — in a somewhat different setting — that achieves an $O(1)$ worst-case skew between the logical clocks of neighbors. Much additional research is required to analyze correctness and performance of these algorithms in the realistic settings of Chess with large networks, message loss, and network topologies that change dynamically. Starting from our current UPPAAL model, it should be relatively easy to construct models for the alternative synchronization algorithms in order to explore their properties.

Analysis of clock synchronization algorithms for wireless sensor networks is an extremely challenging area for quantitative formal methods. One challenge is to come up with the right abstractions that will allow us to verify larger instances of our model. Another challenge is to make more detailled (probabilistic) models of radio communication and to apply probabilistic model checkers and specification tools such as PRISM [9] and CaVi [5].

Several other recent papers report on the application of UPPAAL for the analysis of protocols for wireless sensor networks, see e.g. [6, 5, 18, 8]. In [21], UPPAAL is also used to automatically test the power consumption of wireless sensor networks. Our paper confirms the conclusions of [6, 18]: despite the small number of nodes that can be analyzed, model checking provides valuable insight in the behavior of protocols for wireless sensor networks, insight that is complementary to what can be learned through the application of simulation and testing.

In order to keep up with the design team, it is essential to have frequent meetings between the modeler and the designers. Preferably, the modeler should be part of the design team and work at the same location.

Acknowledgements We are most grateful to Frits van der Wateren for his patient explanations of the subtleties of the gMAC protocol. We thank Hernán Baró Graf for spotting a mistake in an earlier version of our model, and Mark Timmer for pointing us to the literature on communities in networks. Finally, we thank the anonymous reviewers for their comments.

References

1. F.A. Assegei. Decentralized frame synchronization of a TDMA-based wireless sensor network. Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering, 2008.

2. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
3. J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer-Verlag, July/August 1996.
4. R. Fan and N.A. Lynch. Gradient clock synchronization. *Distributed Computing*, 18(4):255–266, 2006.
5. A. Fehnker, M. Fruth, and A. McIver. Graphical modelling for simulation and formal analysis of wireless network protocols. In M. Butler, C.B. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2009.
6. A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the lmac protocol for wireless sensor networks. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, pages 253–272. Springer, 2007.
7. K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, pages 2–13. IEEE Computer Society, 1997.
8. F. Heidarian, J. Schmaltz, and F.W. Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. In A. Cavalcanti and D. Dams, editors, *Proceedings 16th International Symposium of Formal Methods (FM2009), Eindhoven, the Netherlands, November 2-6, 2009*, volume 5850 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2009.
9. M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*, pages 322–323. IEEE Computer Society, 2004.
10. C. Lenzen, T. Locher, and R. Wattenhofer. Clock synchronization with bounded global and local skew. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 509–518. IEEE Computer Society, 2008.
11. L. Meier and L. Thiele. Gradient clock synchronization in sensor networks. Technical Report 219, Computer Engineering and Networks Laboratory, ETH Zurich, 2005.
12. M.E.J. Newman. Detecting community structure in networks. *The European Physical Journal B*, 38:321–330, 2004.
13. R.M. Pussente and V.C. Barbosa. An algorithm for clock synchronization with the gradient property in sensor networks. *Journal of Parallel and Distributed Computing*, 69(3):261 – 265, 2009.
14. QUASIMODO. Preliminary description of case studies, January 2009. Deliverable 5.2 from the FP7 ICT STREP project 214755 (QUASIMODO).
15. M. Schuts, F. Zhu, F. Heidarian, and F.W. Vaandrager. Modelling clock synchronization in the Chess gMAC WSN protocol. In S. Andova et.al, editor, *Proceedings Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09)*, volume 13 of *Electronic Proceedings in Theoretical Computer Science*, pages 41–54, 2009.
16. B. Sundararaman, U. Buy, and A. D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281 – 323, 2005.
17. R. Tjoa, K.L. Chee, P.K. Sivaprasad, S.V. Rao, and J.G Lim. Clock drift reduction for relative time slot tdma-based sensor networks. In *Proceedings of the* 15^{th} *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC2004)*, pages 1042–1047, September 2004.

18. S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of qos properties of biomedical sensor networks. In L. de Alfaro and J. Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 69–78. ACM, 2008.
19. F.W. Vaandrager and A.L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing Journal*, 18(4):433–458, December 2006.
20. F. van der Wateren. Personal communication, April 2009.
21. M. Woehrle, K. Lampka, and L. Thiele. Exploiting timed automata for conformance testing of power measurements. In J. Ouaknine and F. W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, volume 5813 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2009.

Design of safe real-time embedded systems in UPPAAL

Bert Bos, Teun van Kuppeveld, Marcel Verhoef and Jiansheng Xing

Abstract In this chapter we discuss a case study inspired on an example from industrial practice, the development of a self-balancing scooter. It has been carried out to evaluate the use of timed automata, whereby the design of the discrete control software, including fault handling, was modelled using UPPAAL. The case study taught us that the specification could be made more precise due to the development of a UPPAAL model of the system behaviour and that the resulting implementation worked "first time right', according to what was specified in the model. The case study showed that a supposedly simple system unveils its complexity already at this level of abstraction and the model was needed to simplify this structure.

1 Introduction

System specifications are generally written as text documents, supported by figures to show the desired system behaviour. It appears that many behavioural questions remain unasked and lead to modifications during test and integration phase. What we demonstrate is that modelling such requirements unveils issues that were not foreseen in the specifications, but are important for the user behaviour and to clarify the structure of the solution. Note that many time projects initially focus on the main and desired system behaviour and while adding more and more detail into the system description, incorporate system behaviour related to start-up, shutdown and reaction to unsafe situations is added later and informally. However Start-up, shutdown, and safety reactions have a high impact on the user perception of the

Bert Bos, Teun van Kuppeveld, Marcel Verhoef
Chess Embedded Technology, Lichtfabriekplein 1, 2031 TE Haarlem, The Netherlands, e-mail: Bert.Bos@chess.nl, e-mail: Teun.van.Kuppeveld@chess.nl and e-mail: Marcel.Verhoef@chess.nl

Jiansheng Xing
University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, To Be Written, The Netherlands, e-mail: xingj@ewi.utwente.nl

system, as we will illustrate in our example. We will also show that preparing a state model of the system behaviour helps to structure and complete the system behavioural descriptions, and could be done as part of the system level design. This work focuses on system level design, in particular on the interaction between user and system. We intend to model the system behaviour for both nominal and for non-nominal situations.

1.1 Approach followed

We took the self-balancing scooter as a system to experiment with, as this was used before in a case study about heterogeneous design [1, 2]. The result of that study was a working self-balancing scooter [3] such that the real-time control for forward and backward movement worked properly. This is the system behaviour as seen from the point of view of control engineering. However it is , without proper switch-on / switch-off functions, left/right steering or any form of safety handling, which shows as system behaviour to the user. We modelled this system behaviour in Uppaal [4] in order to define and to verify the behaviour.

2 Self-balancing scooter – case study description

The system under study is a self-balancing scooter. In our case we assume a self-balancing scooter with the following additional technical properties:

1. For the implementation of the system control a distributed control platform was foreseen, i.e. that the control electronics is divided over two identical parts, one for each wheel. Each unit performs real-time / continuous time control for one wheel individually. The system behaviour, being distributed over two controllers introduces some interesting complications, for instance controllers need to agree on when to start or stop powering the wheels.
2. The self-balancing scooter is equipped with one power supply, one sensor platform, one on/off switch and one steering lever. The sensor platform exists of a steering joystick, an acceleration sensor and a gyroscope. Both control electronics need to be fed by the voltage supply and sensor platform.
3. The control electronics should be based on a FPGA. This is not essential, but is useful to know.
4. Ideally, the system should not be switched on or off by the user, but switches on and off ”automatically”. In this paper we simplified this an on/off switch controlled by the user.
5. The system should not harm the user. Safety measures must be foreseen. The motors may be powered when all conditions are safe and need to be unpowered when an unsafe situation is detected.

The following figure shows a simplified overview of the mechanical system.

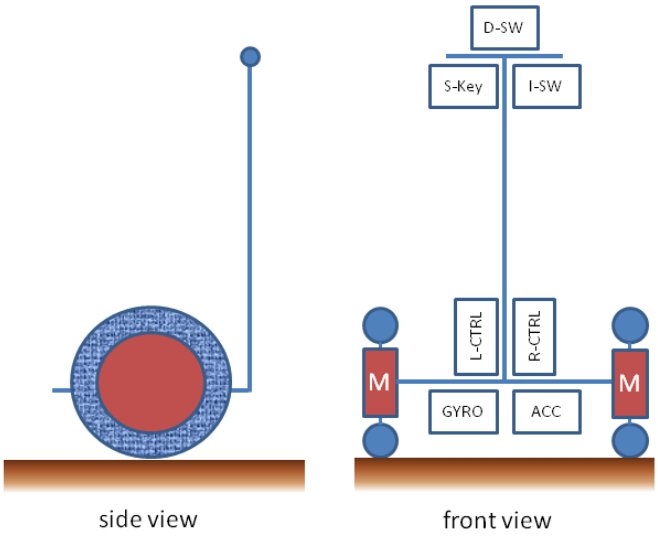


Fig. 1 Synoptic view of the self-balancing scooter

2.1 Informal description of the system behavior

The system behaviour is defined in three use cases:

1. A user scenario to describe the use of the scooter from start of a ride to the finish.
2. A safety scenario, to identify an unsafe situation, react on it and, if possible, how to recover from it.
3. Distributed control, to describe process in the two identical controllers.

2.1.1 User scenario

The ultimate goal is to treat the self-balancing scooter as a classical bicycle, i.e. grab it from the storage, the vehicle detects by itself that the user intends to ride on it so, power the electronics and when safe, start balancing by powering the motors. In the current intermediate step we ask the user to toggle a switch to power the controller electronics. The user holds the vehicle vertical for some time after which the electronic controller powers the motors and the user can step on the vehicle for a ride. To stop using the vehicle, the user simply steps off and switches off the electronics. To control the motion of the vehicle the user leans forward or backward

to move in the forward or backward direction. The user operates a left/right joystick for the left/right rotation.

2.1.2 Safety scenario

Safety is modelled fairly binary: safe or un-safe. Several parameters may lead to an un-safe situation, for instance motor current too high, lean angle too large, battery charge too low, safety key pulled out and data communication errors. We decided to monitor these parameters, and signal an alarm if the pre-set threshold is violated. The handing for all alarms is then identical: immediately stop powering both motors, so that they are in free running mode. Note that breaking the motors in case of the self-balancing scooter is not a good idea, as you would hit the floor harshly. After an alarm occurred and the motors are unpowered, the system should return to a state from where it may recover after the cause of the error has disappeared.. The safety scenario deals with signaling an alarm, react on it and recover from that alarm. During this process new alarms may pop up and must be dealt with.

2.1.3 Distributed control

As stated before, each wheel has its own control electronics. Both controllers receive sensor data and determine individually the signal (power) to their motor. The real-time control is handled autonomously by each of the two controllers. Also the power-on, power-down and (un-) powering the motors are handled autonomously by each of the controllers. However, the controllers need to interact to synchronise their actions: both controller start to (un-) power their motor simultaneously unpower the motors simultaneously in case of an error. Simultaneously means within a limited number of clock cycles. The use case is as follows. A controller powers up after the on/off switch was set to on. After power up it checks whether it is safe to start to power the wheel. In case that it is safe to start it informs the other controller about this and waits for the complementary signal from the other controller. . After both controllers are ready to power the motor, they will do so and start to run the real-time control. When a controller detects an error, it will stop powering its motor and inform the other controller. Note that detection of an error includes receiving an error signal from the other controller..

2.2 Electronics setup

A block diagram of the electronics is shown in figure 2. It consists of two, three phase, Brush-less DC motors of which the angular position is determined by Hall sensors. Each of the motors is powered through an H-bridge, which serves as a power amplifier between the FPGA and the motor. The FPGA board (FPGA = Field

Programmable Gate Array) is the programmable logic that runs the commutation, the real time control and the system behaviour logic. The FPGA also reads the sensor and input data and forwards this to the other FPGA board such that both controllers operate based on the same information. The FPGA boards are connected by a parallel bus interface with error detection.

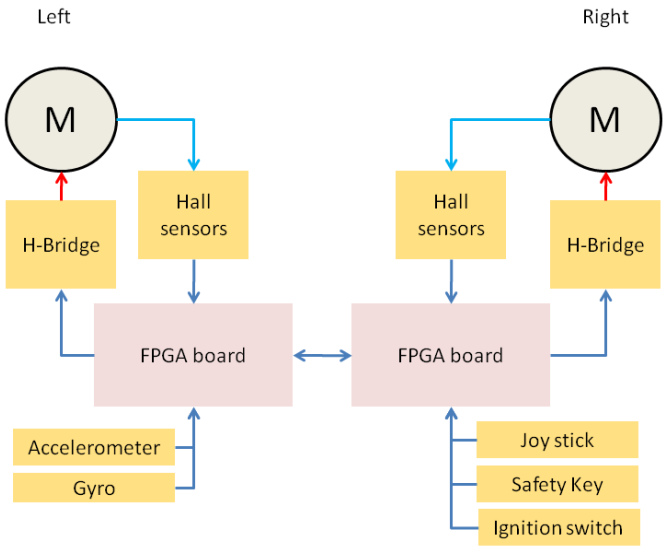


Fig. 2 Overview of the self-balancing scooter electronics architecture

3 UPPAAL model of the self-balancing scooter case study

The Uppaal model evolved several times and was simplified over time, taking away unnecessary processes, moved to focus on distributed system control and makes a clear distinction between the environment and the to be developed control software.

Note that the system design involves a multidisciplinary view regarding the electronics, the controller design, and system behaviour .All aspects may be changed to optimise for the desired system behaviour. After having developed and implemented an untimed Uppaal model, we completed a timed automata model in Uppaal.

The following paragraphs give a description of the Uppaal model and verification results of the timed Uppaal model.

3.1 The environment model

The environment model models that parts that interface with the controller software. These are a user, hardware peripherals and failures.

3.1.1 User

The user process models the behaviour of the user during nominal behaviour and it triggers abnormal behaviour. With *Ignition*! the user toggles the discrete control from IDLE state to CHECK state or from any of the discrete control states back to IDLE. After switching on, the user must hold the vehicle upright to let the controller power the motors. The transition $angle = 0$ is used to simulate holding the self-balancing scooter upright.

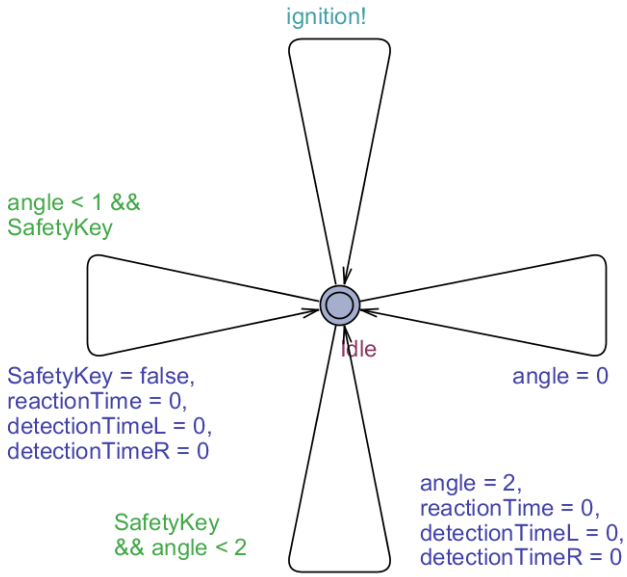


Fig. 3 UPPAAL user model

The user may cause an accident and fall, in which case the vehicle angle becomes too large. This is modelled with the transition $angle = 2$ setting the angle to a larger angle, for example $0.2\,rad$. Similarly, when the driver falls off he/she will unplug the Safety Key. The person, holding the vehicle pulls out the safety key, using the transition $SafetyKey = \texttt{false}$. Both the transitions that simulate en error set a global clock to 0, to measure the time needed between causing an error and the system reaching a safe mode. Either one of the error transitions fire, meaning a single failure is modelled.

3.1.2 Safety switch

The safety switch is the electronics that closes and opens the power lines to the motor. The controller for start-up and shutdown normally does the operation of this switch. In case of emergency, the safety monitor process overrules the controller. The idea is that in case of an emergency, the vehicle control should go to the safest situation. Instead of locking (braking) we have chosen to let the wheels rotate freely. That means that the power lines to the motor will be switched to open loop. The safety switch process models the closing and opening of the switch. In the nominal behaviour, the discrete control opens and closes this switch with the *open*? and *close*? events. The safety monitor can override normal control and open the switch, with the error? event. A Boolean signal enable telling other parts (the controller) whether the safety switch is in the OPEN or in the CLOSE state. A timer *openDelay* models the time needed to open the safety switch.

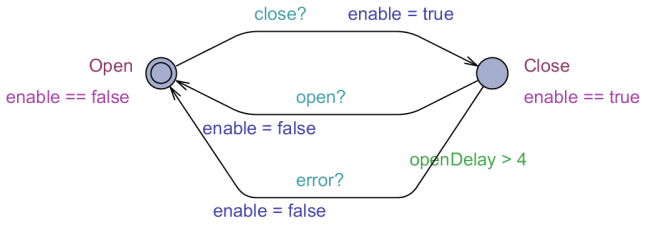


Fig. 4 UPPAAL safety switch model

3.2 The system model

The system model models the desired system behaviour. It consists of two processes: a controller and a safety monitor. These are two independent processes, while in particular the safety monitor should not be hindered by the control process. Also the hardware on which the safety monitor will be executed runs independent from the other electronics and will continue to operate even when the control

3.2.1 The controller

The Controller automaton models the software that controls one motor of the vehicle. It starts from the situation that the main power switch has been switched on and the electronics is powered. The IDLE state is the initial state from where the control starts. The user toggles with the ignition switch (*ignition*? event) to start the use of the self balancing scooter. The controller gets into the CHECK state to verify whether all signals are safe (the safety monitor process condition not unsafe and the

vehicle must be hold upright by the user (sbsAngle ¡= 1). If the conditions are met then the state may progress to the state SYNCHRONISE. In this state, both left and right controller signal that they are in this state and wait until this signal from the other side, to simultaneously progress to the DRIVE state. This last transition closes the safety switch and thus powers the motor.

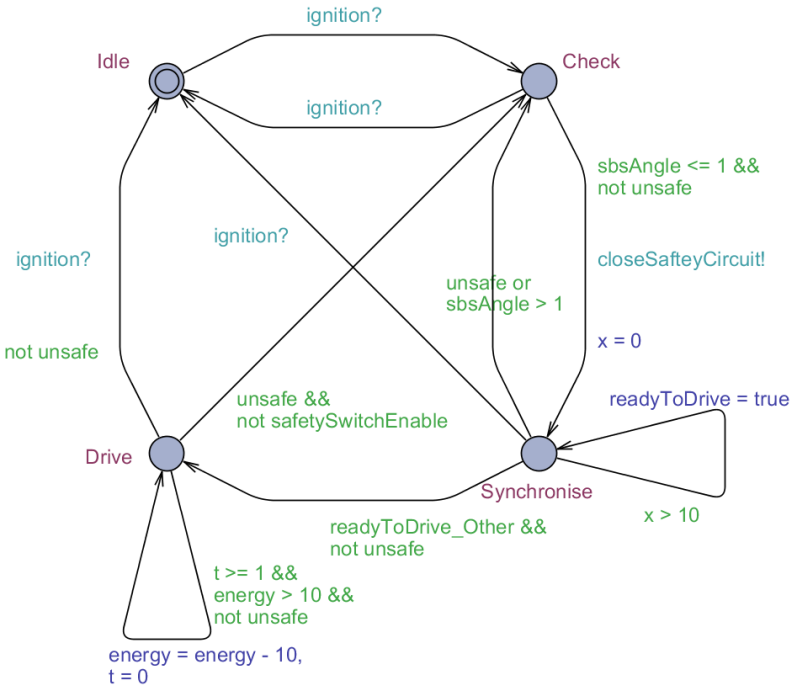


Fig. 5 UPPAAL controller model

The controller not in IDLE state should return to the CHECK state when unsafe becomes true. The user must be able to toggle the ignition switch in each of the states, which makes the controller to return to the IDLE state. This ignition has been blocked for the transition from DRIVE to IDLE in case of an error to give preference to the state change from DRIVE to CHECK in case of an error. A transition was added to the DRIVE state to lower the battery level every 1 time unit. This battery reduction could have been modelled in other states as well, but as the motors are the main power consuming devices this was simplified to model the power consumption in the DRIVE state only.

3.2.2 The safety monitor

The safety monitor monitors parameters that determine the safe/unsafe state of the vehicle. In case that an unsafe situation occurs, the safety motor changes from safe to unsafe and opens the safety switch. This idle state is unsafe. The only initial activity that can take place is to verify all inputs and bring the safety monitor to a safe state. This will allow closing the safety switch.

The safety monitor automaton simulates the software and hardware that monitors the safety related parameters and in case of an out of bounds of a variable or an unplugged Safety Key or an error detected on the other side. It will force the safety switch to open on its transition from SAFE to UNSAFE.

Note that the monitored variables are modelled as signals not as events. For an event to take place both sides, the sender and the receiver, must be available to pass the transition. In physics the event of a Safety Key being pulled out or an angle out of bounds will not wait for the state machine to allow this error to happen; it just happens.

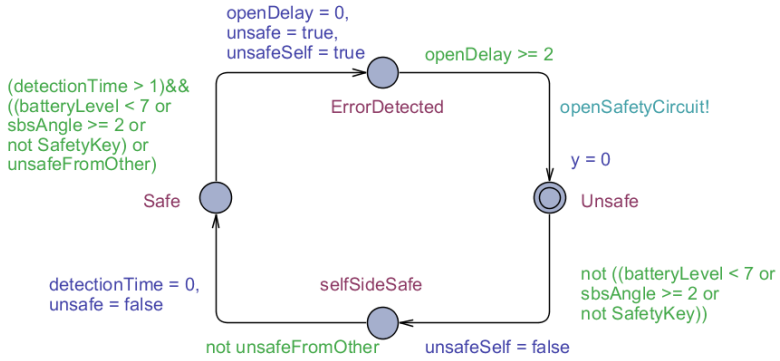


Fig. 6 UPPAAL safety monitor model

After detection of an error moves the safety monitor to an intermediate state, from which it progresses with opening the safety circuit with an event (openSafety-Circuit!) The safety monitor ends in the UNSAFE state. After the unsafe situation disappears, the safety monitor returns to the SAFE state via an intermediate state to wait for the other side to be safe as well. The safety monitor will not close the safety circuit, but only enable the discrete control to close the safety circuit.

3.3 Putting it all together

The system is assembled from the processes that were described in the previous paragraphs. The model describes the combination of environment and discrete control. The processes are given in table 1. It shows that the user and the ignition switch are common for left and right wheel, while the safety switch, safety monitor, control occur twice.

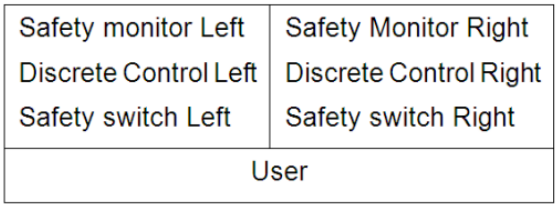


Fig. 7 UPPAAL controller model

3.4 Model verification

A first set of properties were verified from initial conditions that are related to an idle vehicle to the state transitions from the IDLE state via the ignition transition until the DRIVE state for both controllers.

- The controller should be free from deadlocks (A[] not deadlock)
- In case one of the controllers is not in the DRIVE state, the other should not stay in the DRIVE state (A¡¿ not DC_R.Drive imply not DC_L.Drive).
- There exists a path such that both controllers are in the DRIVE state (E¡¿ DC_L.Drive && DC_R.Drive)
- The controller stops at the energylevel = 10. That is A[] energylevel ¿= 10 passes, while A[] energylevel ¿ 10 fails. This shows that the battery will reach a minimum level and the controller stops at the prescribed level.

Error handling is crucial for this system and was the reason to start modelling the system formally. Most important is the case where the vehicle is in the DRIVE state and an error occurs. Therefore the model was initialised in the DRIVE state for both controllers and the user triggers an error with pulling the Safety Key.

E¡¿ (SafwtyKey==false && DC_L.Check && DC_R.Check) && faultReaction ¡ 6

- That the time between detecting an error and being into CHECKING sate is small, i.e. the only actions that the timed automata take is the transition to the CHECKING state. for both sides..
- To prove that as soon as the safety monitor detects an error, it will open the safety switch. For both wheels within the minimum time needed for this.

4 Implementation

The implementation is closely related to the untimed Uppaal model [5].

In the implementation, the Safety Key is a strap around the drivers wrist, ending in a connector bridge. Falling off the vehicle will remove the bridge and open a signal loop, which gives the Safety Key disable signal to the safety monitor.

The sensors were, for practical reasons connected to the left side controller and the readings were forwarded from left to right. To be certain about the integrity of the signals on the right hand side a CRC check was added and a time out was used to know that sensor readings were received on the right hand side. Both the CRC error and the missing sensor reading for one time slot trigger an error to the safety monitor.

5 Conclusions

In the Chess case study with the self balancing scooter we learned the following:

- Simply drawing up an Uppaal model as part of the system design already helps to complete the behavioural description at system level or at least shows hidden complexity.
- Running the simulations and formally verifying the model shows specification errors very early in the development process and avoids finding these mistakes during test and integration.
- The model forces the developer to think about the system behaviour and about the software structure. A complex s model is fairly quickly changed to find a simpler one. The modelling exercise and explaining the model to others forced us to clearly define the states for the discrete control and safety monitor.
- The model helps to specify the system behaviour, in particular switch on / switch off and safety handling and recovery. These parts of system behaviour are often postponed to the design and implementation phase and by that time not well designed due to time constraints. Also these aspects are then designed at engineering level, while they are crucial for the user experience. Structural changes to simplify the the software structure are avoided.
- The setup is also such that it can cope with more than one error the time, or react properly on an additional error while handling the current ones. This is manly due to the independent detection of errors, proceed to run the controller after all error

have disappeared, and due to the somewhat rigid, simple strategy of reacting to all errors in the same manner.
- The system level design of the electronics, such as communication between the FPGA boards was partly determined by the information needed in the Uppaal model, i.e. the Uoaal model leaned us which information was minimally needed to make the distributed contro possible.
- The modeller needs a high level of abstract thinking to imagine the system behaviour as timed automata.

6 References

[1] T. Van Kuppeveld, Model-based redesign of a self-balancing Scooter (University Twente, August 2007). [2] A.M., Bos, Modelling multibody systems in terms of bond graphs, with application to a motorcycle, (Ph.D. thesis, University Twente, The Netherlands, ISBN 90-9001442-X, 1986) [3] J. Simons, Hardware ontwikkeling en besturing voor de step (Hogeschool Amsterdam, 2007). [4] G. Behrmann, A. David, and K. G. Larsen, A tutorial on Uppaal (17th November 2004) [5] R.P., Verstraten, Self balancing scooter (December 15, 2009)

To Be Added.

An Introduction to Schedulability Analysis Using Timed Automata

Alexandre David Kim G. Larsen Arne Skou

1 Introduction

Embedded systems involve the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource constrained manner in terms of for example memory, processing power, bandwidth, energy consumption, as well as timing behavior.

Viewing the application as a collection of (interdependent tasks) various *scheduling principles* may be applied to coordinate the execution of tasks in order to ensure orderly and efficient usage of resources. Based on the physical process to be controlled, timing deadlines may be required for the individual tasks as well as the overall system. The challenge of *schedulability analysis* is now concerned with guaranteeing that the applied scheduling principle(s) ensure that the timining deadlines are met.

For schedulability analysis of single processor systems, a number of industrially applied tools exists benefiting from great succes in real-time scheduling theories; results intiated in the 1970ies and the 1980ies, and by now well-established. However these theories and tools have become seriously challenged by the rapid increase in the use of multi-cores and multiprocessor system-on-chips (MPSoC).

In this chapter we will present a framework for schedulability analysis based on timed automata which overcome the limitation to single-processor architectures, while providing absolute guarantees: if after model checking no violations of deadlines have been found, then it is guaranteed that no violations will occur during execution. In this approach, the (multiprocessor) execution platform, the tasks, the interdependencies between tasks, their execution times, and mapping to the platform are modeled as timed automata [1] allowing efficient tools such as Uppaal [7] to *verify* schedulability using model checking.

The chapter offers a Uppaal modeling framework (download from [4]), that may be instantiated to suit a variety of scheduling scenarios, and that can be easily extended. In particular, the framework includes:

- A rich collection of attributes for tasks, including: off-set, best and worst case execution times, minimum and maximum interarrival time, deadlines, and task priorities.
- Task dependencies.

- Assignment of resources, for example processors or busses, to tasks.
- Scheduling policies including First-In First-Out (FIFO), Earliest Deadline First (EDF), and Fixed Priority Scheduling (FPS).
- Possible preemption of resources.

The combination of task dependencies, execution time uncertainty and preemption makes schedulability of the above framework undecidable [6]. However, the recent support for stopwatch automata [3] in UPPAAL leads to an efficient approximate analysis that has proved adequate on several concrete instances as demonstrated in [5].

The outline of the remainder of the chapter is as follows: in Section 2 we give an introduction to the types of schedulability problems considered, in Section 3 we provide provide the UPPAAL modeling framework. Following this we show in Section **??** how to instantiate the framework for a number of different schedulability problems by way of an example system. Finally, we conclude the paper in Section 4.

2 Schedulability Problems

At the core of any schedulability problem are the notions of tasks and resources. Tasks are jobs that require the usage of resources for a given duration after which tasks are considered done/completed. The added constraints to this basic setup is what defines a specific schedulability problem. In this section, we define a range of classical schedulability problems.

2.1 Tasks

A schedulability problem always consists of a finite set of tasks that we consistently will refer to as $T = \{t_1, t_2, ..., t_n\}$. Each task t_i has a number of attributes that we refer to by the following functions:

- INITIAL_OFFSET: $T \to \mathbb{N}$ — Time offset for initial release of task.
- BCET: $T \to \mathbb{N}_{\geq 0}$ — Best case execution time of task.
- WCET: $T \to \mathbb{N}_{\geq 0}$ — Worst case execution time of task.
- MIN_PERIOD: $T \to \mathbb{N}$ — Minimum time between task releases.
- MAX_PERIOD: $T \to \mathbb{N}$ — Maximum time between task releases.
- OFFSET — The time offset into every period, before the task is released.
- DEADLINE: $T \to \mathbb{N}_{\geq 0}$ — The number of time units within which a task must finish execution after its release. Often, the deadline coincides with the period.

- PRIORITY — Task priority.

These attributes are subject to the obvious constraints that BCET(t) $\leq$ WCET (t) $\leq$ DEADLINE(t) $\leq$ MIN_PERIOD(t) $\leq$ MAX_PERIOD(t). The periods are ignored if the task is non-periodic.

The interpretation of these attributes is that a given task t_i cannot execute for the first OFFSET (t_i) time units and should hereafter execute exactly once in every period of PERIOD (t_i) time units. Each such execution has a duration in the interval [BCET (t_i),WCET (t_i)]. The reason why tasks have a duration interval instead of a specific duration is that tasks are often complex operations that need to be executed and the specific computation of a task depends on conditionals, loops, etc. and can vary between invocations. Furthermore, for multi-processor scheduling, considering only worst-case execution times are not is insufficient as deadline violations can result from certain tasks exhibiting best-case behavior.

We say that a task t is *ready* (to execute) at time τ iff:

1. $\tau \geq$INITIAL_OFFSET(t)
2. t has not executed in the given period dictated by τ.
3. All other constraints on t are satisfied. See 2.2 for a discussion on task constraints.

2.2 Task Dependencies

Task execution is often not just constrained by periods, but also by interdependencies among tasks., for example because one task requires data that is computed by other tasks. Such dependencies among a set of tasks $T = \{t_1, t_2, ..., t_n\}$ are modelled as a directed acyclic graph (V, E) where tasks are nodes (i.e., $V = T$) and dependencies are directed edges between nodes. That is, and edge $(t_i, t_j) \in E$ from task t_i to task t_j indicates that task t_j cannot begin execution until t_i has completed execution.

2.3 Resources

Resources are the elements that execute tasks. Each resource uses a scheduler to determine which task gets executed on a given resource at any point in time. Resources are limited by only allowing the execution of a single task at any given time.

Tasks are *a priori* assigned to resources. For a set of resources $R = \{r_1, ..., r_k\}$ and a set of tasks $T = \{t_1, ..., t_n\}$, we capture with the function ASSIGN : $T \rightarrow R$.

In a real-time system, resources function as different types of processors, communication busses, etc. Combined with task graphs we can use tasks and resources to emulate complex systems with such task interdependency on different processors. For example, if we want to model two tasks t_i and t_j with dependency $t_i \rightarrow t_j$, but the tasks are executed on different processors and t_j

needs the results of t_i to be communication across a data bus, we introduce an auxiliary task t_{ic} that requires the bus resource and update the dependencies to $t_i \rightarrow t_{ic} \rightarrow t_j$. We illustrate this concept in ... **??**

Scheduling Policies In order for a resource to determine which task to execute and which tasks to hold, a resource applies a certain scheduling policy implemented in a scheduler. Scheduling strategies vary greatly in complexity depending on the constraints of the schedulability problem. In this section we discuss a subset of scheduling policies for which we have included models in our scheduling framework.

- **First-In First-Out (FIFO)** Ready tasks are added to a queue in the order they become ready.
- **Earliest Deadline first (EDF)** Ready tasks are added to a sorted list and executed in the order of earliest deadline.
- **Fixed Priority Scheduling (FPS)** Each task is given an extra attribute, PRIORITY, and ready tasks are executed according to the highest priority.

Schedulers operate in such a manner that resources are never idle while there are ready tasks assigned to them. That is, as soon a task has finished execution a new task is set for execution.

Preemption Resources come in two shapes, preemptive and non-preemptive. A non-preemptive resource means that once a task has been assigned to execute on a given resource, that task will run until completion before another task is assigned to the resource. Preemption means that a task assigned to a resource can be temporarily halted from execution if the scheduler decides to assign another task to the resource. We say that the first task has been preempted. A preempted task can later resume execution for the remainder of its duration.

Preemption allows for greater responsiveness to tasks that are close to missing their deadline, but that flexibility is on behalf of increased complexity of the schedulability analysis. The framework we define in the following section will include a model for schedulability analysis with preemption.

2.4 Schedulability

Now, we define what it means for a system to be schedulable. A system of tasks with constraints and resources with scheduling policies is said to be schedulable if no execution satisfying the constraints of the system violates a deadline.

3 Framework Model in UPPAAL

In this section, we describe our UPPAAL framework for analyzing the scheduling problems defined in Section 2. The framework is constructed such that a model

of a particular scheduling problem consists of two different timed automata templates: A generic task template and a scheduler. The scheduler has an internal queue that is used to implement different types of scheduling policies. We describe the templates in this order.

3.1 Modeling Idea

In our scheduling model that is based on [2], we have a number of tasks modeled by a *Task* template and one or more schedulers modeled by a *Scheduler* template. The idea here is that we need one scheduler per process and the scheduler decides which task to run.

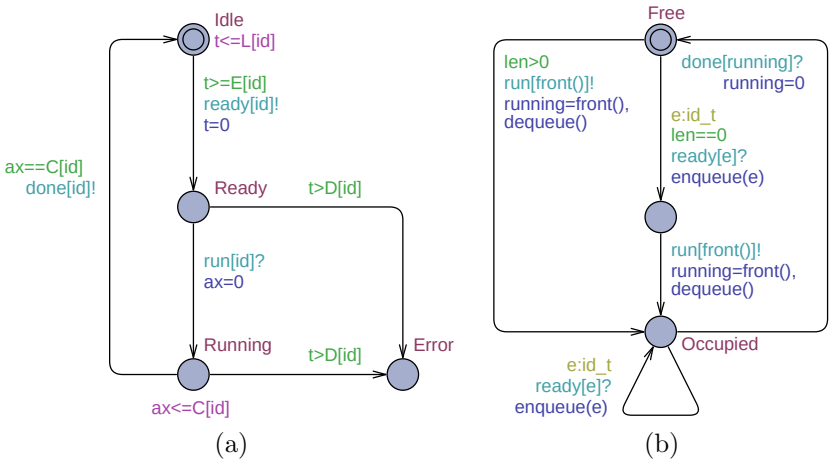


Figure 1: Task (a) and scheduler (b) templates.

The model depicted in Figure 1 naturally divides the scheduling problems in to tasks that are controlled by a scheduler. Each task has its own `id`. In addition the model defines global tables for the release time, computation time, and deadline of every task. In the task template (a) a task `id` is released between `L[id]` and `E[id]`, its computation time is `C[id]`, and its deadline `D[id]`.

As a first step, we model non-preemptive tasks. Initially a task is `Idle` and when it is ready to execute, it signals itself to the scheduler using the channel synchronization `ready[id]!`. Then it waits to be run by the scheduler, which is done by the synchronization `run[id]?`. Finally, when it is done it signals itself to the scheduler with `done[id]!`.

The scheduler keeps a queue of ready tasks internally. The queue is manipulated with the function calls `enqueue(e)` (to enqueue a task `e`) and `dequeue()`. Tasks are read with `front()`. The scheduling policy is implemented in the enqueuing function. The scheduler starts in `Free`. When it receives a ready task with `ready[e]?` (for some `e`), it goes to an intermediate state after queuing the task. Then runs it with `run[front()]!`, and it ends in the `Occupied` state. The task is dequeued from the queue to avoid interference from other tasks that

Listing 1: Type and constant array definitions.

```
1  typedef int [0, N−1] id_t
2
3  const int E[N] = { 200, 200, 100, 100 };
4  const int L[N] = { 400, 200, 200, 100 };
5  const int D[N] = { 400, 200, 200, 100 };
6  const int C[N] = {  50,  40,  20,  10 };
7  const int P[N] = {   0,   1,   2,   3 };
```

will arrive and be queued while it is executing. The scheduler keeps track of the current running task. If more tasks are incoming while the scheduler is occupied, they are queued. The scheduler is signaled with `done[running]?` once the task has finished its execution.

With this model, schedulability can be verified with the following CTL query:

$$\text{A[] forall(i : id\_id) not Task(i).Error}$$

That is, is it always the case that on all execution paths no task will ever be in the `Error` location?

This is the basic model we use in our framwork and we show how it can be extended to multi-processor scheduling with resource sharing or preemptive scheduling. Before making these extensions, we introduce some of the data-structures and functions used in the model.

3.2 Data Structures

For a scheduling problem with N tasks $T = t_0, ..., t_{n-1}$ we define the type `id_t` as shown at the top of listing 1. This type is given as argument to the tasks that can use this identifier to access their parameters defined in the constant arrays that follow the type declaration. These arrays define for every task with its identifier id, its ready interval between `E[id]` and `L[id]`, its deadline `D[id]`, its computation time `C[id]`, and its priority `P[id]`.

3.3 Scheduling Policies

In this section we describe how we model fixed priority scheduling (FPS) and how to extend it to other types of policies. The idea is that tasks that become ready signal themselves to the scheduler that enqueues them in its internal task queue. The scheduler sorts its queue according to a policy and picks the tasks from the front to execute them. Optionally it can stop and run them later. This is illustrated in Fig. 2. In our model we have one such queue per scheduler represented as an array of id_t. We note that we could instead have one global array, depending on the modelling goal if several schedulers are used, e.g., to model distributed systems.

The array has an associated length variable to keep track of the number of queued tasks and the policy is implemented in the enqueue function. This

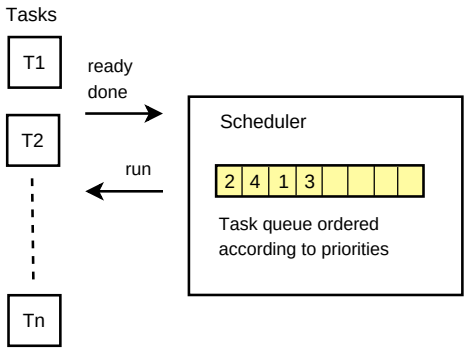


Figure 2: Synchronization between tasks and a scheduler. The scheduler enqueues ready tasks and run the tasks at the front of its internal queue.

function inserts a new task at the right position in the queue, respecting the priority order. Dequeueing consists of removing the front element. Queueing and dequeueing involve shifting the elements in the array. We take particular note on the reset `list[i] = 0` done in dequeueing to keep unused entries at a fixed value (here 0) to avoid unnecessary state-space explosion. Not doing so would mean that the state remembers the history of the queue, which is not needed. The code is given in listing 2.

In addition the scheduler defines the variable `running` to keep track of the current running task. To extend this model to different scheduling policies, one would have to mainly change the enqueueing function.

3.4 Modelling Resource Sharing

To model resource sharing we can extend the model presented in Fig. 1.(a) with an array of booleans to represent if some resource is available or taken. We consider that every task needs one resource to run, which is defined in another array (`need`) as shows in listing 3.

The array `need` specifies which resource is needed. The boolean array `resource` models their availability. The functions are called by the tasks with their identifiers as argument. The function `available` return true if the resource of task `id` is available or false otherwise, the function `take` takes the needed resource, and `release` releases it. We note that the code contains assertions to ensure that the model is consistent, e.g., a resource is taken only if it was available. Figure 3 shows the extension of our previous task template to use these functions. A task is run only if its resource is available, in which case the resource is taken and released when then task completes. This is not very interesting with only one non-preemptive scheduler but we can run two schedulers instead to see the effect. Figure 4 shows Gantt charts of such executions first with one scheduler and then with two schedulers. Tasks 2 and 3 never run at the same time and one is always blocked while the other holds a shared

Listing 2: Functions to manage the task queue and implement the scheduling policy.

```
1  id_t  list [N];
2  int [0, N] len;
3  id_t  running;
4
5  // Enqueue w.r.t. the priority.
6  void enqueue(id_t element)
7  {
8          int tmp;
9          list [len++] = element;
10         if (len > 0)
11         {
12                 int i = len - 1;
13                 while(i > 1 && P[list[i]] > P[list [i-1]])
14                 {
15                         tmp = list[i-1];
16                         list [i-1] = list [i];
17                         list [i] = tmp;
18                         i--;
19                 }
20         }
21 }
22 // Remove the front element of the queue.
23 void dequeue()
24 {
25         int i = 0;
26         len -= 1;
27         while (i < len)
28         {
29                 list [i] = list [i + 1];
30                 i++;
31         }
32         list [i] = 0;
33 }
34 // Return the front element of the queue.
35 id_t front()
36 {
37         return list [0];
38 }
```

Listing 3: Implementation of the shared resource between different tasks.

```
1  const id_t need[N] = { 0, 1, 2, 2 };
2  bool resource[N];
3
4  bool available(id_t id)
5  {
6      return !resource[need[id]];
7  }
8  void take(id_t id)
9  {
10     assert(!resource[need[id]]);
11     resource[need[id]] = true;
12 }
13 void release(id_t id)
14 {
15     assert(resource[need[id]]);
16     resource[need[id]] = false;
17 }
```

resource. The other tasks can run in parallel, provided that one scheduler is available.

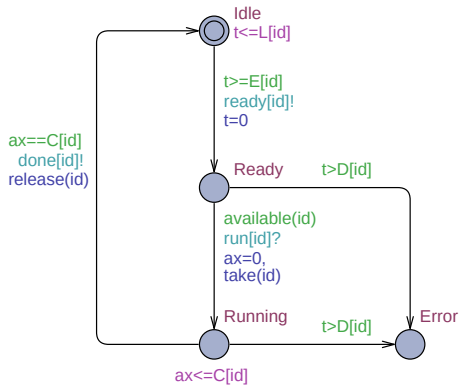


Figure 3: Task template extended with resource sharing.

3.5 Modelling Preemptive Scheduling

In the case of preemptive scheduling we want the scheduler to stop the current executing task and let another task with higher priority run instead. To do that, we can use *stop-watches*. Fig. 5 shows how we extend both the scheduler and the original task model of Fig. 1.

First the scheduler will keep the running task at the front of its queue and

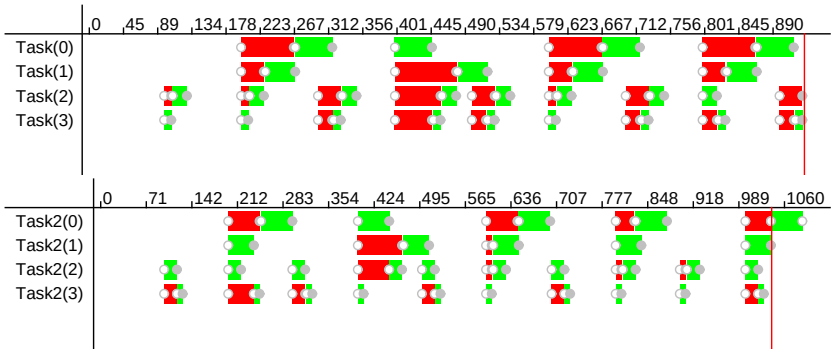


Figure 4: Gantt charts that show the run with one and two schedulers with resource sharing.

remove it when it has terminated. This simplifies the handling of new incoming tasks. When tasks arrive while the scheduler is busy, it immediately stops the running tasks with `stop[running]!` and go back to its previous intermediate state from which it picks the task at the front of the queue to run. Since the newly arrived tasks is enqueued according to its priority, we pick the current highest priority task every time.

Second, on the task model, we use *stop-watches*. When a running task is stopped, it goes to a stopped state with the invariant `ax'==0` that effectively stops the clock that measures the current computation time. From this state, it can resume execution if it gets signaled with `run[id]?` by the scheduler. We add an additional transition to detect if the task violates its deadline while stopped.

4 Conclusion

We have provided a framework allowing the modeling and analysis of a variety of schedulability scenarios. In particular, our framework supports multi-processor systems, rich task models with timing uncertainties in arrival- and execution-times, possible dependencies, a range of scheduling policies, and possible pre-emption of resources.

The support of approximate analysis of stopwatch automata in Uppaal 4.1 is key to the successful schedulability analysis.

Furthermore, the uncertainty on the periods used in our framework could be generalized to more general task arrival where a separate process determines the arrival of tasks. Such situations can be modeled using the structure of our framework by letting the starting of periods be dictated through channel synchronization with the model controlling arrival times. Even with such liberty, the overapproximation is still finite and termination is guaranteed.

The scheduling framework provided in this paper is structured such that adaptation can be made to accommodate other scheduling polices and inter-task

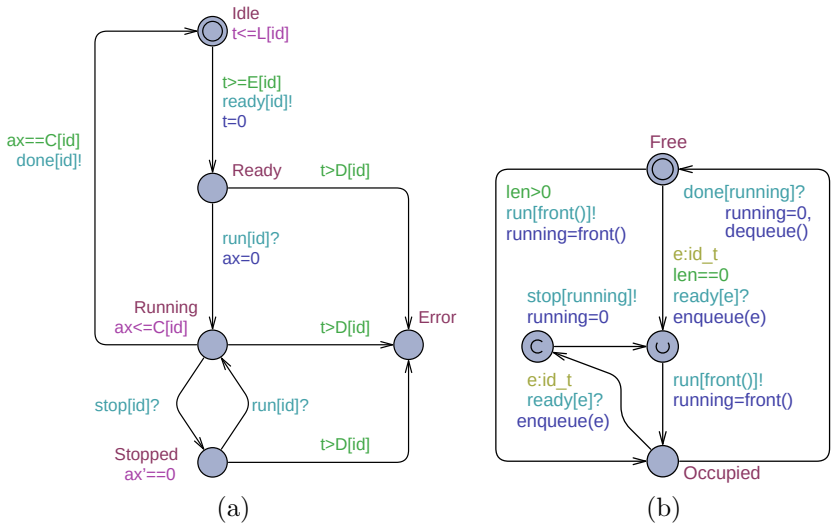


Figure 5: Extensions of the task (a) and scheduler (b) templates to model preemptive scheduling.

constraints. The former can be achieved by adding another policy model similarly to the three built-in policies FIFO, FPS, and EDF. The latter is achieved through the use of the function calls new_period, dependencies_met, and completed.

Acknowledgements

The authors would like to thank Marius Mikučionis for providing the format for listing UPPAAL code.

References

[1] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science (TCS)*, 126(2):183–235, 1994.

[2] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

[3] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidesi, editor, *11th International Conference on Concurrency Theory, (CONCUR'2000)*, number 1877 in Lecture Notes in Computer Science, pages 138–152, University Park, P.A., USA, July 2000. Springer-Verlag.

[4] UPPAAL Scheduling Framework
. http://www.uppaal.com/SchedulingFramework, Jan. 2009.

[5] Aske Brekling Jan Madsen, Michael R. Hansen. A modelling and analysis framework for embedded systems. Chapter 3 in this book.

[6] Pavel Krcál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2004.

[7] UPPAAL. http://www.uppaal.com, Jan. 2005.

Schedulability Analysis of Herschel/Planck Software Using Uppaal

Marius Mikučionis and Kim G. Larsen and Brian Nielsen

Abstract This chapter shows how UPPAAL is applied in schedulability analysis of satellite attitude and orbit control software used in Herschel/Planck mission. Our method transforms the schedulability analysis into reachability analysis performed by UPPAAL. The chapter briefly describes the schedulability requirements and elaborates on the modeling framework designed to handle single processor hardware with a fixed priority preemptive scheduler, detailed task specifications, two resource sharing protocols and voluntary task suspension. The results include qualitative answers (whether the system is schedulable) as well as quantitative (response and blocking time estimates) which are comparable with classical response-time analysis.

Key words: schedulability analysis, timed automata, stop-watch automata, model-checking, verification

1 Introduction

The goal of schedulability analysis is to check whether all tasks finish before their deadline. Traditional approaches like [Burns(1994)] provide generic frameworks which assume worst-case scenario where consecutive response-times are calculated and compared with deadlines. Often, such conservative scenarios are never realized and thus negative results from such analysis may be too pessimistic. The idea is

Marius Mikučionis
Aalborg University, Aalborg, Denmark, e-mail: marius@cs.aau.dk

Kim G. Larsen
Aalborg University, Aalborg, Denmark, e-mail: kgl@cs.aau.dk

Brian Nielsen
Aalborg University, Aalborg, Denmark, e-mail: bnielsen@cs.aau.dk

to base the schedulability analysis on a system model with possibly more details, taking into account specifics of individual tasks. In particular this will allow a safe but far less pessimistic schedulability analysis to be settled using real-time model checking. Moreover, the model-based approach provides a self-contained visual representation of the system with formal, non-ambiguous interpretation, simulation and other possibilities for verification and validation.

Our model-based approach is motivated by and carried out on example applications in a case study of Herschel-Planck satellite system. Compared with classical response-time analysis our model-based approach is found to uniformly provide less pessimistic response-time estimates and allow to conclude schedulability of all tasks, in contrast to negative results obtained from the classical approach.

1.1 The Herschel-Planck Mission

The Herschel-Planck mission consists of two satellites: Herschel and Planck. The satellites have different scientific objectives and thus the sensor and actuator configurations differ, but both satellites share the same computational architecture. The architecture consists of a single processor, a real-time operating system (RTEMS), a basic software layer (BSW) and an application software (ASW).

The goal of the study is to show that ASW tasks and BSW tasks are schedulable on a single processor with no deadline violations. The tasks use preemptive fixed priority scheduler and a mixture of priority ceiling and priority inheritance protocols for resource sharing and extended deadlines (beyond period). In addition, some tasks need to interact with external hardware and effectively suspend their execution for a specified time. Due to suspension, this single-processor system has some similarity to multi-processor systems since parts of activities are executed elsewhere and the classical worst-case response-time analysis (applicable to single-processor systems) is pushed to its limits. One of the results of [Palm(2006)] is that one task may miss its deadline on Herschel (and thus the system is not schedulable) but this violation has never been observed in neither stress testing nor deployment.

Figure 1 shows the parameters which describe each periodic task: period defines how often the task is started, offset – how far into the cycle the task is started (released), deadline is measured from the instance when task is started and worst-case execution time within deadline.

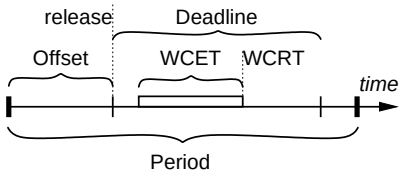


Fig. 1: Task time bounds.

Some tasks access shared resources and those are protected by semaphore locking to ensure exclusive usage. Sometimes tasks use resources repeatedly (locking and unlocking several times). When the resource semaphore is locked, a task may suspend its execution by calling hardware services and waiting for the hardware to finish thus temporarily releasing the processor for other tasks. The processor may be released multiple times during one semaphore lock. In response-time analysis, the processor utilisation is computed by dividing the sum of worst-case execution times by the duration of analysed time window.

Table 1 shows the description of the primary functions task from [Palm(2006)]. The task consists of six activities. Each activity is described by two numbers: CPU time / BSW service time (BSW service time is included in CPU time), followed by resource usage pattern if any. The resource usage is described by the following parameters:

LNS – total number of times the CPU has been released while the resource was locked (task suspension count).
LCS – total time the CPU has been released while the resource was locked (task suspension duration).
LC – total time the resource has been locked.
MaxLC – the longest time the resource has been locked.

For example "Data processing" takes $20577\mu s$ in total, from which it has locked the resource `Icb_R` for $1600\mu s$, and from which CPU has been released (execution suspended) for $1200\mu s$.

Table 1: The sequence of primary functions task from [Palm(2006)].

Primary Functions	
- Data processing	**20577**/2521
	Icb_R(LNS: 2, LCS: **1200**, LC: **1600**, MaxLC: 800)
- Guidance	**3440**/0
- Attitude determination	**3751**/1777
	Sgm_R(LNS: 5, LCS: **121**, LC: **1218**, MaxLC: 236)
- PerformExtraChecks	**42**/0
- SCM controller	**3479**/2096
	PmReq_R(LNS: 4, LCS: **1650**, LC: **3300**, MaxLC: 3300)
- Command RWL	**2752**/85

2 Model-Checking Schedulability Methodology

The main idea is to translate schedulability analysis problem into a reachability problem for timed automata and use the real-time model-checker UPPAAL to check that none of the deadlines are violated, derive worst-case blocking and response-

times and processor utilization. We refer to the previous chapter for UPPAAL concepts.

Figure 2 shows the work-flow of response-time analysis (performed by Terma A/S) and schedulability analysis using UPPAAL: the task timing informations are obtained from ASW and BSW documentation, worst-case execution times (WCET) of BSW are obtained from BSW documentation [Terma A/S(Issue 9)] and ASW timings are obtained from simulation measurements. In addition the UPPAAL model uses information about the individual task flows, i.e. the timing of resource locks, CPU execution and suspension.

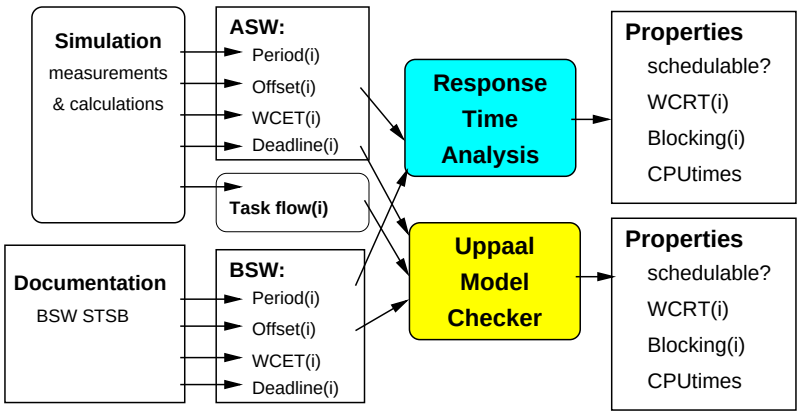


Fig. 2: Work-flow of schedulability analysis.

The UPPAAL framework consists of the following process models: a fixed priority preemptive CPU scheduler, a number of task models, and one process for ensuring global invariants. We provide different templates for task models: one for periodic tasks and several for tasks with dependencies, all of which are parameterised with explicit sequence of task actions and may be customised to a particular resource sharing protocol. We also investigate the scalability of the approach by allowing different best-case execution times (BCET) as a percentage discount from WCET. In practice it is possible to put realistic BCETs, but we choose this parametrisation for the sake of systematic exploration. Our approach uses the same task descriptions as [Palm(2006)].

The following outlines the main modelling ingredients:

- One template for the CPU scheduler.
- One template for the “idle” task to keep track of CPU usage times.
- One template for all BSW tasks, where resources are locked based on priority inheritance protocol.
- One template for the MainCycle ASW task, which is released periodically, starts other ASW tasks and locks resources based on the priority ceiling protocol.

- One template for all other ASW tasks, which are released by synchronisations, and locks resources based on priority ceiling protocol.
- Task specialisation is performed during process instantiation by providing individual list of operations encoded into a *flow* array.
- Each task (either ASW or BSW) uses the following clocks and data variables:
  - Task and its clocks are parameterised by an identifier *id*.
  - A local clock *x* controls periodic releases of the task. The task then moves to an error state if *x* is greater than its deadline.
  - A local clock *sub* controls progress and execution of individual operations.
  - A local integer *ic* is an operation counter.
  - The worst-case response-time for task *id* is modelled by a stopwatch $WCRT[id]$ which is reset when the task is started and is allowed to progress only when the task is ready (global invariant $WCRT[id]' == ready[id]$ ensures that). The worst-case response-time is estimated as maximum value of $WCRT[id]$.
  - An `error` location is reachable and *error* variable is set to *true* if there is a possibility to finish after deadline.

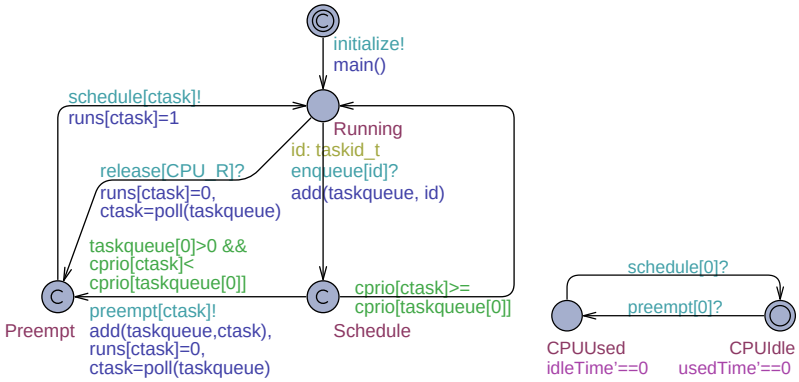
Further we explain the most important model templates, while the complete model is available for download at http://www.cs.aau.dk/∼marius/Terma/ .

2.1 Scheduler Model

Figure 3a shows the model of the scheduler. In the beginning, the Scheduler initialises the system (computes the current task priorities by computing default priority based on `id` and starts the tasks with zero offset) and in location `Running` waits for tasks to become ready or current task to release the CPU resource. When some task becomes ready, it adds itself to the `taskqueue` and signals on the `enqueue` channel, thus moving the Scheduler to location `Schedule`. From the location `Schedule`, the Scheduler compares the priority of a current task `cprio[ctask]` with the highest priority in the queue `cprio[taskqueue[0]]` and either returns to `Running` (nothing to reschedule) or preempts the current task `ctask`, puts it into `taskqueue` and schedules the highest priority task from `taskqueue`.

A task releases the CPU by a signal `release[CPU_R]`, in which case the Scheduler pulls the highest priority task from `taskqueue` and optionally notifies it with broadcast synchronisation on channel `schedule`.

The `taskqueue` always contains at least one ready task: IdleTask. Figure 3b shows how IdleTask reacts to Scheduler events. It also computes the CPU usage time using stopwatch `usedTime` and the total CPU load is then calculated as $\frac{\text{usedTime}}{\text{globalTime}}$.



(a) Template for CPU scheduler. (b) Idle task model.

Fig. 3: Models for CPU scheduler and the simplest task.

2.2 Tasks Templates

Task template is a generalization of a task process. We provide three task templates which share the same timed automata structure except some minor differences: BSW (started periodically, uses priority inheritance), ASW (started by other task, uses priority ceiling) and MainCycle (started periodically, starts other tasks and uses priority ceiling). The templates are instantiated with a concrete task description: period, offset, deadline and resource usage sequence we call task flow.

Figure 4 shows a template used by MainCycle which is started periodically. At first MainCycle waits for `Offset` time to elapse and moves to location `Idle` by setting the clock x to `Period`. Then the process is forced to leave the `Idle` location immediately, to signal other ASW tasks, add itself to the ready task queue and arrive to location `WaitForCPU`. When MainCycle receives notification from the scheduler it moves to location `GotCPU` and starts processing commands from the *flow* array.

Declaration of task flow array type is shown in Fig. 5a: `flow_t` is an array of operations `operation_t`, and operations are tuples of operation type `optype_t`, resource identifier `resid_t` and a timing argument `time_t` which is an integer.Figure 5b shows the beginning of the flow for the primary function task.

There are four types of operations:

1. LOCK is executed from location `tryLock` where the process attempts to acquire the resource. It blocks if the resource is not available and retries by adding itself to the processor queue again when the resource is released. It continues to location `Next` by locking the resource if the resource is available.
2. UNLOCK simply releases the resource and moves on to location `Next`. The implementation of locking and unlocking for both protocols is straightforward and fits into 28 lines of code.

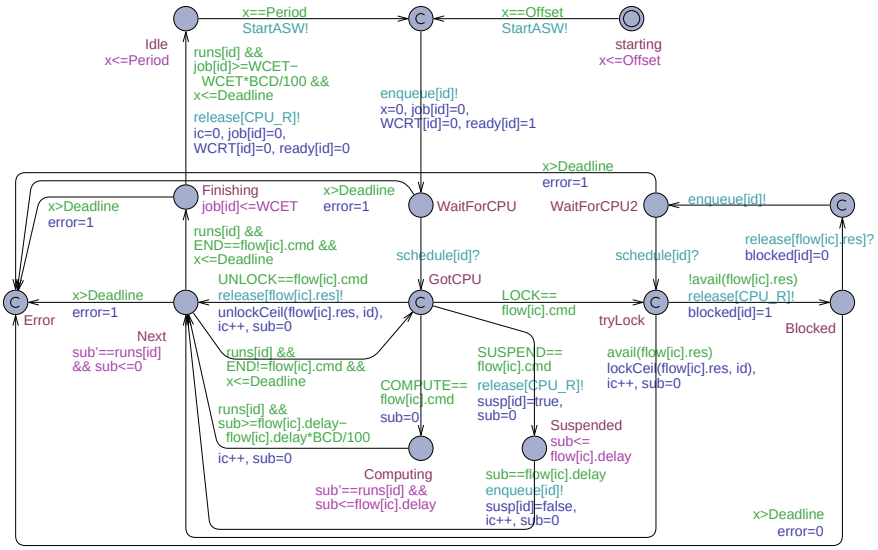


Fig. 4: MainCycle task: periodically starts ASW functions.

```
optype_t ::= END | COMPUTE | LOCK |
             UNLOCK | SUSPEND
resid_t  ::= Icb_R | Sgm_R | PmReq_R |
             Other_R
time_t   ::= int[0, 10000000]
operation_t ::= optype_t resid_t time_t
flow_t   ::= operation_t*
```

(a) Declaration of task flow type.

ic:	cmd:	res:	delay:
0	LOCK	Icb_R	0
1	COMPUTE	CPU_R	400
2	SUSPEND	CPU_R	1200
3	UNLOCK	Icb_R	1200
4	COMPUTE	CPU_R	20177
	…	…	…
18	END		

(b) Flow of primary functions task.

Fig. 5: Structure and an instance of task flow.

3. SUSPEND releases the processor for the specified amount of time, adds itself to the queue and moves to location `Next`. The task progress clock $job[id]$ is not increasing but the response measurement clock $WCRT[id]$ is.
4. COMPUTE makes the task stay in location `Computing` for the specified duration of CPU time, i.e. the clock *sub* is stopped whenever the task is preempted ($runs[id]$ is set to 0). Once the required amount of CPU time is consumed, the process moves on to location `Next`. For scalability study we relax the guard by `BCD` percent of time, allowing the task to finish slightly earlier than WCET.

From location `Next`, the process is forced by the $runs[id]$ invariant to continue with the next operation: if it is not the END and it is running, then it moves back to `GotCPU` to process next operation, and it moves to `Finishing` if it's the END. In the `Finishing` location the process consumed CPU for the remaining time so that the complete WCET is exhausted and then it moves back to `Idle`. From locations

`Next` and `Finishing` the outgoing edges are constrained to check whether the deadline has been reached since the last task release (when x was set to 0), and edges force the process into `Error` location if $x > Deadline$.

The *flow* for MainCycle is trivial: it computes for its WCET while keeping a lock on Sgm_R. A more sophisticated example of flow is shown in Listing 1 where the timing numbers are taken from description in Table 1: the task attempts to lock the resource `Icb_R`, when the resource is locked it actively uses the CPU for $400\mu s$ (because according to the description the resource is locked for $1600\mu s$ and CPU is not used for $1200\mu s$ due to suspension), then CPU is suspended for $1200\mu s$, `Icb_R` is released and CPU is used for the remaining task execution.

Listing 1: The data processing part of operation flow for PrimaryF task.

```
const ASWFlow_t PF_f = { // Primary Functions, ----- Data processing:
    { LOCK,    Icb_R,  0 },          // 0)  acquire lock on Icb_R
    { COMPUTE, CPU_R, 1600-1200 }, // 1) execute with Icb_R being locked
    { SUSPEND, CPU_R, 1200 },       // 2)  suspend/release CPU, while Icb_R is locked
    { UNLOCK, Icb_R, 0 },           // 3)  release lock on Icb_R
    { COMPUTE, CPU_R, 20577-(1600-1200) }, // 4) execute without Icb_R
     ...
};
```

The template for BSW tasks is almost the same as MainCycle, except that 1) BSW tasks do not have to start other ASW tasks and thus from `Idle` they go directly to `WaitForCPU` with enqueueing edge, 2) instead of the ceiling protocol (*lockCeil* and *unlockCeil*) it uses priority inheritance (*lockInh* and *unlockInh*) and 3) it boosts the owners priority by calling $boostPrio(flow[ic].res, id)$ on the edge from `tryLock` to `Blocked`. BSW tasks have their own local clock x, while MainCycle shares its x with other ASW tasks.

We use only LCS (CPU suspension time while resource is locked) and LC (total locking time) from Table 1, where we assume that LC−LCS is the CPU busy time while the resource is locked.

Listing 1 shows an example of detailed control flow structure for PrimaryF task, where the numbers mean the time duration and comments relate each step to an item in Table 1.

2.3 System Model Instantiation

Listing 2 shows how tasks are instantiated with task identifier, offset, period, flow, deadline and shared ASW clock. In total there are 32 tasks, where id=13 is reserved for priority ceiling.

Listing 2: Task instantiation.

```
//                    taskid , Offset , Period , flow ,    WCET, Deadline
RTEMS_RTC    =      BSW(1,    0,  10000, WCET_f,  13,      1000);
AswSync_SyncPulseIsr=BSW(2,   0,250000, WCET_f,  70,      1000);
Hk_SamplerIsr  =    BSW(3,62500,125000, WCET_f, 70,      1000);
 ...
mainCycle =    MainCycle(16,20000,250000,  400,   230220, ASWclock);
```

```
 ...
primaryF   =          ASW(21,StartASW,Done, PF_f, 34050, 59600, ASWclock);
 ...
Bkgnd_P        =     BSW(33,    0,250000, WCET_f, 200, 250000);
```

Listing 3 shows system declaration with a . The variable `cycle` counts cycle number as an heuristic progress measure which allows UPPAAL to use the sweep-line method to reduce the verification memory consumption. The cycle is incremented after a period of 250ms and is being reset after some specified `CYCLELIMIT` in the Global process. The process Global also takes care of global invariants on $job[i]$ and $WCRT[i]$ stopwatches of each task i.

Listing 3: System declaration using UPPAAL priorities.

```
system Scheduler, RTEMS_RTC, AswSync_SyncPulselsr, Hk_SamplerIsr, SwCyc_CycStartIsr,
    SwCyc_CycEndIsr, Rt1553_Isr, Bc1553_Isr, Spw_Isr, Obdh_Isr, RtSdb_P_1, RtSdb_P_2, RtSdb_P_3,
    FdirEvents, NominalEvents_1, mainCycle, HkSampler_P_2, HkSampler_P_1, Acb_P, IoCyc_P,
    primaryF, rCSControlF, Obt_P, Hk_P, StsMon_P, TmGen_P, Sgm_P, TcRouter_P, Cmd_P,
    NominalEvents_2, secondF_1, secondF_2, Bkgnd_P, IdleTask, Global;
progress { cycle; }
```

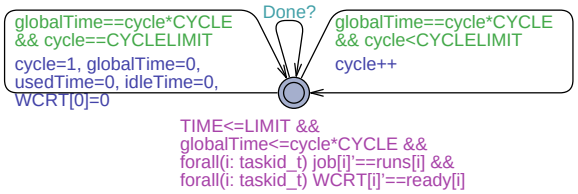


Fig. 6: Global process enforce invariants on stopwatches and cyclic progress.

2.4 Verification Queries

The following is a list of queries used to check schedulability properties:

- Check if the system is schedulable (the error state is not reachable): `E<> error`
- Check if any task can be blocked at all: `E<> exists(i:taskid_t) blocked[i]`
- Find the total worst CPU usage: `sup: usedTime, idleTime`
- Find the worst-case response-times: `sup: WCRT[0], WCRT[1], ... WCRT[33]`
- Find worst-case blocking times, where $B[i]$ is a stopwatch growing when task i is blocked: `sup: B[0], B[1], B[2], ... B[33]`

A `sup`-query explores the entire reachable state space and computes the maximum (supremum) value of an argument expression. This is particularly useful for computing several bounds at once.

3 Results

Our results provide three important pieces of information: visualisation of a schedule in a Gantt chart, worst-case response-times estimates and CPU utilisation and verification benchmarks.

A Gantt chart can be used to visualise a trace of the system, thus providing a rich picture for inspection. For example, the generated Gantt chart in Figure 7 shows that Cmd_P is blocked more than 5 times during the first cycle, while blocking times for PrimaryF (21) and StsMon_P (25) are significantly long only starting from the second cycle.

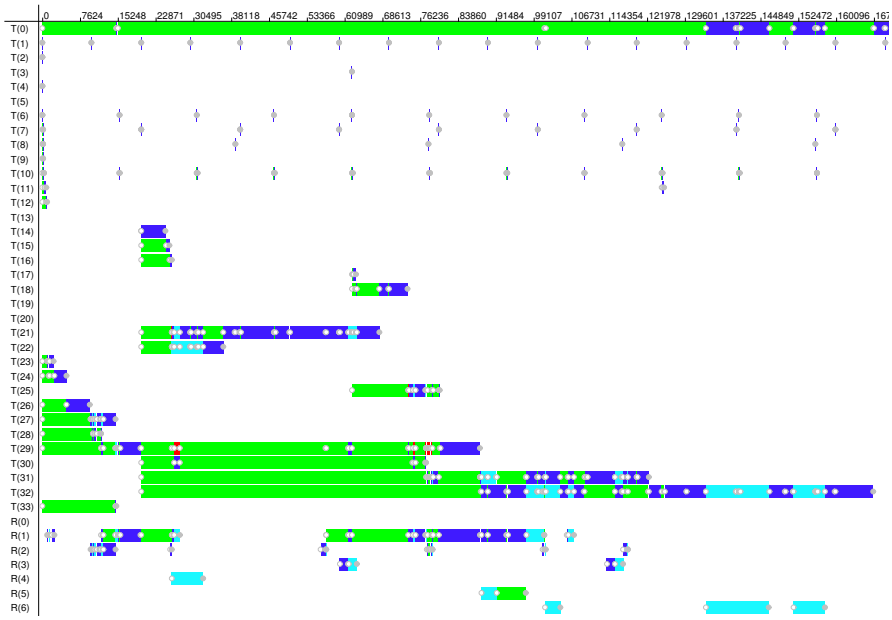


Fig. 7: Gantt chart of the first cycle, generated by UPPAAL TIGA: task T(i) is green when ready, blue – executing, red – blocked, cyan – suspended, resource R(j) is blue when locked and owner uses CPU, green – locked but the owner is preempted, cyan – locked but owner is suspended.

In [Palm(2006)] the CPU utilisation for a 20-250ms window is estimated as 62.4%. Our estimate for the entire worst-case cycle is 63.65% which is slightly larger, possibly due to the fact that it also includes the consumption during the 0-20ms window. See [Mikučionis et al(2010)Mikučionis, Larsen, Rasmussen, Nielsen, Skou, Palm, Pedersen, and Hou for additional insight on how the cycle limit affects verification resources and results.

Table 2 shows the worst-case response-times obtained from UPPAAL analysis with 0%, 5% and 10% BCET deviation from WCET in comparison with response-

times acquired by Terma. We note that in all cases the WCRT estimates provided by UPPAAL are smaller (hence less pessimistic) than those originally obtained [Palm(2006)]. In particular, we note that the task PrimaryF (task 21) is found to be schedulable using model-checking with up to 10% deviation for best-case execution times, but most probably not schedulable from 14% (a trace leading to deadline violation is found), in contrast to the original negative result obtained by Terma.

Table 2: Specification, blocking and worst-case response-times of individual tasks.

		Specification			WCRT			
ID	Task	Period	WCET	Deadline	Terma	0%	5%	10%
1	RTEMS_RTC	10.000	0.013	1.000	0.050	0.013	0.013	0.013
2	AswSync_SyncPulseIsr	250.000	0.070	1.000	0.120	0.083	0.083	0.083
3	Hk_SamplerIsr	125.000	0.070	1.000	0.120	0.070	0.070	0.070
4	SwCyc_CycStartIsr	250.000	0.200	1.000	0.320	0.103	0.103	0.103
5	SwCyc_CycEndIsr	250.000	0.100	1.000	0.220	0.113	0.113	0.113
6	Rt1553_Isr	15.625	0.070	1.000	0.290	0.173	0.173	0.173
7	Bc1553_Isr	20.000	0.070	1.000	0.360	0.243	0.243	0.243
8	Spw_Isr	39.000	0.070	2.000	0.430	0.313	0.313	0.313
9	Obdh_Isr	250.000	0.070	2.000	0.500	0.383	0.383	0.383
10	RtSdb_P_1	15.625	0.150	15.625	4.330	0.533	0.533	0.533
11	RtSdb_P_2	125.000	0.400	15.625	4.870	0.933	0.933	0.933
12	RtSdb_P_3	250.000	0.170	15.625	5.110	1.103	1.103	1.103
14	**FdirEvents**	250.000	5.000	230.220	7.180	5.553	5.553	5.553
15	**NominalEvents_1**	250.000	0.720	230.220	7.900	6.273	6.273	6.273
16	**MainCycle**	250.000	0.400	230.220	8.370	6.273	6.273	6.273
17	HkSampler_P_2	125.000	0.500	62.500	11.960	5.380	7.350	8.153
18	HkSampler_P_1	250.000	6.000	62.500	18.460	11.615	13.653	14.153
19	Acb_P	250.000	6.000	50.000	24.680	6.473	6.473	6.473
20	IoCyc_P	250.000	3.000	50.000	27.820	9.473	9.473	9.473
21	**PrimaryF**	250.000	34.050	59.600	**65.47**	54.115	56.382	58.586
22	**RCSControlF**	250.000	4.070	239.600	76.040	53.994	56.943	58.095
23	Obt_P	1000.000	1.100	100.000	74.720	2.503	2.513	2.523
24	Hk_P	250.000	2.750	250.000	6.800	4.953	4.963	4.973
25	StsMon_P	250.000	3.300	125.000	85.050	17.863	27.935	28.086
26	TmGen_P	250.000	4.860	250.000	77.650	9.813	9.823	9.833
27	Sgm_P	250.000	4.020	250.000	18.680	14.796	14.880	14.973
28	TcRouter_P	250.000	0.500	250.000	19.310	11.896	11.906	14.442
29	Cmd_P	250.000	14.000	250.000	114.920	94.346	99.607	101.563
30	**NominalEvents_2**	250.000	1.780	230.220	102.760	65.177	69.612	72.235
31	**SecondaryF_1**	250.000	20.960	189.600	141.550	110.666	114.921	122.140
32	**SecondaryF_2**	250.000	39.690	230.220	204.050	154.556	162.177	165.103
33	Bkgnd_P	250.000	0.200	250.000	154.090	15.046	139.712	147.160

On a Linux server with Intel Xeon E5420 2.5GHz processor UPPAAL takes 2min 40s to verify that the system is schedulable, 6min 30s to find WCRTs with 0% BCET deviation. In case of 10% BCET deviation it took slightly over 6 days to establish schedulability and slightly over 7 days of 6 parallel runs to find all WCRTs. Table 3 shows the amount of verification resources UPPAAL requires to verify schedulability with different task execution time windows and model time limits. In this study we used compact data structure (CDS) to store the clock valuations in contrast to difference bound matrices (DBM) in previous study, which explains why the verification is slower, but the memory usage is limited and varies very little across model time limits.

In addition UPPAAL reported that the system is not schedulable when the task execution time window is larger than 14%. We found that the cycle limit granularity (used to define the progress measure) affects performance as well as the outcome: the larger cycles lead to error state being reachable because larger cycles result in the coarser stop-watch over-approximation. For example, binary search method revealed that with a 20% task execution window the error is reachable within the first 250ms period when the cycle is larger than 8017*ms* and otherwise it is not. However the error is reachable in the second 250ms period even if the cycle is as small as 2ms (verification took 24 hours).

Table 3: Verification statistics for different task execution time windows and exploration limits: the percentage denotes difference between WCET and BCET, limit is in terms of 250ms cycles (∞ stands for no limit, i.e. full exploration), memory in MB, time in seconds.

limit	0%			5%			10%			14%		
	states	mem	time	states	mem	time	states	mem	time, s	states	mem	time
1	1300	51.2	1.47	485077	83.0	903.1	1481162	124.1	4962.8	3348246	186.9	23986.5
2	2522	53.7	2.45	806914	96.8	1619.9	2414679	139.7	7755.2	5253778	198.7	33299.2
4	4981	54.5	4.62	1499700	97.2	2881.8	4421630	138.3	13720.0	9231399	274.6	51176.6
8	9928	54.7	8.48	2828776	97.8	5325.1	9093562	156.5	31122.3	18240030	364.6	102932.4
16	19805	55.3	16.11	5366015	112.0	9952.0	17798572	176.0	60124.5	35432003	520.4	158816.7
∞	196336	58.8	159.64	52728344	553.9	97507.4	181869652	1682.2	530604.9	error may be reachable		

4 Discussion

We have shown how UPPAAL can be applied for schedulability analysis of a system with a single CPU, fixed priorities preemptive scheduler, mixture of periodic tasks and tasks with dependencies, and mixed resource sharing protocols. Worst-case response-times (WCRT), blocking times and CPU utilisation are estimated by using model-checker according to detailed task models. Our modelling patterns use stopwatches in a simple and intuitive way. A break-through in verification scalability for large systems (more than 30 tasks) is achieved by employing the sweep-line method.

The task templates are demonstrated to be generic through many instantiations with arbitrary computation sequences and specialised for particular resource sharing. The framework is modular and extensible to accommodate a different scheduler and control flow can be expanded with additional instructions if some task algorithm is even more complicated. In addition, UPPAAL allows easy visualisation of the schedule in Gantt chart and the system behaviour can be examined in both symbolic and concrete simulators.

The case study results include a self-contained non-ambiguous model which formalises many informal assumptions described in [Palm(2006)] in human language. The verification results demonstrate that the timing estimates correlate with figures

from the response-time analysis [Palm(2006)]. The worst-case response-time of PrimaryF is indeed very close to its deadline, but overall, all estimates by UPPAAL are lower (more optimistic) and they all ($WCRT_{21}$ in particular) are below deadlines, whereas the classical response-time analysis found that PrimaryF may not finish before deadline and does not provide any more insight on how the deadline is violated or whether such behaviour is realizable.

By relaxing the lower bound of task execution time we showed that the system is probably not schedulable if BCET deviates from WCET by 15% or more. We found that it is better to start exploration with small task execution windows with large progress cycles first and limit the model time (effectively limiting the verification resources), then progress gradually with larger windows and then use smaller cycles to refine over-approximation. The large task execution windows (e.g. 20% with small progress cycles, or simple case of 50% with large cycles) can take days just to find the error and potentially much longer if there is no error.

We plan to conduct a similar study to allow sporadic tasks and apply statistical model-checking methods to investigate the probability of deadline violation as a cheaper means to detect errors.

So far we have not addressed margin analysis (as part of response-time analysis), but we see no principle obstacle to use the binary search method to find upper bounds for task execution times.

4.1 Related Work

Process algebraic approach has resulted in many methods for specification and schedulability analysis of real-time systems. For example [Ben-Abdallah et al(1998)Ben-Abdallah, Choi, Clarke, Ki
provide an overview of this long tradition.

In [Waszniowski and Hanzálek(2008)] it is shown how a multitasking application running under a real-time operating system compliant with an OSEK/VDX standard can be modelled by timed automata. Use of this methodology is demonstrated on an automated gearbox case study and the worst-case response-times obtained from model-checking is compared with those provided by classical schedulability analysis showing that the model-checking approach provides less pessimistic results due to a more detailed model and exhaustive state-space exploration.

The Times tool [Amnell et al(2002)Amnell, Fersman, Mokrushin, Pettersson, and Yi] can be used to analyse single processor systems, however it supports only highest locker protocol (simplified priority ceiling protocol) [Fersman(2003)]. Approaches like [Bøgholm et al(2008)Bøgholm, Kragh-Hansen, Olsen, Thomsen, and Larsen] and [Brekling et al(2009)Brekling, Hansen, and Madsen] provides external transformation into UPPAAL [Behrmann et al(2004)Behrmann, David, and Larsen] timed-automata for schedulability analysis.

References

[Amnell et al(2002)Amnell, Fersman, Mokrushin, Pettersson, and Yi] Amnell T, Fersman E, Mokrushin L, Pettersson P, Yi W (2002) TIMES – a tool for modelling and implementation of embedded systems. In: TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, London, UK, pp 460–464

[Behrmann et al(2004)Behrmann, David, and Larsen] Behrmann G, David A, Larsen K (2004) A tutorial on Uppaal. Lecture Notes in Computer Science pp 200–236

[Ben-Abdallah et al(1998)Ben-Abdallah, Choi, Clarke, Kim, Lee, and Xie] Ben-Abdallah H, Choi JY, Clarke D, Kim YS, Lee I, Xie HL (1998) A process algebraic approach to the schedulability analysis of real-time systems. Real-Time Systems 15:189–219, URL http://dx.doi.org/10.1023/A:1008047130023, 10.1023/A:1008047130023

[Bøgholm et al(2008)Bøgholm, Kragh-Hansen, Olsen, Thomsen, and Larsen] Bøgholm T, Kragh-Hansen H, Olsen P, Thomsen B, Larsen KG (2008) Model-based schedulability analysis of safety critical hard real-time java programs. In: Bollella G, Locke CD (eds) JTRES, ACM, ACM International Conference Proceeding Series, vol 343, pp 106–114

[Brekling et al(2009)Brekling, Hansen, and Madsen] Brekling A, Hansen M, Madsen J (2009) Moves – a framework for modelling and verifying embedded systems. In: Microelectronics (ICM), 2009 International Conference on, pp 149–152, DOI 10.1109/ICM.2009.5418667

[Burns(1994)] Burns A (1994) Preemptive priority based scheduling: An appropriate engineering approach. In: Principles of Real-Time Systems, Prentice Hall, pp 225–248

[Fersman(2003)] Fersman E (2003) A generic approach to schedulability analysis of real-time systems. Acta Universitatis Upsaliensis

[Mikučionis et al(2010)Mikučionis, Larsen, Rasmussen, Nielsen, Skou, Palm, Pedersen, and Hougaard] Mikučionis M, Larsen KG, Rasmussen JI, Nielsen B, Skou A, Palm SU, Pedersen JS, Hougaard P (2010) Schedulability analysis using uppaal: Herschel-planck case study. In: Margaria T (ed) ISoLA 2010 – 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, vol Lecture Notes in Computer Science

[Palm(2006)] Palm S (2006) Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S

[Terma A/S(Issue 9)] Terma A/S (Issue 9) Software timing and sizing budgets. Tech. rep., Terma A/S

[Waszniowski and Hanzálek(2008)] Waszniowski L, Hanzálek Z (2008) Formal verification of multitasking applications based on timed automata model. Real-Time Systems 38(1):39–65

Chapter 1
An Introduction to Automatic Synthesis of Discrete and Timed Controllers

Franck Cassez, Kim Larsen, Jean-François Raskin, and Pierre-Alain Reynier

1.1 Introduction

In this chapter, we introduce models and algorithms for the automatic synthesis of controllers for discrete and timed (infinite state) systems. The techniques that we expose here are based on the game metaphor [5, 4]: when designing an embedded controller, you can see the controller as interacting with its environment. As the actions taken by the environment are uncontrollable, those actions should be considered as *adversarial*. Indeed, a controller should be correct no matter how the environment in which it operates behaves. The models, algorithms and tools presented here are applied to an industrial case study in the next chapter. This case study was provided to us by HYDAC ELECTRONIC GMBH within the Quasimodo project.

The objective of the chapter is to allow the reader to understand timed game automata [3] as a model for solving timed control problems. With this objective in mind, we define the notions of game graphs, controllable and uncontrollable actions, strategies, and winning objectives. We also give a gentle introduction to the main algorithmic ideas that are used to solve games played on graphs. Those techniques are used in the tool UPPAAL-TIGA [1]. A good understanding of the techniques used in UPPAAL-TIGA should help the users when modeling control problems and formulating queries about their models.

The chapter is organized as follows. In section 1.2, we introduce an example of a timed control problem called the‘Chinese juggler control problem”. This example allows us to illustrate the game metaphor for formalizing the timed control problem. In Section 1.3, we introduce the basic definitions underlying the game approach to controller synthesis. In Section 1.4, we outline two algorithms that are used to solve (untimed) reachability and safety games respectively. In Section 1.5, we show how the concepts developed in Sections 1.3 and 1.4 can be extended to timed systems. In Section 1.6, we summarize the main ideas underlying the algorithms for solving timed games. In Section 1.7, we give an introduction to the tool UPPAAL-TIGA and

show how to model and automatically solve the Chinese juggler control problem with timed game automata.

1.2 The Chinese juggler control problem

In this section, we introduce a running example that we use later in this chapter to illustrate how timed controllers can be automatically synthesized using the tool UPPAAL-TIGA. The example also allows us to illustrate the game metaphor for controller synthesis that underlies the development of the theory in Sections 1.3 and 1.4.

A Chinese juggler has to spin plates that are on sticks to prevent them from falling, see Fig. 1.1 for an illustration. Assume, for our example, that the juggler wants to handle two plates, called `Plate 1` and `Plate 2`. Plates crash after a while if they are not spun. Initially, each plate is spinning on its stick and the spin is fast enough so that they will stay stable for at least 5 seconds. The juggler has to maintain them stable for as long as possible (forever if possible). For that, the juggler can spin each plate but he can spin only one of the plates at a time. When he decides to spin `Plate` $i \in \{1,2\}$, he should do it for at least 1 time unit. If he decides to do it for t time units then `Plate` i stays stable for 3 time units if $1 \leq t \leq 2$, and for 5 time units if $t > 2$.

Now, assume that there is also a mosquito in the room. When the mosquito touches one of the two plates, it reduces the spinning of the plate, and as a result its remaining stability time is decreased by 1 time unit. When the mosquito touches the plate, it gets afraid and this guarantees that it will not touch any plate again before D time units have elapsed (after that amount of time the mosquito has forgotten and he is not afraid any more).

We want to answer the following question (CP):

> Given a value for D, does the Chinese juggler have a way to spin the plates so that none of the two plates ever falls down no matter what the behavior of the mosquito is ?

Let us first try to understand how this timed control problem can be seen as a two player game. In the system underlying our example, we have several components: the Chinese juggler, the plates, and the mosquito. Clearly, only the behavior of the Chinese juggler is under control. The plates and the mosquito are part of the environment: when a plate has not been spun enough, it can fall at any time, and the behavior of the mosquito is out of control of the juggler, i.e. the mosquito decides when it touches plates. As a consequence, we can see the control problem as *opposing* two players: on one hand the Chinese juggler (Player 1), and on the other hand the plates and the mosquito (Player 2).

During this game, at any point in time, the Chinese juggler may decide to spin one of the two plates. If he decides to do so, he will do it for at least one time unit. Then either he decides to continue to spin the plate, or to stop and remain idle for a while, or to start spinning the other plate. The alternatives that are offered to

Fig. 1.1 A Chinese juggler (cartoon courtesy of Jean Cardinal.)

the juggler along time can be understood as *moves* in the underlying game. The mosquito, if it has not touched a plate in the preceding D time units, may decide to touch one of the two plates whenever it wishes to do so. Again, those alternatives can be seen as moves in the underlying game. Similarly, when a plate does not spin fast enough then it may crash at any moment. To summarize, the only moves that we *control* are the moves of the Chinese juggler, they are the moves of Player 1, all the other moves are *uncontrollable*, they are the moves of Player 2. We must be prepared to face all the moves available to the mosquito and to the plates.

Second, we need to understand what the *objectives* of the two players are. The objective of the Chinese juggler is to avoid the plates to crash. For the objective of the plates and the mosquito (Player 2), it may not be as clear. The mosquito flies randomly in the room and touches one of the plates on occasion. But clearly, we want to devise a strategy for the Chinese juggler such that, *whatever the behavior of the mosquito is* (within the hypothesis that it does not touch twice the plates within less than D time units), the plates never crash. So, even if we do not know exactly the intention (or the exact specification) of the mosquito, it is safe to be prepared for the worst case scenario. So the kind of game that we consider are *zero sum games*: a set of behaviors (of the system) is identified as good for Player 1, and the complement of this set (all other behaviors) are considered as good for Player 2.

1.3 Control as a two-player games

Now that we agree that control problems can be seen as two-player games, we introduce the precise definitions underlying the theory of two-player games played on graphs. Later we extend those notions with dense time. After presenting timed games, we show how to model the Chinese juggler problem with timed game au-

tomata and how we can answer question (*CP*); moreover if the answer is yes we also show how to synthesize automatically a winning strategy for the Chinese juggler using the tool UPPAAL-TIGA.

1.3.1 Game structures

A *game structure* is a tuple $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ where L is a finite (non-empty) set of locations, $\ell_{\mathsf{init}} \in L$ is the *initial location* of the game, Act_1 and Act_2 are the two disjoint *sets of actions* for Player 1 (the controller) and Player 2 (the environment) respectively, and $E \subseteq L \times \mathsf{Act}_1 \cup \mathsf{Act}_2 \times L$ is a *set of edges* between the locations of the game labelled by actions that belong either to Player 1 or to Player 2. Intuitively, edges labeled with elements from Act_1 belong to Player 1 and are controllable (represented by plain arrows) while edges labeled by elements from Act_2 belong to Player 2 and are uncontrollable (represented by dashed arrows). We let $\mathsf{Enabled}_i(\ell)$ be the set of actions of Player $i \in \{1,2\}$ available at location ℓ i.e. $\mathsf{Enabled}_i(\ell) = \{\alpha \in \mathsf{Act}_i \mid \exists(\ell, \alpha, \ell') \in E\}$.

We require that for all $\ell \in L$, $\mathsf{Enabled}_1(\ell) \neq \varnothing$, so that Player 1 is always able to propose an action to play in any location of the game.

Example 1. Fig. 1.2 depicts a game structure. The set of locations is $L = \{\mathtt{L0}, \mathtt{L1}, \mathtt{L2}, \mathtt{L3}\}$. In location L0, Player 1 can choose between action a or action b, while Player 2 can choose between action $u1$ and action $u2$. L0 (inner circle) is the initial location of the game. The edge $(\mathtt{L0}, a, \mathtt{L1})$ belongs to Player 1 and the edge $(\mathtt{L0}, u1, \mathtt{L1})$ belongs to Player 2. □

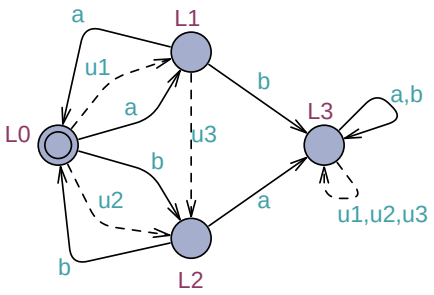


Fig. 1.2 An example of a game structure

The way players are playing on a game structure $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ is defined as follows. Initially, a pebble lies on ℓ_{init}, the initial location of game. Then the game is played in rounds. Let us assume that, for the current round, the pebble lies on location $\ell \in L$. Then, first, Player 1 chooses some action $\alpha \in \mathsf{Enabled}_1(\ell)$.

Then Player 2 decides where to move the pebble onto the successor locations of ℓ while respecting the following rule: for moving the pebble she uses either an edge labeled with an action of Act_2 or an edge[1] labeled with the action α chosen by Player 1. By interacting in such a way for an infinite number of rounds, Player 1 and Player 2 are constructing a *play*. Formally a play $\rho = \ell_0\ell_1\ldots\ell_n\ldots$ of the game structure G is an infinite sequence of locations. We let $\rho[i] = \ell_i, i \geq 0$ and denote $\mathsf{Play}(G)$ for the set of plays of G.

Example 2. Let us illustrate the notion of play using the example of Fig. 1.2. As L0 is the initial location of the game structure G, the pebble initially lies on L0. Then Player 1 is asked to make a choice among the actions that are available for her in location L0. This set is $\{a,b\}$. Assume that she chooses a. In this case, there are two possibilities. Either, Player 2 chooses to let Player 1 play and the pebble is moved using an edge labeled with the letter a. In our example, there is only one such edge, and so the pebble is moved on location L1. By this interaction, a finite prefix L0L1 of play is built. Or, Player 2 chooses to overtake Player 1 and to play $u2$; in that case, the finite prefix L0 L2 of a play is built. Assume that the second situation applies. Then a new round starts in L2. In that location, there is no uncontrollable transition, so if Player 1 chooses a then the pebble is moved to L3 and if she chooses b, it is moved to L0, etc. □

1.3.2 Winning objectives and strategies

We have seen that the interaction between Player 1 (the controller) and Player 2 (the environment) on a game structure $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ generates a play which is an infinite sequence $\ell_0\ell_1\ldots\ell_n\ldots$ of locations in the game graph, i.e. the sequence of locations traversed by the pebble during the course of the game. Such a sequence models one behavior of the system under control, and this behavior could be considered as a *good* behavior or as a *bad* behavior depending on what we expect from our system[2]. In the game terminology, such a classification of good and bad behaviors leads to the notion of winning objective. A *winning objective* for a game structure G is a set of infinite sequences of locations, the intention being that such sequences represent the good behaviors of the system.

Example 3. Assume that in our running example, Player 1 has the objective to reach the set of locations $\{\mathtt{L3},\mathtt{L4}\}$. In this case the winning objective will contain all the plays $\ell_0\ell_1\ell_2\ldots\ell_n\ldots$ such that $\ell_i = \mathtt{L3} \vee \ell_i = \mathtt{L4}$ for some $i \geq 0$. □

In the example above, the winning objective is a so-called *reachability objective* as it specifies a set of locations that we want to visit. In this chapter, we concentrate on two classes of objectives: *reachability* and *safety*. Given a set of *target* locations

[1] There might be more than one α-successor of ℓ. In this case, Player 2 resolves the non-deterministic choice of the α-successor.

[2] As stated earlier, we play zero-sum games and in this case a play is either good or bad.

$T \subseteq L$, we define the set of winning plays of the *reachability objective defined by* T as the set of plays $\mathsf{Reach}_G(T) = \{\rho \in \mathsf{Play}(G) \text{ s.t. } \exists i \geq 0 \cdot \rho[i] \in T\}$. Given a set of *safe* locations $S \subseteq L$, we define the set of winning plays of the *safety objective defined by* S as the set $\mathsf{Safe}_G(S) = \{\rho \in \mathsf{Play}(G) \text{ s.t. } \forall i \geq 0 \cdot \rho[i] \in S\}$. In the sequel, we often use Obj to represent a set of winning plays.

The winning objective specifies what the good plays for Player 1 are. Those good behaviors can be enforced by the controller (Player 1) if she has a strategy to force the play to be within the winning objective no matter what the strategy played by Player 2 is (so without the help of Player 2). For the later definition to be completely clear, we need to define more precisely what a *strategy* is. In our games, a strategy for Player 1 determines what actions from Act_1 to pick during the course of the game. In general, a strategy may depend on the history of the game for deciding what the good action to play is. Nevertheless, for reachability and safety objectives, the situation is simpler and it can be shown that strategies that only depend on the current position of the pebble are sufficient: those strategies are called "memoryless" strategies. So, in this chapter we concentrate on such simple strategies. We now define them formally. A *(memoryless) strategy* for Player 1 is a function $\lambda_1 : L \to \mathsf{Act}_1$, i.e. it is a function that given the current location $\ell \in L$ chooses an action $\lambda_1(\ell) \in \mathsf{Enabled}_1(\ell)$ for Player 1.

Let us now define the possible behaviors in the game structure $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ when Player 1 plays according to the strategy λ_1. Remember that Player 1, in the game above, chooses an action at each round. Then Player 2 chooses between the edges labeled by this action or labeled with one of her own actions. The set of behaviors in this case is thus the set of paths that start in ℓ_{init} and use only edges that are either labeled with actions of Player 2 or labeled with actions that are prescribed by the strategy λ_1. We can also see a strategy for Player 1 as cutting out edges of Player 1 that are not chosen by the strategy. Let us define that formally. We call the *outcome of the strategy* λ_1 in the game $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ the set of plays

$$\mathsf{Outcome}(G, \lambda_1) = \{\rho \mid \forall i \geq 0, \exists a \in \lambda_1(\rho[i]) \cup \mathsf{Act}_2 \cdot (\rho[i], a, \rho[i+1]) \in E\}.$$

A strategy λ_1 is *winning* for the objective Obj in G if $\mathsf{Outcome}(G, \lambda_1) \subseteq \mathsf{Obj}$.

Example 4. Let us consider again the example of Fig. 1.2. Assume that the winning objective for Player 1 is to reach location L3, i.e. $\mathsf{Obj} = \mathsf{Reach}_G(\{\mathtt{L3}\})$. Let us consider the strategy λ_1 defined as follows: $\mathtt{L0} \mapsto b$, $\mathtt{L1} \mapsto b$, $\mathtt{L2} \mapsto a$, and $\mathtt{L3} \mapsto a$. It should be clear that no matter how Player 2 plays when the play starts in L0, the result of the interaction with this strategy λ_1 is a play that reaches L3. For example, let us consider the following scenario: in L0, instead of playing b as chosen by Player 1, Player 2 moves the pebble to location L1. From there, instead of playing b as asked by Player 1 (this would lead directly to L3), Player 2 moves the pebble to L2. From L2, Player 1 chooses a and Player 2 has no other choices than to move the pebble to L3. So, under any adversarial behavior of Player 2, Player 1 can force the pebble to reach location L3. As a consequence, λ_1 is a winning strategy for Player 1 to win the reachability game defined by the objective $\mathsf{Obj} = \mathsf{Reach}_G(\{\mathtt{L3}\})$. □

1.4 Solving two-player games

In the previous section, we have defined two-player game structures, reachability and safety winning objectives, strategies for Player 1, and we have explained when a strategy for Player 1 is winning. In this section, we introduce the basic ideas that are underlying algorithms for solving games with safety and reachability objectives.

To understand the basic ideas behind the algorithms for solving reachability and safety games, we must first concentrate on what happens in one round, i.e. we need to consider one-step objectives. A *one step objective* is defined by a set of locations $T \subseteq L$. In a location $\ell \in L$ of the game $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$, Player 1 wins the one step objective T if there exists an action $\alpha \in \mathsf{Act}_1$ such that all edges labeled by α and all edges labeled with Act_2 actions lead to a location in T, i.e. ℓ is such that

$$\exists \alpha \in \mathsf{Act}_1 \cdot \forall \beta \in \mathsf{Act}_2 \cup \{\alpha\} \cdot \forall (\ell, \beta, \ell') \in E : \ell' \in T.$$

In that case, we say that ℓ is a *controllable predecessor* of T, and we denote by $\mathsf{CPre}(T)$ the set of locations that are controllable predecessors of T.

Example 5. To illustrate the definition of *controllable predecessors*, we use Fig. 1.3. First, let us consider the set of locations $T_1 = \{\mathtt{L1}, \mathtt{L3}\}$. The location L0 is a controllable predecessor of T_1. Indeed, in L0 if Player 1 chooses a, no matter what is the choice of Player 2 (to move the pebble using an edge labeled with a or to play an edge labeled with her own actions) the pebble will be either in L1 or L3 after the round, so it will lie in T_1. Second, let us consider the set of locations $T_2 = \{\mathtt{L1}, \mathtt{L2}\}$. The location L0 is not a controllable predecessor of T_2. Indeed, neither a nor b ensures that the pebble will lie in T_2 as Player 2 can choose to go to L3 using a or $u2$ in the first case, and Player 2 can decide to go to L1 using $u1$ in the second case. □

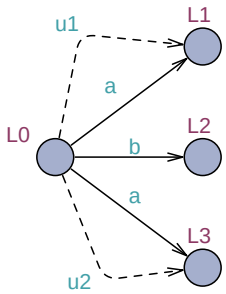


Fig. 1.3 Controllable and uncontrollable predecessors

Now that we understand what it means for a location ℓ to be a controllable predecessor of a set of locations T, we provide algorithms to solve reachability and safety games. Let us start with reachability games. Let $G = (L, \ell_{\mathsf{init}}, \mathsf{Act}_1, \mathsf{Act}_2, E)$

be a two-player game structure and $\mathsf{Obj} = \mathsf{Reach}_G(T)$ be the reachability objective for Player 1.

The algorithm that computes the set of winning locations for the reachability objective $\mathsf{Reach}_G(T)$ works by induction on the number of rounds needed for Player 1 to win. Clearly, all the locations in T are winning in 0 rounds, let us denote this set of locations by W_0. Now, it should be clear that the set of controllable predecessors of T are locations that are winning in 1 step. By taking the union of this set with W_0, we obtain the set of locations from which Player 1 can force a visit to T in 0 or 1 rounds, i.e. $W_1 = W_0 \cup \mathsf{CPre}(W_0)$. Generalizing this reasoning, we get that $W_i = W_{i-1} \cup \mathsf{CPre}(W_{i-1}), i \geq 1$ is the set of locations from which Player 1 can force a visit to the set T in less than i rounds. Clearly, we have that $W_0 \subseteq W_1 \subseteq \cdots \subseteq W_i \subseteq L$. As L is a finite set, the monotonic sequence of W_i reaches a fixed point W for some $k \leq |L|$ and $W = W_k = W_{k-1}$. The set W is the set of locations from which Player 1 has a strategy to force a visit to T in a finite number of steps. If $\ell_{\mathsf{init}} \in W$ then Player 1 has a winning strategy from the initial location of the game. From the computation of this sequence, we can extract a winning strategy for all locations in W as follows. Let $\ell \in W$ be such that $\ell \in W_i, i \geq 1$ and $\ell \notin W_{i-1}$. Define $\lambda_1(\ell)$ to be any action a such $a \in \mathsf{Act}_1$ and all the edges labeled with a that leave the location ℓ go to a location that belongs to the set W_{i-1}; because of the definition of W_i and CPre, such an action a is guaranteed to exist.

Example 6. Let us consider again the example of Fig. 1.2 with the reachability objective $\mathsf{Obj} = \mathsf{Reach}_G(\{\mathtt{L3}\})$. Let $W_0 = \{\mathtt{L3}\}$, and let us compute the set of controllable predecessors of W_0. The locations L2 and L3 are controllable predecessors of W_0. So $W_1 = \{\mathtt{L2}, \mathtt{L3}\}$ is the set of locations from which Player 1 can ensure a visit in $\{\mathtt{L3}\}$ in 0 or 1 rounds. It should be clear from the computation of the controllable predecessors of W_0 that Player 1 has to choose the action a when the pebble lies on L2. This gives a winning strategy for Player 1 in $\mathtt{L}_2$. Now, let us consider the locations that are controllable predecessors of W_1. This set is $\{\mathtt{L1}, \mathtt{L2}, \mathtt{L3}\}$. Indeed in L1 Player 1 can choose b and in this case, either Player 2 moves the pebble to L3 using edge $(\mathtt{L1}, b, \mathtt{L3})$ or she moves the pebble to L2 using the edge labeled by $u3$. In the two cases, the pebble lies on $\{\mathtt{L2}, \mathtt{L3}\}$ when starting the next round of the game. If we continue like that we obtain that all the locations of G are winning for the objective Obj and in the process we can construct a winning strategy for Player 1. □

Let us now turn to safety games. Remember that in a safety game defined by a set $S \subseteq L$ of locations, Player 1 has the objective to stay within set S forever, i.e. $\mathsf{Obj} = \mathsf{Safe}_G(S)$. Let us define, as for reachability, a sequence of sets of locations that approximate the set of winning locations for Player 1. Clearly, $W_0 = S$ is the set of locations from which Player 1 can ensure to stay within S for at least 0 rounds. Now, $W_1 = W_0 \cap \mathsf{CPre}(W_0)$ is the set of locations from which Player 1 can ensure to stay within S for at least 1 round, and more generally, $W_i = W_{i-1} \cap \mathsf{CPre}(W_{i-1}), i \geq 1$ is the set of locations from which Player 1 can ensure to stay within S for ta least i rounds. Clearly, we have that $L \supseteq W_0 \supseteq W_1 \supseteq \cdots \supseteq W_i \supseteq \cdots \supseteq \varnothing$. As L is a finite set, we must reach a fixed point W for some $k \leq |L|$ and $W = W_k = W_{k-1}$, and so the sequence eventually stabilizes on the set of locations from which Player 1 can

force to stay within S forever, i.e. on the set of locations from which Player 1 has a strategy to win the safety game defined by S.

Example 7. Let us consider again example of Fig. 1.2 but now with the objective $\mathsf{Obj} = \mathsf{Safe}_G(\{\mathtt{L}_0, \mathtt{L}_2\})$. So the objective for Player 1 is now to avoid locations $\mathtt{L}_1$ and $\mathtt{L}_3$. Let us compute the sequence of sets of locations that approximate the winning set for Player 1. By definition of this sequence, $W_0 = \{\mathtt{L}_0, \mathtt{L}_2\}$. Let us compute the controllable predecessors of this set of locations: $\mathsf{CPre}(W_0) = \{\mathtt{L2}\}$. Indeed, L0 is not a controllable predecessor of $\{\mathtt{L0}, \mathtt{L2}\}$ as, from L0, Player 2 can force to move the pebble onto the location L1 by choosing to play the edge labeled by $u2$. While $\mathtt{L}_2$ is a controllable predecessor of the set W_0 as in $\mathtt{L}_2$ Player 1 can move the pebble onto the location $\mathtt{L}_0 \in \{\mathtt{L}_0, \mathtt{L}_1\}$ by playing the action b, so $W_1 = \{\mathtt{L2}\}$. And clearly, $\mathsf{CPre}(W_1) = \varnothing$. So, there is no location in G from which Player 1 can ensure to stay within $\{\mathtt{L0}, \mathtt{L2}\}$ forever and Player 1 cannot win the game.

Let us now change the objective and consider $\mathsf{Obj} = \mathsf{Safe}_G(\{\mathtt{L}_0, \mathtt{L}_1, \mathtt{L}_2\})$. We start the computation with $W_0 = \{\mathtt{L}_0, \mathtt{L}_1, \mathtt{L}_2\}$, and compute $W_1 = W_0 \cap \mathsf{CPre}(W_0)$. All the edges leaving $\mathtt{L}_0$ reach a location in W_0 so $\mathtt{L}_0 \in \mathsf{CPre}(W_0)$, in $\mathtt{L}_1$ all edges of Player 2 and all edges of Player 1 labeled with a reach a location in W_0 so $\mathtt{L}_1 \in \mathsf{CPre}(W_0)$, and in $\mathtt{L}_2$ all edges of Player 2 and all edges of Player 1 labeled with b reach a location in W_0 so $\mathtt{L}_2 \in \mathsf{CPre}(W_0)$. The sequence of sets stabilizes as $W_1 = W_0$, and so Player 1 has a strategy to win the safety objective $\mathsf{Obj} = \mathsf{Safe}_G(\{\mathtt{L}_0, \mathtt{L}_1, \mathtt{L}_2\})$ from all locations in $\{\mathtt{L}_0, \mathtt{L}_1, \mathtt{L}_2\}$. □

Remark 1. The main drawback of the algorithms that we have outlined above is that they compute winning information about locations that are not necessarily reachable by an interaction between Player 1 and Player 2 from the initial location. In practice, that can deteriorate the performances of the algorithms. There are solutions to avoid that problem, see for example the on-the-fly algorithm of [2], but the description of those solutions goes beyond the objectives of this introduction.

1.5 Adding time to game structures

To add time to game structures, we adapt the syntax of timed automata as defined in Chapter XXX and partition discrete transitions as controllable and uncontrollable. A timed game automaton $G = (L, \ell_{\mathsf{init}}, X, \mathsf{Inv}, \mathsf{Act}_1, \mathsf{Act}_2, E)$ is a structure, where:

- L is a finite set of discrete locations and ℓ_{init} is the initial location of the timed game;
- X is a finite set of clocks, and we denote by $\mathsf{Constr}(X)$ the set of *clock constraints*, i.e. conjunctions of atomic constraints of the forms $x \sim c$ or $x - y \sim c$, where $c \in \mathbb{N}$ and $x, y \in X$;
- $\mathsf{Inv} : L \to \mathsf{Constr}(X)$ is a function that labels each location $\ell \in L$ with an invariant $\mathsf{Inv}(\ell)$ that restricts the possible values that clocks in X can take when the control of the automaton is in location ℓ,

- Act_1 are the actions of Player 1, Act_2 are the actions of Player 2 such that $\mathsf{Act}_1 \cap \mathsf{Act}_2 = \varnothing$, and $E \subseteq L \times (\mathsf{Act}_1 \cup \mathsf{Act}_2) \times \mathsf{Constr}(X) \times 2^X \times L$ is the set of discrete transitions of the timed game. A tuple $(\ell, \alpha, \phi, R, \ell') \in E$ is a transition that goes from location ℓ to location ℓ', that is labeled with action α (if $\alpha \in \mathsf{Act}_1$ then the transition is controllable, otherwise it is uncontrollable), with guard ϕ (the transition can be taken only if the values of clocks satisfy the guard), and reset set R (the clocks in the set R are reset when the transition is taken).

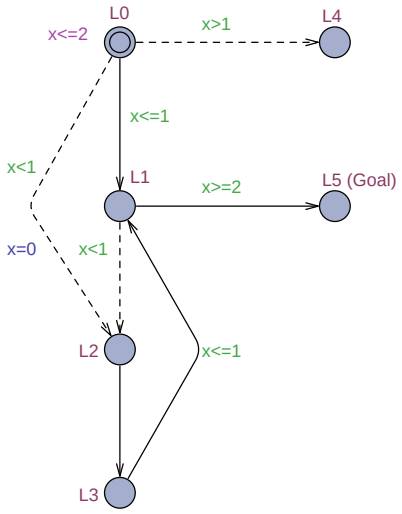


Fig. 1.4 An example of a timed game automaton

Example 8. An example of a timed game automaton is given in Fig. 1.4. The only syntactical difference with plain timed automata is induced by the partition of the alphabet of labels for the transitions: the transitions labeled with an element of Act_1 belong to Player 1, and the transitions labeled with an element of Act_2 belong to Player 2. As for untimed games, the edges controlled by Player 1 are depicted by plain edges, and the edges controlled by Player 2 are depicted as dashed edges.

A state of a timed automaton is a pair (ℓ, v), where ℓ is a location and v is a valuation for the clocks, i.e. a function $v : X \to \mathbb{R}_{\geq 0}$ that assigns to each clock $x \in X$ a positive real number $v(x)$. In a timed automaton, when the automaton is in a state (ℓ, v), time can elapse as long as it does not violate $\mathsf{Inv}(\ell)$ (the invariant that labels ℓ). For example in the timed game automaton of Fig. 1.4, from state $(\mathrm{L0}, v)$ with $v(x) = 1$, time can elapse for t time units if $1 + t \leq 2$, in that case state $(\mathrm{L0}, v')$ is reached with $v'(x) = v(x) + t$.

A transition $(\ell_1, a, \phi, R, \ell_2)$ can be taken in state (ℓ, v) whenever $\ell = \ell_1$, the guard ϕ is satisfied by v, which is denoted by $v \models \phi$, and the clock valuation $v[R := 0]$,

which maps a clock $x \in X \setminus R$ to $v(x)$, and a clock $x \in R$ to 0, is such that it satisfies $\mathsf{Inv}(\ell_2)$, i.e. $v[R := 0] \models \mathsf{Inv}(\ell_2)$. For instance, in the timed game automaton of Fig. 1.4, in state $(\ell_0, \frac{1}{2})$, Player 2 can take the uncontrollable transition to ℓ_2 as the guard on x is satisfied ($\frac{1}{2} < 1$.) The state that is reached after this transition is the pair $(\ell_2, 0)$ as the clock x is reset by this transition.

For a more systematic presentation of the semantics of timed automata, the reader is referred to Chapter XXX. In this section, we focus on intuitions and do not always give all the formal definitions.

1.5.1 Rounds in timed games

Remember that *untimed* two-player games are played for an infinite number of rounds. Each round is played as follows: Player 1 chooses one action $\alpha \in \mathsf{Act}_1$ among the actions that label the controllable transitions leaving the location where the pebble lieson , and then Player 2 moves the pebble by using a transition that is labeled either by α or by an action from Act_2 (an uncontrollable transition.)

In timed games, we additionally need to know at what time Player 1 wants to play. So in addition to an action to play, Player 1 chooses a delay t. Then given a pair (t, α), Player 2 either decides to wait for t time units and to take a transition that is labeled with the letter $\alpha \in \mathsf{Act}_1$, and for which the guard on the transition is satisfied, or Player 2 decides to wait for a delay of $t' \leq t$ and use a transition labeled by an action from Act_2, and for which the guard evaluates to true.

Example 9. Let us consider the timed game automaton of Fig. 1.4. As in this example, there are at most one controllable and one uncontrollable transition out of each location, we did not give names to the transitions. This example is a timed game automaton with a reachability objective for Player 1: her objective is to reach the location labeled with goal. Initially the pebble lies on L0 and the value of the clock x is equal to 0. Let us assume that Player 1 proposes to wait exactly for 1 time unit and to take the transition that leads to L1. In this case, the two following scenarios are possible. Either Player 2 lets time elapse for at least 1 time unit, the value of the clock x is then equal to 1, and the pebble can be moved on location L1 as proposed by Player 1 (indeed, the guard $x \leq 1$ is satisfied.) Or, Player 2 decides to wait for $t < 1$ time units, and to move the pebble to location L2 using the uncontrollable edge from L0 to L2. Again, this is possible because after waiting for $t < 1$ time units, the value of x is less than 1 time unit, and so the guard $x < 1$ on the transition from L0 to L2 is satisfied.

Assume for now that Player 2 follows the second scenario. The pebble is now lying on L2 and the value of clock x is equal to 0 (as it has been reset when moving the pebble using the transition from L0 to L2.) From that position, let us assume that Player 1 chooses to wait for $\frac{1}{2}$ time units and proposes to move to location L3. As there is no alternative for Player 2, time elapses for $\frac{1}{2}$ time units, and the pebble is moved from L2 to L3.

From there, Player 1 chooses to wait for $\frac{1}{2}$ time units and proposes to move the pebble to L1. Again, as there is no alternative for Player 2, time elapses for $\frac{1}{2}$ time units and the pebble is moved from L3 to L1. When the pebble arrives on L1, the value of the clock x is equal to $\frac{1}{2}+\frac{1}{2}=1$. Then Player 1 chooses to wait say for $1\frac{1}{4}$ time units and to move the pebble to location goal. This is a valid move as the value of x is then equal to $1+1\frac{1}{4}=2\frac{1}{4}$ and so the guard $x \geq 2$ is satisfied, again as Player 2 has no other alternatives, the pebble is moved on location goal, and the play is winning for Player 1. □

When moving the pebble according to the rules defined above, a *timed play* of the form $(\ell_0,v_0) \xrightarrow{(t_0,e_0)} (\ell_1,v_1) \xrightarrow{(t_1,e_1)} \dots \xrightarrow{(t_{n-1},e_{n-1})} (\ell_n,v_n) \xrightarrow{(t_n,e_n)} \dots$ is generated by the interaction between the two players. In this timed play, each (t_i,e_i) specifies the time that has elapsed and the transition that has been taken during the round i.

As for untimed games, objectives are defined by a set of discrete locations $T \subseteq L$ of the automaton that Player 1 wants to reach for reachability games, or by a set of discrete locations $S \subseteq L$ in which Player 1 wants to stay in for safety games. For reachability and safety objectives, it can be shown that Player 1 has a winning strategy if and only if she has a winning memoryless strategy [3]. For timed games, a *memoryless strategy* is a function $\lambda_1 : L \times \mathbb{R}_{\geq 0}{}^{|X|} \to \mathbb{R}_{\geq 0} \times \mathsf{Act}_1$ that specifies, given the current state of the game (ℓ,v), the time $t \in \mathbb{R}_{\geq 0}$ to wait and the action $\alpha \in \mathsf{Act}_1$ to play.

1.6 Solving two-player timed games

We have seen that, in the case of untimed games, reachability and safety objectives can be solved using a notion of *controllable predecessors*. This notion can be extended to timed games. Again, we do not formalize all the details here but we give enough intuition so that the reader can understand the main ideas behind the algorithms for solving timed games.

Intuitively, a state (ℓ,v) is a controllable predecessor of a set of states $T = \{(\ell_0,v_0),(\ell_1,v_1),\dots,(l_n,v_n),\dots\}$, if there exist $\alpha \in \mathsf{Act}_1$ and a delay $t \in \mathbb{R}_{\geq 0}$ such that the following four conditions hold:

1. for all delays t', $0 \leq t' \leq t$, $v+t' \models \mathsf{Inv}(\ell)$, i.e. time can elapse from (ℓ,v) for t time unit without violating the invariant labeling ℓ;
2. there exists a transition $e=(\ell,\alpha,\phi,R,\ell')$ such that $v+t \models \phi$ and $v+t[R:=0] \models \mathsf{Inv}(\ell')$, i.e. there is a transition labelled with α that can be taken after t time units;
3. for all transitions $e=(\ell,\alpha,\phi,R,\ell')$ such that $v+t$ satisfies ϕ, $(\ell',v+t[R:=0])$ belongs to T, i.e. any choice of a transition labelled with α taken after t time units leads to T;
4. for all transitions $e=(\ell,u,\phi,R,\ell')$ and delays t' such that $0 \leq t' \leq t$, $u \in Act_2$, and $v+t'$ satisfies ϕ, then $(\ell',v+t'[R:=0])$ belongs to T, i.e. any uncontrollable transition that can be taken within t time units leads to T.

The sets of states that we have to handle are infinite, so they cannot be represented in extension. We need a symbolic data structure able to represent infinite sets. Those sets can be represented symbolically using formulas in an adequate constraint language. All sets manipulated during the computation of the timed controllable predecessors are representable by union of clock constraints. To illustrate the use of clock constraints and how the computation of the controllable predecessor in the timed setting is done, we consider our running example of Fig. 1.4.

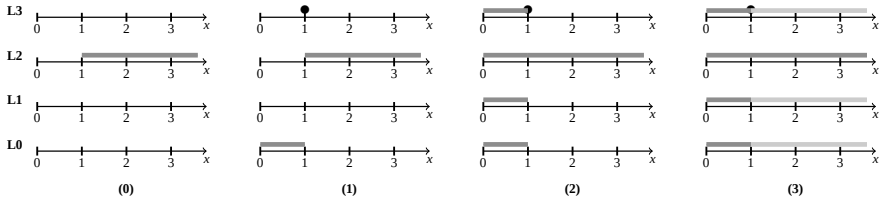


Fig. 1.5 Computation of the timed controllable states.

Example 10. The computation of the set of winning states is depicted in Fig. 1.5. The first part of the picture, marked (1), depicts the set of states of the form $(\mathtt{L1}, v)$ with $v \geq 1$. All those states are winning in 1 step because when $x \geq 1$, the uncontrollable transition from L1 to L2 cannot be taken by Player 2 (as it is guarded by $x < 1$), and by waiting until clock x reaches a value equal to or greater than 2, Player 1 can move the pebble from L1 to location goal. The part marked (2) of the picture depicts the set of states that are winning in at most 2 discrete steps. The states that have been added are controllable predecessors of the states that are winning in 1 step. First, let us consider $(\mathtt{L0}, v)$ with $v(x) = 1$. This state is winning as, on the one hand, none of the uncontrollable transitions is enabled in this state, and on the other hand, the controllable transition from L0 to L1 is enabled, and when it is taken the game reaches a winning state (in 1 step.) Second, consider the set of states $(\mathtt{L3}, v)$ with $v(x) \leq 1$. From all those states, Player 1 can wait until $x = 1$ and then she can take the controllable transition to L2, reaching a set of winning states in 1 step. The states depicted in part (3) and (4) are computed in a similar manner. □

1.7 The Chinese juggler control problem in UPPAAL-TIGA

UPPAAL-TIGA is a tool developed at Aalborg University. It handles timed game automata as presented in the previous section. The tool can be downloaded from `http://www.cs.aau.dk/~adavid/tiga/download.html`. We refer the reader to the user manual for details about the features and the usage of UPPAAL-TIGA in practice. In this section, we show how to model the Chinese juggler control problem with timed game automata. We use screenshots from the tool to illustrate

Kim, can we add this on the tiga web site ???

its user interface. The interested reader can download the UPPAAL-TIGA model of our running example from `http://...` .

Note that for the sequel, we assume that the reader is familiar with notation of the UPPAAL tool as described in Chapter YY. The models that are used here are game extensions of the UPPAAL models, we make it clear what are those extensions in the sequel.

1.7.1 Modeling of the components

A timed game in UPPAAL-TIGA is modeled compositionally by defining timed game automata that specify the behavior of the components of the system. This modeling approach is similar to the one used for regular models in UPPAAL (see Chapter YY for additional material on compositional modeling). As in UPPAAL models, components in UPPAAL-TIGA synchronize using shared events (implemented by channels). For the rest of this section, we assume that the reader is familiar with this modeling paradigm.

Fig. 1.6 shows the timed automaton model for `Plate` $i \in \{1,2\}$. The timed game automaton has the set of locations $\{$`Stable`, `Spinning`, `Longspinning`, `Crashed`$\}$. The location `Stable` intends to model the situation when the plate is stable, the location `Crashed` models the situation when the plate has crashed, `Spinning` models the situation when the juggler does spin the plate spin for a time $t \leq$ `STABSHORT` seconds (where `STABSHORT` is a constant equal to 2), and `Longspinning` when the juggler does spin the plate for more than `STABSHORT` seconds.

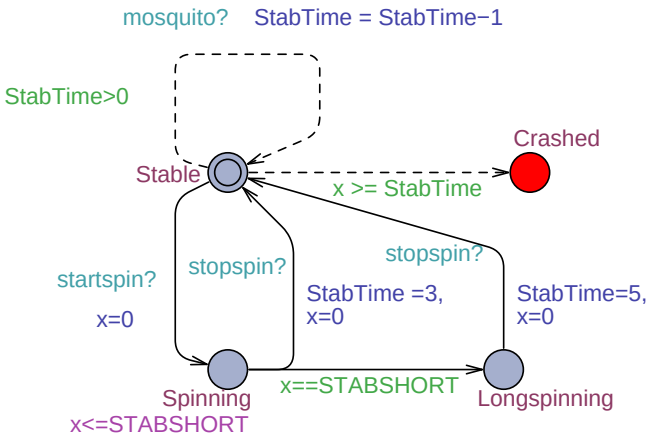


Fig. 1.6 A model for the plate.

The automaton uses one clock x. The use of x is twofold. First, when the control is in location `Stable`, the variable x records the time elapsed since the plate was last spun by the juggler. When the control is in `Spinning` or `Longspinning`, x records the time elapsed since the plate has last been spun under the impulsion of the juggler. Let us now have a look at the transitions between control locations.

First, we consider the uncontrollable transitions. There are two uncontrollable transitions that leave `Stable`. The self loop is taken whenever the mosquito touches the plate (this is ensured by the synchronization on the event `mosquito?`). The effect is to substract value 1 from the integer variable `StabTime` that models the length of the time interval during which the plate is guaranteed to stay stable without being spun by the juggler. This can be done only if the guard $\texttt{StabTime} > 0$ is true (making sure that the value of `StabTime` cannot become negative.)

The uncontrollable transition going from `Stable` to `Crashed` can be taken (by Player 2) whenever the value of the clock x exceeds the time for which the plate is guaranteed to be stable (since the last time it has been spun by the juggler.) As this transition is uncontrollable, Player 2 can decide to take it at any time when the guard is true. Player 2 may not take the transition immediately when the guard becomes true but we cannot rely on this: that is why it is an uncontrollable transition in our model.

Second, we consider the controllable transitions. The transition between locations `Spinning` and `Longspinning` is taken exactly when the value of x is equal to `STABSHORT`. It accounts for the fact that the juggler is spinning the plate for an interval of more than `STABSHORT` seconds. In fact, the behavior of this transition is deterministic and so it could have been defined as uncontrollable, that would not make any difference. The other three controllable transitions are related to actions controlled by the juggler. When the plate is `Stable`, the juggler can decide to give it more spinning by emitting the event `startspin!`. This has the effect to trigger this transition (reception of the event `startspin!`) and to move the control to location `Spinning`. The control leaves the location `Spinning`:

- either because the juggler has decided to stop spinning (event `stopspin?`) before `STABSHORT` seconds, in that case, the control moves back to location `Stable`, the clock x is reset, and the interval for which the plate is guaranteed to be stable is 3 seconds (update `StabTime=3`),
- or because the juggler has spun the plate for `STABSHORT` seconds, and the control moves to `Longspinning`. This later location is left when the event `stopspin?` occurs, in that case the control moves back to `Stable` and the plate is guaranteed to be stable for 5 seconds (update `StabTime=5`).

This template timed game automaton is instantiated twice, one time for `Plate 1` and one time for `Plate 2`.

We can now have a look at the other components of our model. Fig. 1.7 depicts a model of the mosquito. The mosquito can at any time touch one of the two plates provided that he has not touched a plate within the last D times units (this is forced by the guard $y \geq D$). This last constraint is enforced using the clock y which is reset each time a plate is touched. The self-loop is labeled with the event `mosquito!`

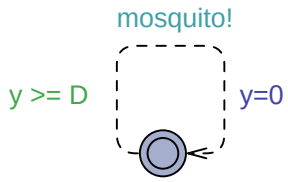


Fig. 1.7 A model for the mosquito

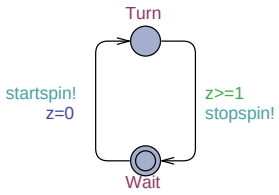


Fig. 1.8 A model for the juggler

which is either received by Plate 1 or Plate 2. The transition is uncontrollable as it belongs to the mosquito and not to the controller that we want to synthesize.

Finally, the juggler is modeled by the timed automaton given in Fig. 1.8. The juggler can be in two different states that are modeled by two locations: `Wait` models the situation when the juggler does not spin any of the two plates, `Turn` models the situation when the juggler spins one of the plates. The events `startspin!` and `stopspin!` are synchronized with either Plate 1 or Plate 2. Clock z is used to express that the juggler should spin a plate for at least 1 time unit.

1.7.2 Analysis of the model

We can now analyze the model of the Chinese Juggler presented above with the tool UPPAAL-TIGA. We want to determine if the Juggler has a strategy to win the timed game for the safety objective ‘none of the two plates ever crashes””. This control objective is expressed by the following expression in the UPPAAL-TIGA syntax:

```
control: A[] not (Plate1.Crashed or Plate2.Crashed)
```

This formula asks to find a control strategy (keyword `control`) for the juggler such that on all resulting plays (modality `A`), it is always the case (modality [ ]) that `(Plate1.Crashed or Plate2.Crashed)` is false.

If we impose to the mosquito to stay away from the two plates for at least $D = 2$ seconds after touching one of the plates, then the Juggler has a strategy to win. UPPAAL-TIGA is able to determine that property, and furthermore, the tool also synthesizes a winning strategy. The strategy that the tool synthesizes is as follows:

.... Kim, can you provide a picture of the winning strategy for the parameters as described above ?

Now, if we set $D = 1$, then the Juggler does not have a strategy to win as the mosquito can act very fast.

1.8 Conclusion

In this chapter, we have introduced the basic concepts and algorithmic ideas that underly the automatic synthesis of discrete and timed controllers for systems modeled by game automata and timed game automata. We have shown that the game

metaphor is natural to model control problems. Even if those ideas are relatively recent, they have been implemented into the tool UPPAAL-TIGA and they can be applied to interesting case studies.

In the next chapter, we show how to use UPPAAL-TIGA to automatically synthesize a controller to regulate a pressure accumulator and to optimize its energy consumption.

References

1. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *CAV - International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer, 2007.
2. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR - International Conference Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.
3. Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS - Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer-Verlag, 1995.
4. Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL - Annual Symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.
5. Peter J. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

Chapter 1
Timed Controller Synthesis: An Industrial Case Study

Franck Cassez, Kim Larsen, Jean-François Raskin, and Pierre-Alain Reynier

1.1 Introduction

The design of controllers for embedded systems is a difficult engineering task. Controllers have to enforce properties like safety properties (e.g. "nothing bad will happen"), or reachability properties (e.g. "something good will happen"), and ideally they should do that in an efficient way, e.g. consume the least possible amount of energy. The foundations of automatic synthesis of discrete and timed controllers have been presented in the preceding chapter 1. In this chapter, we illustrate the application of these approaches with an industrial case study provided by the HYDAC ELECTRONIC GMBH company in the context of the European research project Quasimodo [**?**]. We present in the sequel how to use (in a systematic way) the tool UPPAAL-TIGA [**?**] (see chapter 1 for an introduction to the tool) for the synthesis, together with tools for the verification and the simulation of a provable correct and near optimal controller for real industrial applications.

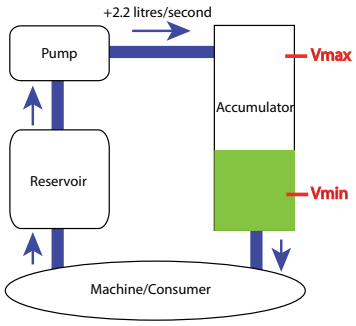


Fig. 1.1 Overview of the system.

The system to be controlled is depicted in Fig. 1.1 and is composed of:

1. a machine which consumes oil,
2. a reservoir containing oil,
3. an accumulator containing oil and a fixed amount of gas in order to put the oil under pressure, and
4. a pump.

When the system is operating, the machine consumes oil under pressure out of the accumulator. The level of the oil, and so the pressure within the accumulator (the amount of gas being constant), can be controlled using the pump to introduce additional oil in the accumulator (increasing the gas pressure). The control objective is twofold: first the level of oil into the accumulator (and so the gas pressure) which is controlled using the pump must be maintained within a safe interval; second the controller should try to minimize the level of oil such that the accumulated energy in the system is kept minimal (a lower level of oil, and thus a lower gas pressure, reduces the wear on the pump and minimizes energy consumption). The maximum output rate of the pump is 2.2litre/sec, whereas the maximum rate of the consumer is 2.5litre/sec: proper operation of the pump should thus anticipate the maximum rate of the machine and pump oil in in advance to ensure that the pressure stays with the given bounds.

To solve the HYDAC ELECTRONIC GMBH control problem, we use three complementary tools for three different purposes: UPPAAL-TIGA for synthesis, the tool PHAVER [**?**, **?**] for verification, and SIMULINK [**?**] for simulation. In this chapter, we are mainly interested in the synthesis step. For this phase, we show how to construct a (game) model of this case study which has the following properties:

- it is simple enough to be solved automatically using algorithmic methods implemented into UPPAAL-TIGA;
- it ensures that the synthesized controllers can be easily implemented, because it is robust.

To meet those two requirements, we consider an idealized version of the environment in which the controller is embedded, but we put additional constraints into the winning objective of the controller that ensure the robustness of winning strategies. As the winning strategies are computed on a simplified model of the system, we show how to embed automatically the synthesized strategies into a more detailed model of the environment, and how to automatically prove their correctness using the tool PHAVER [**?**, **?**] for analyzing hybrid systems. While the verification model allows us to establish correctness of the controller that is obtained automatically using UPPAAL-TIGA, it does not allow us to learn its expected performance in an environment where noise is not completely antagonist but follows some probabilistic rules. For this kind of analysis, we consider a third model of the environment and we analyze the performance of our synthesized controller using SIMULINK.

To show the advantages of our approach, we compare the performances of the controller we have automatically synthesized with two other control strategies. The first control strategy is a simple two-point control strategy where the pump is turned on when the volume of oil reaches a floor value and turned off when the volume of

oil reaches a ceiling value. The second control strategy is a strategy designed by the engineers at HYDAC ELECTRONIC GMBH with the help of SIMULINK.

Structure of the chapter

In section 1.2, we present in more details the HYDAC ELECTRONIC GMBH control problem. In section 1.3, we present our construction of a suitable abstract model of the system, and the strategy we have obtained using the synthesis algorithm of UPPAAL-TIGA. In section 1.4, we embed the controllers into a continuous hybrid model of the environment and use the tool PHAVER to verify their correctness and robustness: we prove that strategies obtained using UPPAAL-TIGA are indeed correct and robust. In section 1.5, we analyze and compare the performances in term of mean volume of the three controllers using SIMULINK.

1.2 The Oil Pump Control Problem

In this section, we describe the components of the HYDAC ELECTRONIC GMBH case study and describe the different constraints they must satisfy. In addition, we provide a first model of these components, using hybrid automata notations. Though we will not use these models in the sequel to synthesize controllers (synthesis on real hybrid systems is very costly and beyond the scope of any tool), they offer a very precise specification of the system. Then we explain the control objectives for the system to design.

1.2.1 The Machine

The oil consumption of the machine is cyclic. One cycle of consumption, as given by HYDAC ELECTRONIC GMBH, is depicted in Fig. 1.5. Each period of consumption is characterized by a rate of consumption m_r (expressed as a number of litres per second), a time of beginning, and a duration. We assume that the cycle is known *a priori*: we do not consider the problem of identifying the cycle (which can be performed as a pre-processing step). At time 2, the rate of the machine goes to $1.2l/s$ for two seconds. From 8 to 10 it is 1.2 again and from 10 to 12 it goes up to 2.5 (which is more than the maximal output of the pump). From 14 to 16 it is 1.7 and from 16 to 18 it is 0.5. Even if the consumption is cyclic and known in advance, the rate is subject to *noise*: if the mean consumption for a period is $c\ l/s$ (with $c > 0$), in reality it always lies within that period in the interval $[c-\varepsilon, c+\varepsilon]$, where ε is fixed to $0.1\ l/s$. This property is noted F.

Fig. 1.2 The Machine

Fig. 1.3 The Accumulator

Fig. 1.4 Model of the pump

Fig. 1.5 Model of the scheduler

To model the machine, we use a timed automaton with 2 variables. The discrete variable m_r models the consumption rate of the machine, and the clock variable t is used to measure time within a cycle. The variable m_r is shared with the model of the accumulator. The timed automaton is given in Fig. 1.2. The noise on the rate of consumption is modeled in the model for the accumulator (see Fig. 1.3). We indicate for each location of the timed automaton its invariant in square brackets. For instance, one can stay in location i_1 as long as the value of the clock t is less than or equal to 2.

1.2.2 The Pump

The pump is either *On* or *Off*, and we assume it is initially *Off*. The operation of the pump must respect the following *latency* constraint: there must always be two seconds between any change of state of the pump, i.e. if it is turned *On* (respectively *Off*) at time t, it must stay *On* (respectively *Off*) at least until time $t+2$: we note P_1 this property. When it is *On*, its *output* is equal to $2.2l/s$. We model the pump with a two-state timed automaton given in Fig. 1.4 with two variables. The discrete variable p_r models the pumping rate of oil of the pump, and is shared with the accumulator. The clock z ensures that 2 t.u. have elapsed between two switches.

1.2.3 The Accumulator.

To model the behavior of the accumulator, we use a one-state hybrid automaton given in Fig. 1.3 that uses four variables. The variable v models the volume of oil within the accumulator, its evolution depends on the value of the variables m_r (the rate of consumption depending of the machine) and p_r (the rate of incoming oil from the pump). To model the imprecision on the rate of the consumption of the machine, the dynamics of the volume also depends on the parameter ε and is naturally given by the differential inclusion $dv/dt \in [p_r - m_r^-(\varepsilon), p_r - m_r^+(\varepsilon)]$ with $m_r^{\bowtie}(x) = m_r \bowtie x$ if $m_r > 0$ and m_r otherwise. The variable *Vacc* models the accumulated volume of oil along time in the accumulator. It is initially equal to 0 and its dynamic is naturally defined by the equation $dVacc/dt = v$.

1.2.4 The Control Problem

The controller must operate the pump (switch it *on* and *off*, respecting the latency constraint) to ensure the two main requirements:

- (R_1): the level of oil $v(t)$ at time t (measured in litres) into the accumulator must always stay within two *safety* bounds $[V_{min}; V_{max}]$, in the sequel $V_{min} = 4.9l$ and $V_{max} = 25.1l$;
- (R_2): a large amount of oil in the accumulator implies a high pressure of gas in the accumulator. This requires more energy from the pump to fill in the accumulator and also speeds up the wear of the machine. This is why the level of oil should be kept minimal during operation, in the sense that $\int_{t=0}^{t=T} v(t)dt$, that is $V_{acc}(T)$, is minimal for a given operation period T.

While (R_1) is a *safety requirement* and so must never be violated by any controller, (R_2) is an *optimality* requirement and will be used to compare different controllers.

Note that as the power of the pump is not always larger than the demand of the machine during one period of consumption (see Fig. 1.5 between 10 and 12), some extra amount of oil must be present in the accumulator before that period of consumption to ensure that the amount of oil stays above V_{min} (requirement R_1) is not violated[1].

1.2.5 Additional Requirements on the Controller

When designing a controller, we must decide what are the possible actions that the controller can take. Here are some considerations about that. First, as the consumptions of the machine may deviate slightly from the ideal values (this is called *noise*), it is necessary to allow the controller to check periodically the level of oil in the accumulator (as it is not predictable in the long run). Second, as the consumption of the machine has a cyclic behavior, the controller should use this information to optimize the level of oil. So, it is natural to allow the controller to take control decisions at predefined instants during the cycle. Finally, we want a robust solution in the sense that if the controller has to turn the pump *on* (or *off*) at time t, it can do it a little before or after, that is at time $t \pm \Delta$ for a small Δ without impairing safety. This robustness requirement will be taken into account in the synthesis and verification phases described later.

[1] It might be too late to switch the pump on when the volume reaches V_{min} and so the controller may have to anticipate.

1.2.6 Two existing solutions

In the next sections, we will show how to use synthesis algorithms implemented in UPPAAL-TIGA to obtain a simple but still efficient controller for the oil pump. This controller will be compared to two other solutions that have been previously considered by the HYDAC ELECTRONIC GMBH company.

The first one is called the *Bang-Bang controller*. Using the sensor for oil volume in the accumulator, the Bang-Bang controller turns *on* the pump when a *floor* volume value V_1 is reached and turns *off* the pump when a *ceiling* volume value V_2 is reached. The Bang-Bang controller is thus a simple two-point controller, but it does not exploit the timing information about the consumption periods within a cycle.

To obtain better performances in term of energy consumption, engineers at HYDAC ELECTRONIC GMBH have designed a controller that exploit this timing. This second controller is called the *Smart controller* and works as follows [**?**]: in the first cycle the Bang-Bang controller is used and the pressure $p(t)$ is measured, the corresponding volume $v(t)$ is computed and recorded every 10ms. According to the sampled values $v(t)$ computed in the initial cycle, an optimization procedure computes the points at which to start/stop the pump on the next cycle (this optimization procedure was given to us in the form of a C code executable into SIMULINK; unfortunately we do not have a mathematical specification of it). On this next cycle the values $p(t)$ are again recorded every 10ms which is the basis for the computation of the start/stop commands for the next cycle and so on. If the pressure leaves a predefined safety interval, the Bang-Bang controller is launched again. Though simulations of SIMULINK models developed by HYDAC ELECTRONIC GMBH reveal no unsafe behaviour, the engineers have not been able to verify neither its correctness nor its robustness. As we will see later, this strategy (we use the switching points in time obtained with SIMULINK when the C code is run) is not safe in the long run in presence of noise.

1.3 The UPPAAL-TIGA Model for Controller Synthesis

The hybrid automaton model presented in the previous section can be interpreted as a game in which the controller only supervises the pump. In this section, we show how to automatically synthesize, from a game model of the system and using UPPAAL-TIGA, an efficient controller for the Hydac case study. UPPAAL-TIGA is a recent extension of the tool UPPAAL which is able to solve timed games (see chapter 1 for a presentation of the tool).

1.3.1 Game Models of Control Problems

While modeling control problems with games is very *natural* and *appealing*, we must keep in mind several important aspects. First, solving timed games is computationally hard, so we should aim at game models that are sufficiently abstract. Second, when modeling a system with a game model, we must also be careful about the information that is available to each player in the model. The current version of UPPAAL-TIGA offers *games of perfect information* (see [**?**] for steps towards games for imperfect information into UPPAAL-TIGA.) In games of perfect information, the two players have access to the full description of the state of the system. For simple objectives like safety or reachability, the strategies of the players are functions from states to actions. To follow such strategies, the implementation of the controller must have access to the information contained in the states of the model. In practice, this information is acquired using sensors, timers, etc.

1.3.2 The UPPAAL-TIGA *Model*

We describe in the next paragraphs how we have obtained our game model for the hybrid automaton of the HYDAC ELECTRONIC GMBH case study. First, to keep the game model simple enough and to remain in a decidable framework[2], we have designed a model which: (a) considers one cycle of consumption; (b) uses an abstract model of the fluctuations of the rate; (c) uses a discretization of the dynamics within the system. Note that since the discretization impacts both the controller and the environment, it is neither an over- nor an under-approximation of the hybrid game model and thus we can not deduce directly the correctness of our controllers. However, our methodology includes a verification step based on PHAVER which allows us to prove this correctness. Second, to make sure that the winning strategies that will be computed by UPPAAL-TIGA are implementable, the states of our game model only contain the following information, which can be made available to an implementation:

- the volume of oil at the beginning of the cycle; we thus only measure the oil once per cycle, leading to more simple controllers.
- the ideal volume as predicted by the consumption period in the cycle;
- the current time within the cycle;
- the state of the pump (*on* or *off*).

Third, to ensure robustness of our strategies, *i.e.* that their implementations are correct under imprecision on measures of volume or time, we consider some margin parameter m which roughly represents how much the volume can deviate because of these imprecision in the measure. We will consider values in range $[0.1; 0.4] litre$ for measurement imprecision.

[2] The existence of winning strategies for enriched timed games with extra cost variables in undecidable, see [**?**].

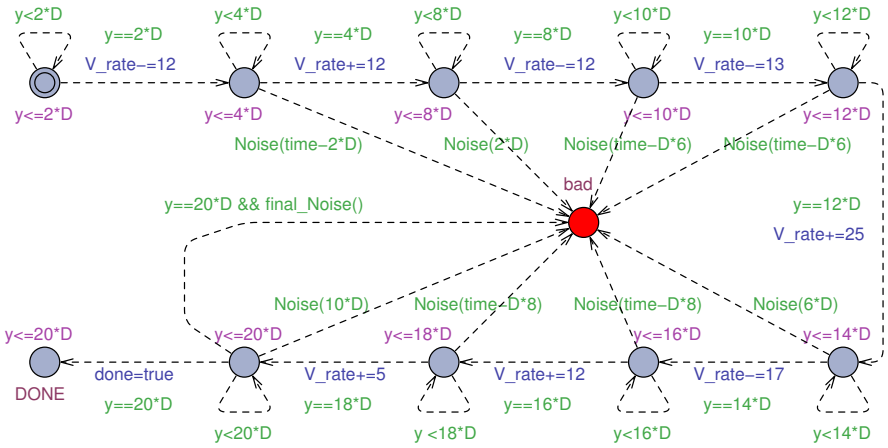


Fig. 1.6 Model of the cyclic consumption of the machine

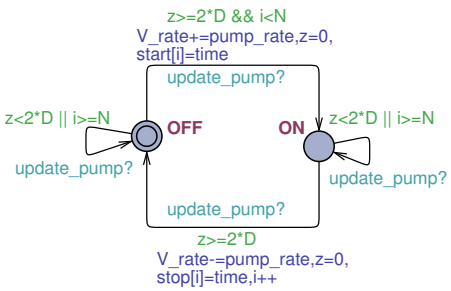


Fig. 1.7 Model of the pump

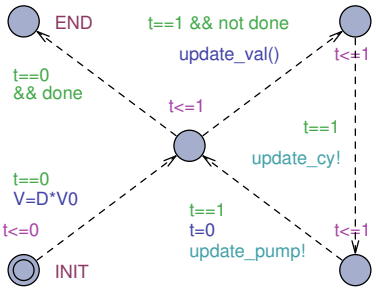


Fig. 1.8 Model of the scheduler

Global Variables

First, we discretize the time w.r.t. ratio stored in variable D, such that D time units represent one second. Second, we represent the current volume of oil by the variable V. We consider a precision of $0.1l$ and thus multiply the value of the volume by 10 to use integers. This volume evolves according to a rate stored in variable V_rate and the accumulated volume is stored in the variable V_acc[3]. Finally, we also use an integer variable time which measures the global time since the beginning of the cycle.

The Model of the Machine

The model for the behaviour of the machine is represented on Fig. 1.6. Note that all the transitions are uncontrollable (represented by dashed arrows). The construction

[3] To avoid integers divisions, we multiply all these values by D.

of the nodes (except the middle one labelled bad) follows easily from the cyclic definition of the consumption of the machine. When a time at which the rate of consumption changes is reached, we simply update the value of the variable V_rate. The additional central node called bad is used to model the uncertainty on the value of V due to the fluctuations of the consumption of the machine. This noise is not related with the parameter m introduced previously as it concerns some possible fluctuations in the consumption rate of the machine around its nominal rate. The function Noise (Fig. 1.9) checks whether the value of V, if modified by these fluctuations in the consumption rate of the machine, may be outside the safe interval $[V_{min}+0.1, V_{max}-0.1]$ [4]. The function final_Noise (Fig. 1.9) checks the same but for the volume obtained at the end of cycle and against the interval represented by V1F and V2F. Note that this modelling allows in some sense to perform partial observation using a tool for games of perfect information. Indeed, the natural modelling would modify at each step the actual value of the variable V and the strategies would then be aware of the amount of fluctuations. In our model the ideal value of V is predictable because it directly depends on the current time and from the point of view of the controller it does not give any information about the fluctuation.

The Model of the Pump

The model for the pump is represented on Fig. 1.7 and is very similar to the timed automaton given on Fig. 1.4. Note that the transitions are all controllable (plain arrows) and that we impose a bit more than P_1 as we require that 2 seconds have elapsed at the beginning of the cycle before switching on the pump. Moreover, an additional integer variable i is used to count how many times the pump has been started on. We use parameter N to bound this number of activations, which is set to 2 in the following. Note also that the time points of activation/deactivation of the pump are stored in two vectors start and stop.

```
bool Noise(int s){
// s is the number of t.u. of consumption
 return (V-s<(Vmin+1)*D|V+s>(Vmax-1)*D);}

bool final_Noise(){
// 20*D t.u. of consumption in 1 cycle
 return (V-20*D<V1F*D|V+20*D>V2F*D);}
void update_val(){
  int V_pred = V;
  time++;
  V+=V_rate;
  V_acc+=V+V_pred;
}
```

Fig. 1.9 Functions embedded in UppAal Tiga models

[4] For robustness, we restrain safety constraints to $m = 0.1\ l$.

The Model of the Scheduler

We use a third automaton represented on Fig. 1.8 to schedule the composition. Initially it sets the value of the volume to V0 and then it repeats the following actions: it first updates the global variables V, V_acc and time through function update_val. Then the scheduling is performed using the two channels update_cy[5] and update_pump. When the end of the cycle of the machine is reached, the corresponding automaton sets the Boolean variable done to true, which forces the scheduler to go to location END.

Composition

We denote by $\mathscr{A}$ the automaton obtained by the composition of the three automata described above. We consider as parameters the initial value of the volume, say V_0, and the target interval I_2, corresponding to V1F and V2F, and write $\mathscr{A}(V_0, I_2)$ the composed system.

1.3.3 Global Approach for Synthesis

Even if the game model that we consider is abstract and restricted to one cycle, note that our modelling enforces the constraints expressed in section 1.2. Indeed, R_1 is enforced through function Noise, F is handled through the two functions Noise and final_Noise, and P_1 is expressed explicitly in the model of the pump. To extend our analysis from one cycle to any number of cycles, and to optimize objective R_2, we formulate the following control objective (for some fixed margin $m \in \mathbb{Q}_{>0}$) :

Property $(*)$: Find some interval $I_1 = [V_1, V_2] \subseteq [4.9; 25.1]$ such that

(i) I_1 is *m-stable*: from all initial volume $V_0 \in I_1$, there exists a strategy for the controller to ensure that, whatever the fluctuations on the consumption, the value of the volume is always between 5 l and 25 l and the volume at the end of the cycle is within interval $I_2 = [V_1 + m, V_2 - m]$,

(ii) I_1 is *optimal* among m-stable intervals: the supremum, over $V_0 \in I_1$ and over the strategies satisfying (i), of the accumulated volume is minimal.

The strategies that fulfill that control objective have a nice *inductive property*: as the value of the volume of oil at the end of the cycle is ensured to be within I_2, and $I_2 \subset I_1$ if $m > 0$, the strategies computed on our one cycle model can be safely repeated as many times as desired. Moreover, the choice of the margin parameter m will be done so as to ensure robustness. We will verify in PHAVER that even in presence of imprecision, the final volume, if it does not belong to I_2, belongs to I_1:

[5] We did not represent this synchronization on Fig. 1.6 to ease the reading.

this is the reason why we fix a strict-subinterval of I_1 as a target in the synthesis phase.

We now describe a procedure to compute an interval verifying Property $(*)$, and the associated strategies. We proceed as follows[6]:

1. For each $V_0 \in [4.9;25.1]$, and target final interval $J \subseteq [4.9;25.1]$, compute (by a binary search) the minimal accumulated volume $\mathsf{Score}(V_0,J)$ that can be guaranteed. This value $\mathsf{Score}(V_0,J)$ is

   $$\min\{K \in \mathbb{N} \mid \mathscr{A}(V_0,J) \models \texttt{control: A<> Sched.END and V\_acc<=K}\}$$

2. Compute an interval $I_1 \subseteq [4.9;25.1]$ such that, for $I_2 = [V_1+m, V_2-m]$:
   a. $\forall V_0 \in I_1$, $\mathscr{A}(V_0,I_2) \models \texttt{control: A<> Sched.END}$
   b. the value $\mathsf{Score}(I_1) = \max\{\mathsf{Score}(V_0,I_2) \mid V_0 \in I_1\}$ is minimal.

3. For each $V_0 \in I_1$, compute a control strategy $\mathscr{S}(V_0)$ for the control objective `A<> Sched.END and V_acc<=K` with `K` set to $\mathsf{Score}(V_0,I_2)$. This strategy is defined by four dates of start/stop of the pump [7] and, by definition of $\mathsf{Score}(V_0,I_2)$, minimizes the accumulated volume.

It is worth noticing that the value Score is computed using the variable V_acc which is deduced from intermediary values of variable V. Since V corresponds to the value of the volume with no noise, V_acc represents the *mean value* of the accumulated volume for a given execution.

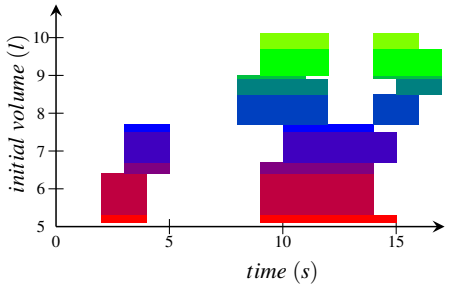


Fig. 1.10 For a margin parameter $m = 0.4l$ and a granularity of 1 (D=1 in the UPPAAL-TIGA model), we obtain as optimal stable interval the interval $I_1 = [5.1,10]$. The corresponding optimal strategies are represented opposite.

Results.

For a margin $m = 0.4l$ and a granularity of 1 (D=1 in the UPPAAL-TIGA model), we obtain as optimal stable interval the interval $I_1 = [5.1,10]$. The corresponding optimal strategies are represented on Fig. 1.10. For each value of the initial volume

[6] Control objectives are formulated as "control: P" following UPPAAL-TIGA syntax, where P is a TCTL formula specifying either a safety property `A[]` ϕ or a liveness property `A<>` ϕ.

[7] It is easy to obtain these times using the vectors start and stop of the pump.

in the interval I_1, the corresponding period of activation of the pump is represented. We have represented volumes which share the same strategy in the same color. For the 50 initial possible values of volume, we obtain 10 different strategies (first row of Table 1.1). The overall strategy we synthesize thus measures the volume just once at the beginning of each cycle and plays the corresponding "local strategy" until the beginning of next cycle.

Gran.	Margin	Stable interval	Nb of strategies	Mean volume
1	4	$[5.1, 10]$	10	8.45
1	3	$[5.1, 9.8]$	10	8.35
1	2	$[5.1, 9.6]$	9	8.25
1	1	$[5.1, 9.4]$	9	8.2
2	4	$[5.1, 8.9]$	14	8.05
2	3	$[5.1, 8.7]$	14	7.95
2	2	$[5.1, 8.5]$	11	7.95
2	1	$[5.1, 8.3]$	11	7.95

Table 1.1 Main characteristics of the strategies synthesized with UPPAAL-TIGA.

Table 1.1 represents the results obtained for different granularities and margins. It gives the optimal stable interval I that is computed, (note that it is smaller if we allow a smaller margin or a finer granularity), the number of different local strategies, and the value of worst case mean volume which is obtained as $\mathsf{Score}(I)/20$. These strategies are evaluated in sections 1.4 and 1.5.

1.4 Checking Correctness and Robustness of Controllers

In this section, we address the verification of the correctness and robustness of the three solutions mentioned in the previous sections. To analyze the correctness and the robustness of the three controllers, we use the tool PHAVER [**?**, **?**] for analysing hybrid systems. Robustness is checked according to the type of controller we use: for the Bang-Bang controller, it amounts to saying that the volume cannot be measured accurately and also that the rate fluctuates ($\pm 0.1 l/s$); for the Smart controller, robustness against rate fluctuation cannot be checked because we do not have a precise mathematical model of the algorithm designed by the HYDAC ELECTRONIC GMBH engineers; for our synthesized controller, we take into account the rate fluctuation, the imprecision on the measure of the volume and the imprecision on the measure of time.

PHAVER allows us to consider a rich continuous time model of the system where we can take into account the fluctuations of consumption of the machine as well as adequate models of imprecision inherent to any real implementation. The PHAVER models of the controllers are given below and the other models of the cycle and ma-

chine are the timed automata of Fig. 1.7 and Fig. 1.6. Our models take into account the fluctuations in the consumption rate of the machine as well the imprecision on the measure of the volume. We now review the results for the three controllers.

1.4.1 The Bang-Bang controller

The PHAVER code of the Bang-Bang controller is given in Fig. 1.11. This automaton turns *on* the pump when a floor volume value is reached and turns *off* the pump when a ceiling value is reached.

```
1:  // ------------------------------------
    // bang bang Controller automaton
3:  // this controller starts in on or off and then
    // switch on or off when a bound is reached
5:  // ------------------------------------

7:  eps1:=0.06; // imprecision on the volume measure
    margin_min:=0.86; // best we can do
9:  margin_max:=0.06; // best we can do

11: automaton controller
    input_var: v; // v is given by the cycle+tank automaton
13:
    synclabs: switch_on , switch_off ; // synchronized with the cycle+tank
15:
    loc on: while v <= VMAX - margin_max + eps1 wait {true}
17:     when v>=VMAX-margin_max-eps1 sync switch_off do {true} goto off;

19: loc off: while v>=VMIN+margin_min - eps1 wait {true}
      when v<=VMIN+margin_min+eps1 sync switch_on do {true} goto on;
21:
     initially : off & true ; // values for no noise
23: end
```

Fig. 1.11 Bang-Bang controller in PHAVER

To ensure robustness (and implementability) of this control strategy, we introduce imprecision in the measure of the oil volume: when the volume is read it may differ by at most $\varepsilon = 0.06\ l$ from the actual value (precision of the sensor). Tuning this controller amounts to choosing the tightest values for the floor and ceiling values at which the controller switches the pump (from *on* to *off* or the other way). In our experiment we found that 5.76 and 25.04 are the best margins we can expect.
With this PHAVER model and the previous margins[8], we are able to show that: (1) this control strategy enforces the safety requirement R_1, i.e. the volume of oil stays within the bounds $[4.9; 25.1]$; (2) the set of reachable states for initial volume equal to 10 l can be computed and it is depicted in Fig. 1.12; this means that this controlled system is "cyclic" from the end of the first cycle on, and the same interval

[8] And another suitable piece of PHAVER program to perform the needed computations.

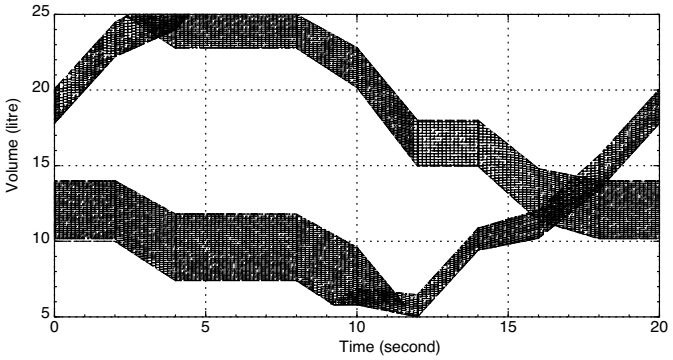


Fig. 1.12 Cyclic Behavior of the Bang-Bang controller with Noise

$[10.16;14]$ (for the volume) repeats every other cycle. It is thus possible to compute (with PHAVER) the interval of the accumulated volume over the two cycles: for this controller, the upper bound is 307 and the mean volume is $307/20 = 15.35$.

1.4.2 The Smart Controller

The Smart Controller designed by HYDAC ELECTRONIC GMBH is specified by a 400 line C program and computes the start/stop timestamps for the next cycle according to what was observed in the previous cycle (see end of section 1.2). This controller requires to sample the plant every 10ms in order to compute the strategy to apply in the next cycle: although it is theoretically possible to specify this controller in PHAVER, this would require at least 100×20 discrete locations to store the sampled data in the previous cycle. It is thus not realistic to do this as PHAVER would not be able to complete an analysis of this model in a reasonable amount of time.

Instead we have built the PHAVER controller (given in Fig. 1.13) that corresponds to the behaviour of the smart controller in a stationary regime, and in the absence of noise. It turns *on* and *off* so that the pump is active exactly during the three intervals $[2.16;4.16]$, $[9.05;11.42]$ and $[13.96;16.04]$ during each cycle. Indeed using simulation, the engineers of HYDAC ELECTRONIC GMBH had discovered that the behavior of their controllers in the absence of noise was cyclic (stable on several cycles) if they started with an amount of oil equal to 10.3 *l*. This is confirmed by the simulations we report on at the end in Fig. 1.19 and by Fig. 1.14, obtained with PHAVER showing that the smart controller stabilizes with no fluctuations in the rate.

However, our simplified version of the Smart controller given in Fig. 1.13 (without imprecision on the timestamps of start and stop of the pump), is not robust against the fluctuations of the rate: the behavior of the system in the presence of

```
1:  // ------------------------------------
    // HYDAC smart controller automaton
3:  // this controller starts in off and then
    // switch on or off at given time points
5:  // ------------------------------------

7:  // time between switch is at least 2

9:  automaton controller

11: contr_var: t; // this is the time reference of the controller
    synclabs: switch_on , switch_off, tau1 ; // synchronized with the cycle+tank
13:
    loc off1: while t<=2.16 wait {t'==1}
15:   when t==2.16 sync switch_on do {t'==t} goto on1;

17: loc on1: while t<=4.16 wait {t'==1}
      when t==4.16 sync switch_off do {t'==t} goto off2;
19:
    loc off2: while t<=9.05 wait {t'==1}
21:   when t==9.05 sync switch_on do {t'==t} goto on2;

23: loc on2: while t<=11.42 wait {t'==1}
      when t==11.42 sync switch_off do {t'==t} goto off3;
25:
    loc off3: while t<=13.96 wait {t'==1}
27:   when t==13.96 sync switch_on do {t'==t} goto on3;

29: loc on3: while t<=16.04 wait {t'==1}
      when t==16.04 sync switch_off do {t'==t} goto last;
31:
    loc last: while t<=20 wait {t'==1}
33:   when t==20 sync tau1 do {t'==0} goto off1;

35: initially : off1 & t==0 ;

37: end
```

Fig. 1.13 HYDAC ELECTRONIC GMBH Smart Controller in PHAVER

noise is depicted in Fig. 1.15 and it can be shown with our PHAVER models that after four cycles, the safety requirement R_1 can be violated. Unfortunately, there is no way of proving the correctness of the *full* Smart controller with PHAVER, and SIMULINK only gives an average case. In this sense we cannot trust the Smart controller for ensuring the safety property.

The ideal Smart Controller (no noise on the rate) produces an average accumulated volume of around 221 per cycle i.e. an average volume of 11.05.

1.4.3 Controller Computed with UPPAAL-TIGA

We now study the correctness and robustness of the controller synthesized with UPPAAL-TIGA. This verification phase is necessary because during the synthesis phase we have used a very abstract model of the system and also discrete time. To force robustness and correctness, we have imposed additional requirements on

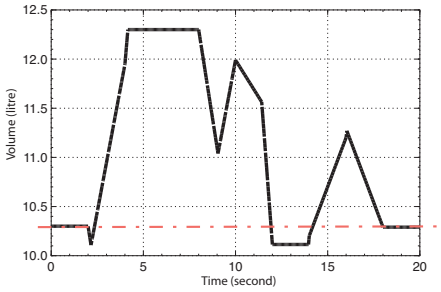


Fig. 1.14 Smart Controller / no fluctuations

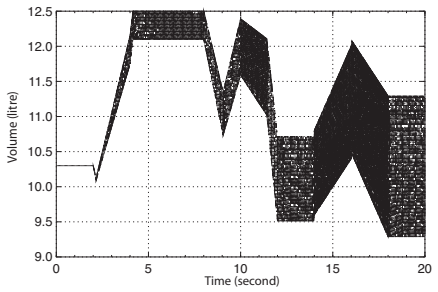


Fig. 1.15 Smart Controller / fluctuations

the winning strategies (our inductive property together with the margin). Instead of proving by hand that the model and the objective yield to a correct and robust controller we perform a formal post-check of the controller in the presence of noise and imprecision. We summarize here the results of this verification phase. In the sequel we use the controller for granularity 2 and margin 4: this controller can be seen as 14 different local controllers, each one managing one of the 14 intervals in which the initial volume can be at the beginning of a cycle. We will focus on those strategies here but we have automated the process and the others may be treated along the same lines.

To make sure that our strategies are implementable, we have verified them in presence of fluctuations of the rate consumption and two types of imprecision: on the time-stamp of start/stop of the pump (we use $\Delta = 0.01$ second), and on the measure of the initial volume, the imprecision being $0.06\,l$. Fig. 1.16 shows how the volume is controlled over 3 cycles: after the first one at $t = 20$, we measure the real volume with uncertainty ($0.06\,l$) and use the corresponding controller from 20 to 40 and for 40 we again switch to another one.

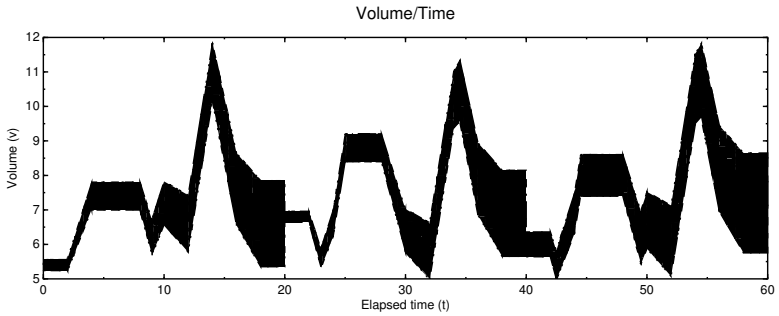


Fig. 1.16 The Pump Controlled over 3 Cycles

We have designed a generic PHAVER model for controllers with 2 starts and 2 stops during one cycle which is given in Fig. 1.17.

```
1:  // ------------------------------------
    // TIGA controller automaton
3:  // this controller starts in off and then
    // switch on or off at time points defined in another file
5:  // ------------------------------------

7:  // time between switch is at least 2
    delta:=0.01; // this defines the maximum error on
9:               // the date at which start/stop are performed

11: automaton controller

13: contr_var: t; // this is the time reference of the controller
    synclabs: switch_on , switch_off ; // synchronized with the cycle+tank
15:
    loc off1: while t<=start1+delta wait {t'==1}
17:   when t>=start1-delta sync switch_on do {t'==t} goto on1;

19: loc on1: while t<=stop1+delta wait {t'==1}
      when t>=stop1-delta sync switch_off do {t'==t} goto off2;
21:
    loc off2: while t<=start2+delta wait {t'==1}
23:   when t>=start2-delta sync switch_on do {t'==t} goto on2;

25: loc on2: while t<=stop2+delta wait {t'==1}
      when t>=stop2-delta sync switch_off do {t'==t} goto last;
27:
    loc last: while true wait {true} ;
29:
     initially : off1 & t==0 ;
31:
    end
```

Fig. 1.17 Generic PHAVER Controller with two Start/Stop(s).

For example, the controller for initial volume within $[5.7;6.3]$ is obtained by setting `start`i and `stop`i with the correct values for this initial volume. In this automaton, there is a variable δ (`delta`) which models the interval in which we issue the start/stop commands: we cannot measure time with infinite accurracy and thus we will only be able to issue the start/stop actions in an interval around the precise time points given by the controller: if the ideal synthesized controller has to issue `switch_on` at 2.5, the implementation of the controller can only ensure it will be issued in $[2.5-\delta;2.5+\delta]$. We use the model for the cycle and pump automaton given Fig. 1.4 and 1.5. The values v_1 and v_2 are set according to the controller we want to check (e.g. $v_1 = 5.7$ and $v_2 = 6.3$ for the controller which has to be used for the volume within $[5.7;6.3]$). As our controllers should handle all the possible values of the volume at the beginning of a cycle, as well as to be robust w.r.t. errors in the volume measurement, we add a variable (ε) `eps` which models this error: it means we use the controller for $[5.7;6.3]$ on a larger interval which is given by $[5.7-\varepsilon;6.3+\varepsilon]$. Still our controller should ensure that the final volume is within $[5.1;8.9]$. To ensure overlapping and full coverage of the initial volume range, we

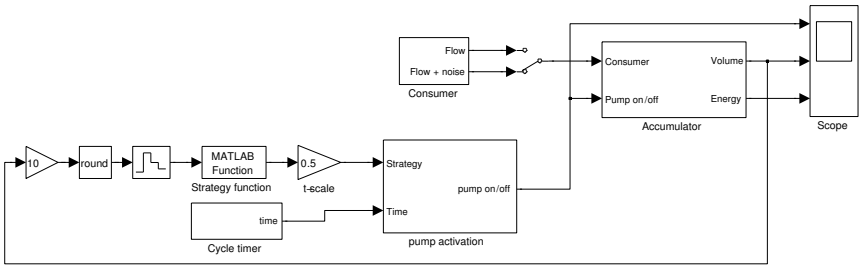


Fig. 1.18 The overall SIMULINK model.

need to set ε larger than 0.05: we choose 0.06 for the following experiments[9]. To validate the controller synthesized with UPPAAL-TIGA we check the following:

1. we set $\delta = 0.01$ second, $\varepsilon = 0.06$, and the maximum rate fluctuation is $f = 0.1$;
2. we check that the set of reachable states of each of the 14 controllers is within $[V_{min}; V_{max}]$ which is the safety requirement of the accumulator;
3. we check that, starting from $I_\varepsilon = [5.1 - \varepsilon; 8.9 + \varepsilon]$ the final values of the volume are within the interval $[5.1; 8.9]$. Thus we have an inductive proof that our controller is safe and robust w.r.t. triple (δ, ε, f).

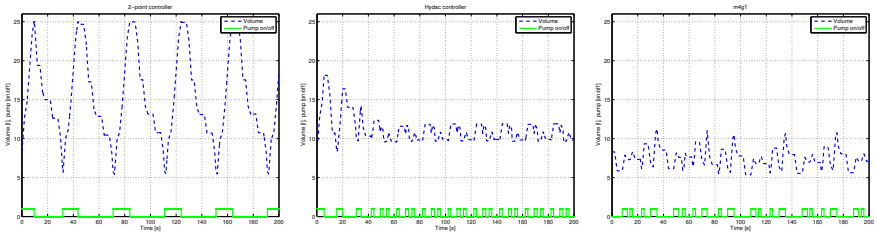
1.5 Simulation and Performances of the Controllers

In this section, we report on results obtained by simulating the three controller types in SIMULINK, with the purpose of evaluating their performance in terms of the accumulated volume of oil.

SIMULINK models of the *Bang-Bang* controller as well as of the *Smart* controller of HYDAC ELECTRONIC GMBH have been generously provided by the company. As for the eight controllers – differing in granularity and margin – synthesized by UPPAAL-TIGA, we have made a RUBY script which takes UPPAAL-TIGA strategies as input and transforms them into SIMULINK's *m*-format.

Fig. 1.18 shows the SIMULINK block diagram for simulation of the strategies synthesized by UPPAAL-TIGA. The diagram consist of built-in functions and four subsystems: Consumer, Accumulator, Cycle and Pump (we omit the details of the subsystems). The Consumer subsystem defines the flow rates used by the machine with the addition of noise: here the choice of a uniform distribution on the interval $[-\varepsilon, +\varepsilon]$ with $\varepsilon = 0.1l/s$ has been made. The Accumulator subsystem implements the continuous dynamics of the accumulator with a specified initial volume ($8.3l$ for the simulations). In order to use the synthesized strategies the volume is scaled by a factor 10, then rounded and fed into a zero-order hold function with a

[9] If the real volume is 5.65, we may obtain a measure of 5.7 or 5.6: what we check is that both the controllers for 5.7 and 5.6 will ensure the final volume is the interval $[5.1; 8.9]$.



(a) Bang-Bang controller (b) Smart Controller (c) Synthesized Controller

Fig. 1.19 The three controller types with SIMULINK

sample time of 20s. This ensures that the volume is kept constant during each cycle, which is feed into the strategy function. The Pump activation subsystem takes as input the on/off timestamps from the strategy (for the given input volume of the current cycle) and a Cycle timer, that holds the current time for each cycle.

Now, the plots in Fig. 1.19 are the result of SIMULINK simulations of the controllers, illustrating the volume of the accumulator as well as the state of the pump (on or off) for a duration of 200 s, i.e. 10 cycles. Though the simulations do not reveal the known violation of the safety requirement R_1 in the HYDAC Smart controller case, the simulations yield useful information concerning the performance of the controllers. In particular, the simulations indicate that the accumulated oil volume for all controllers grow linearly with time. Also, there is clear evidence that the strategies synthesized by UPPAAL-TIGA outperform the Smart controller of HYDAC – which is not robust – and also the Bang-Bang controller – which is robust but far from optimal.

Controller	Acc. volume	Mean volume	Mean volume (TIGA)
Bang-Bang	2689	13.45	-
HYDAC	2232	11.16	-
G1M4	1511	7.56	8.45
G1M3	1511	7.56	8.35
G1M2	1518	7.59	8.25
G1M1	1518	7.59	8,2
G2M4	1527	7.64	8.05
G2M3	1513	7.57	7.95
G2M2	1500	7.5	7.95
G2M1	1489	7.44	7.95

Table 1.2 Performance characteristics based on SIMULINK simulations.

This is highlighted in Table 1.2, giving – for each of the ten strategies – the simulation results for the accumulated volume of oil , the corresponding mean volume as well as the worst case mean volume according to synthesis of UPPAAL-TIGA. The

table shows – as could be expected – that UPPAAL-TIGA's worst case mean volumes consistently are slightly more pessimistic than their simulation counter-parts. More interestingly, the simulation reveals that the performances of the synthesized controllers (e.g. G2M1) provide a vast improvement both of the Smart Controller of HYDAC ELECTRONIC GMBH (33%) and of the Bang-Bang Controller (45%).

1.6 Conclusion

In this paper we have presented a model-based methodology for the systematic development of robust and near-optimal controllers. The methodology applies a chain of tools for automatic synthesis (UPPAAL-TIGA), verification (PHAVER) and simulation (SIMULINK). Initially, sufficiently simple and abstract game models are used for synthesis. The correctness and robustness of the strategies are then verified using continuous hybrid models and – finally – the performance of the strategies are evaluated using simulation models.

Applied to the industrial case study provided by HYDAC ELECTRONIC GMBH, our method provides control strategies which outperforms the *Smart* controller as well as the simple *Bang-Bang* controller considered by the company. More important – whereas correctness and robustness of the Smart controller is unsettled – the strategies synthesized by our method are provably correct and robust. We believe that the case study demonstrates the maturity and industrial relevance of our tools.

Directions for further work include:

- Improve the performance of our controller further by optimizing over several cycles, and/or
- Improve the performance of our controller further by adding some predefined points when we can measure the volume (even with imprecision).
- Consideration of other imprecision, e.g. with respect to the timing of consumer demands.
- Consideration of other optimization criteria. An interesting feature of the *Smart* controller of HYDAC ELECTRONIC GMBH seems to be that the oil volume is kept in a rather narrow interval, a feature which could possibly be beneficial for increasing the life-time of the Accumulator.
- Use the emerging version of UPPAAL-TIGA supporting synthesis under partial observability in order to allow more accurate initial game models.

Energy Consumption in the Chess WSN: A MoDeST Case Study

Henrik Bohnenkamp and Joost-Pieter Katoen and Haidi Yue

Abstract In this chapter we demonstrate the use of MoDeST, as introduced in the previous chapter, by means of a case study carried out in the Quasimodo project. We analyse the energy efficiency of a Wireless Sensor Network MAC protocol, gMAC for short, which was patented by the Dutch company CHESS BV.

1 Introduction

Quasimodo-Partner CHESS has developed a battery-powered wireless sensor node, which communicates with neighbouring nodes via broadcast communication and relays information through the network via gossiping. The limited power source of the individual nodes requires most efficient communication between nodes in order to extend the network lifetime. CHESS has developed gMAC, a TDMA (*time-division multiple access*) protocol with mechanisms to detect collisions between sending nodes. The protocol attempts to establish a sending schedule which minimises the collisions between communicating nodes.

In this chapter, we present a case-study where the energy efficiency of gMAC is analysed. The protocol is modelled in MoDeST [1], and analysed by discrete-event simulation with the Möbius [5] tool set. In particular, we concentrate on the energy efficiency of the protocol with respect to message dissemination. Whereas TDMA protocols have by design a very predictable energy demand over time, we analyse the energy needed to distribute a message through the whole network, which depends on the quality of the send schedule.

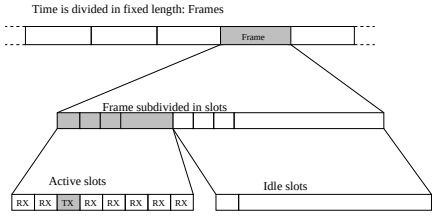
The analysis of this case-study is carried out with the discrete event simulator of the Möbius performance evaluation environment [5], which provides the means to obtain statistical data ranging from mean values to distributions (see also previous

Henrik Bohnenkamp and Joost-Pieter Katoen and Haidi Yue
Modeling and Verification of Systems (MOVES), RWTH Aachen University, D-52056 Aachen, Germany. e-mail: {henrik|katoen|haidi.yue}@cs.rwth-aachen.de

chapter). The connection between MoDeST and Möbius is facilitated in a fully-transparent way by the tool MoToR [3].

Overview. In Section 2 we describe the basic mechanisms of gMAC. In Section 3, we describe how we modelled gMAC in MoDeST. In Section 4 we describe the Experimental setup of our simulations. In Section 5 we discuss the results. In Section 7 we briefly review other case studies carried out with MoDeST/Möbius. We conclude with Section 7. This chapter is based on [11].

2 The gMAC Protocol



(a)

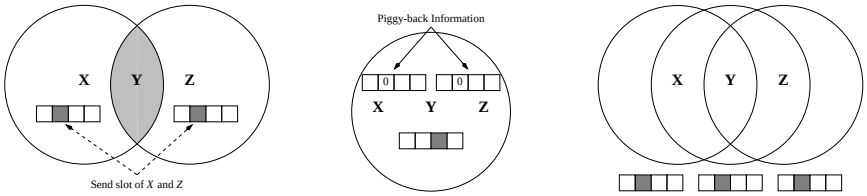
mode	transmit	receive	idle
current	11.3 mA	12.3 mA	0.9 μA

(b)

Fig. 1 TDMA frame structure and energy demands

The gMAC protocol is a TDMA protocol, *i.e.*, time is divided in fixed-length *frames* (*cf.* Fig. 1(a)). A frame is divided in an active and idle period, and both periods are subdivided into *slots* of equal length. A node in the network is synchronised with its immediate neighbours at the beginning of a frame. One active slot is chosen as *send slot* (the *TX slot*). All other active slots are *receive slots* (*RX slots*). During the idle period, the radio is put in idle mode to save energy. In an RX slot, a node listens for incoming messages from neighbouring nodes, in its TX slot it sends a message. When the active period is over, it switches to idle mode again, and so forth. With S the number of slots within a frame and $A \leq S$ the number of active slots, A is a crucial parameter in the protocol design, as the active operation phase costs much more energy than the idle phase. The CHESS network nodes are equipped with an ATMega64 micro-controller and a Nordic nRF24L01 [10] packet radio. The energy demands of the nRF24L01 radio are summarised in the table of Fig. 1(b). A is usually much smaller than S. In the gMAC protocol implementation with the aforementioned processor, usually S=1129, and $A = 10$, give or take.

When a node is powered on, it randomly chooses an active slot as TX slot. In each RX slot, it can receive a message of at most one other node. The *hidden node problem* describes the scenario when more than one node sends messages to the same node in the same slot.



(a) Hidden node situation (b) Piggybacking (c) Problematic scenario for piggy-back technique

Fig. 2 Hidden node problem and its detection

Figure 2(a) depicts a situation where nodes X, Y, Z are positioned such that the middle node Y is within the transmission range (the circles) of both other nodes, and both X and Z are outside each others range. If X and Z select the same TX slot, then their messages will collide in the intersection of their ranges. They cannot sense this themselves, and Y will receive no message at all as it cannot distinguish a collision from the situation where no message was sent.

The mechanism of the gMAC protocol to detect collisions and hidden nodes is based on *piggy-back information* of the payload messages. With each message, the sender's perspective on the current slot allocation is also transmitted. This can be represented as a sequence $(b_0, b_1, ..., b_{A-1})$ with $b_i \in \{0,1\}$ for $0 \leq i < A$. $b_i = 0$ indicates that nothing has been received in slot i, either because nobody sent something, or due to a collision or message loss. $b_i = 1$ indicates that the sender received something in slot i, or that slot i is the sender's own current send slot. In the example in Figure 2(a), since Y cannot receive anything in the second slot, it writes a 0 in its piggy-back information at the corresponding position and reports this to X and Z on its turn to send, in the third slot (*cf.* Figure 2(b)). Based on this information from Y, nodes X and Z can conclude that there was a collision in their send slot. The gMAC protocol then stipulates that X and Z pick randomly a new send slot among the free active slots, to avoid further collisions. Note that it is possible that no free slot is available when a node needs one. This can happen when the nodes are in a very crowded environment and the number of neighbours exceeds the number of active slots (some of our simulation configurations cover this situation). In this case, the node will keep the old send slot despite the detected collision in that slot. Although the piggy-back technique helps to detect many collisions, there are still some it cannot find. In Figure 2(c), node Y has the same send slot as X and Z, *i.e.*, they send and receive at the same time and will therefore never receive anything from each other in this slot, hence the collisions between them will neither be detected nor resolved. The reason for this is that the piggy-back technique requires at least one common neighbour which is not involved in the conflict, so that it can report the collision. The gMAC protocol provides one more mechanism to break this kind of conflict. When a node reaches its send slot, it can decide with a certain probability p to *not* send, but to listen. This gives a node a chance to overhear what is going on in its own

send slot, and an opportunity to pick a new send slot, if necessary. We call a node where this mechanism is enabled *probably silent*. A typical value for p is $1/16$.

3 Modelling gMAC in MoDeST

In this section we describe the key components of the MoDeST model of the gMAC case study. The most important language concepts are introduced in Chapter 10,and should be read first. For a comprehensive overview of MoDeST we refer to [1].

MoDeST: Special Features. In the MoDeST/Möbius tool chain, several design decisions had to be made to implement the MoDeST language. The reason is that the MoDeST reference [1] gives implementation freedom in some aspects of the language. In particular, the atomic assignment blocks, {=...=}, are only semantically defined: they are supposed to compute a function or random variable, depending only on the current variable valuation. With which language to describe this function is left open. During the integration of MoDeST with the Möbius simulator, the choice was made to implement a non-trivial fragment of the C language. This fragment allows the definition of high-level constructs such as arrays and structs, block-local variables, and has control structures such as if-then-else, while- and for-loops, and rudimentary output functions, similar to C's `printf()`. It is also possible to call external C-library functions, such as function from the libm library. While this does not increase the expressivity of MoDeST, it proves beneficial in terms of efficiency. For example, without while-loops in assignment blocks, such loops must be emulated with the heavy-weight when- and do-loop constructs of MoDeST. This usually creates a huge number of intermediate states, which would hamper the analysis of the model.

The MoDeST model of the gMAC protocol makes extensive use of the introduced C-features.

Basic model. A node is modelled as a MoDeST process, `Node`, with parameter `id`, which is its unique identifier. A local clock `c` is used to measure the length of slots. It is reset to 0 at the start of each active slot. An integer `my_send_slot` denotes the current send slot of the node. Furthermore, a node maintains its perspective about the current slot allocation as an array `view` of booleans of length A. The neighbourhood relation of a node is stored in an array, which is initialised at the beginning of a simulation run. In our simulations, we have used a 15×15 grid structure for the network.

Communication between nodes is modelled by means of a global array of *buffers*, which is accessible as shared memory by all nodes. Every node accesses all buffers of the nodes in its transmission range while writing, but only its own buffer `bufs[id]`, when receiving. A buffer contains several variables (*cf.* Figure 3), among them `is_written_to`, `writers` and `writing_completed`. The three variables are used as counting semaphores to coordinate sending and receiving nodes. Furthermore, they are used to detect failed transmissions. If sending

is successful, the view of the sender, which contains information for collision detection, will be copied to the receiver's buffer `received[]`.

```
typedef struct {
 int msg;                   //  message
 int is_written_to;         // a node sends to this buffer;
                            // >1 => collision has occurred
 int writer;                // # nodes working on the buffer
 int writing_completed;     // to indic. that
                            // node has finished sending.
 int is_sending;            // direct neighbor is sending.
 int received[activeslots]; // piggybacking info of sender

} buffer;

buffer bufs[NETWORKSIZE];    // message buffer for each node
```

Fig. 3 Buffer structure

The behaviour of the nodes is modelled using extensive data-manipulation. We use atomic assignment blocks in combination with a preceding `when(...)` conditional to implement a test-and-set operation, which ensures mutual exclusion between processes. A simple example for a P- and V-operation on a semaphore `s` is

```
when(s == 0) {= s = 1=}; /* crit. section */ {= s = 0 =}
```

In Figure 4, we see a sketch of the send operation as modelled in MoDeST. When a node decides to send, it first sets its `is_sending` variable in `bufs[id]` to 1 to signal that it is sending. This variable is used to detect collisions between direct neighbours. This kind of collision can not be detected by the piggy-back technique, if there are no common neighbours. Before a node actually sends in its slot, it must wait until the guard time has passed and the actual send period begins. In the actual send period, three steps are executed, which are marked with the three action names `start_sending`, `msg_sent`, and `reset_channel`[1]. Each of these actions is accompanied by a {=...=} block in an atomic manner, where the actual operations are carried out. For `start_sending` this means to increase counters `is_written_to` and `writers` in all buffers of the neighbour nodes. Then, when the send period is over, action `msg_sent` is executed. The node indicates that sending is completed and decreases the counter `writers` in all neighbouring buffers. Then it checks if the sending was successful, by examining if the variable `is_written_to` in the neighbours buffer is equal to 1. If this is the case, the sender's view will be copied to the neighbours buffer. Otherwise, if `is_written_to>1` or `is_sending==1`, the node knows that a communication has failed with the neighbour. In this case, the global variable `real_collisions`,

[1] These action names have only a descriptive purpose here. Although MoDeST allows synchronisation over actions between different processes, this feature is not used here. This was a design decision in order to be able to extend the model with clock drifts and jitter later.

used to keep statistics of failed communications, is increased by 1. Finally, at the end of the send slot, with action `reset_channel`, the buffers of the neighbours are cleaned and the next slot is prepared.

```
// is_sending=1
// macro SEND()
// we assume c == 0 here
when (c == guard_time)  // guard_time is over
  start_sending {=
   // add energy spent on sending
    energy[id] += 0.001*PL_TRANSMITTING,
    for all neighbors:
      bufs[neighbor].is_written_to += 1
      bufs[neighbor].writers += 1
  =};
when (c == slot_length - guard_time) // end of send period
  msg_sent {=
    for all  neighbors:
      bufs[neighbor].writers -= 1,
      // > 1 means collision
      if(bufs[neighbor].is_written_to == 1){
        // copy view array to bufs
      }
      if(   bufs[neighbors].is_written_to >1
         || bufs[neighbors].is_sending=1){
        real_collisions+=1
      }
  =};
when (c == slot_length) // end of slot
  reset_channel  {=
    // increase slot number
    c = 0,
    for all  neighbors:
      // set everything in bufs[neighbor] to 0
  =}
```

Fig. 4 Mechanism for sending

In Figure 5, we see the tasks that are executed when a message has been successfully sent to a node and can thus be *received*. This fragment is executed at the end of the send period of the sending nodes, and only if the counter `is_written_to` in the node's buffer is equal to 1. In that case, the node will set its view of the current slot to 1 and check for collisions in its own send slot through the piggyback information from the sender. If a collision is detected, a new free send slot is chosen probabilistically. Before a node can receive a message as described, it has to *listen* first. This is modelled as sketched in Figure 6. With the `alt` construct, a choice between the case that a message is in-bound and can be received is modelled (`RECEIVE()` stands as a placeholder for the model fragment in Figure 5), or that nothing has come in. The condition for the latter case is that local clock *c* pro-

```
// macro RECEIVE()
when(bufs[id].is_written_to == 1 // exactly one process wrote
     && bufs[id].writers == 0 // the one sender is done
                              // with writing to buffer
     && c >= slot_length - guard_time ) // end of send period
  msg_received    {=
    view[slot_nr] = 1,         // we received something in slot
      if (pigyback-information indicates collision )
         // choose new send slot randomly
  =}
```

Fig. 5 Mechanism for receiving

gresses without disturbance until the end of the slot. We thus have a choice between the timed guard `c == slot_length` and the untimed guard of the `RECEIVE()` macro. If `c == slot_length` becomes true, it means, nothing has been received in that slot, hence the view of this slot is set to 0 and the next slot is prepared. The

```
// macro LISTEN()
when ( current_slot != my_send_slot ) // we are listening
listen {= energy[id] += 0.001*PL_RECEIVING  =};
  alt {
    :: when(c == slot_length)
       {=  view[slot_nr] = 0   // nothing received  =}
    :: RECEIVE() /* as explained above */
  };
  when (c == slot_length)
  {= ... increase slot number ...,  c = 0 =}
```

Fig. 6 Listening for incoming message

described behaviour is put together in process `Node`, and all the Nodes are put in a parallel composition, as sketched in Figure 7.

The use of energy is modelled by updating a global array `energy[]` with the energy consumed by the node. We assume the energy drainage to be constant in the different modes of the node (sending, receiving, idle), which allows us to update the variable with simple linear expressions (*cf.* Figures 4 and 6).

Probably silent nodes. One mechanism of the gMAC protocol to increase collision awareness is for a node to choose probabilistically in its send slot whether to send, as described before, or to listen. This extension can be modelled very quickly in MoDeST with the probabilistic choice operator, *palt*. The line `when (my_send_slot == slot_nr) SEND()` in Figure 7 is replaced by the fragment in Figure 8. The probability chosen in this fragment is $p = 1/16$.

```
process Node(int id) {
 // initialisations
 do {  // repeat for each slot
  :: when (slot_nr > activeslots) ...  // idle until frame ends
  :: when (my_send_slot==slot_nr) SEND()... // my turn to send
  :: LISTEN()...                          // otherwise, listen
 }
}

par{
 :: hide all actions in (Node(1))
 :: hide all actions in (Node(2))
  ...
}
```

Fig. 7 Structure of process `Node` and parallel composition

```
when (my_send_slot == slot_nr)
 palt {
  :1: alt {
      :: when(c == slot_length)  // nothing received
         {= view[slot_nr] = 0 =}
      :: RECEIVE();
      }
  :15: SEND(id, c, rate)
   }
```

Fig. 8 Sending or listening

4 Experimental Setup

For our experiment, we use the unit disk graph radio model to model the neighbourhood relation between nodes. Nodes with a distance larger than some radius r can thus never receive nor interfere with each other. Within this radius, the situations as depicted in Fig. 2 can occur. All network nodes are assumed to have the same transmission range. Message losses are assumed to be due to collisions only. gMAC incorporates a mechanism to synchronise clocks of neighbouring nodes. The correctness of this clock-synchronisation mechanism is being analysed in Chapter 4.In a realistic network, clock synchronisation can be assumed to have taken place before the network starts doing something useful, which allows us to abstract from the clock-synchronisation mechanism. The only aspect of it that we must consider is the fact that a node does not send during the whole length of a send slot, but only a fraction of it. This does influence the energy consumption, and therefore we incorporate these shorter sending times in our model.

Set-up. The basic model of our experiments is a 15×15 grid network of 225 nodes. Each node has a distance of 1 to its respective horizontal and vertical neighbours (*i.e.*, the distance to the diagonal neighbours is $\sqrt{2}$). A frame consists of $S = 1129$

slots. The number A of active slots is a crucial parameter in the protocol, and we analyse the behaviour of the gMAC protocol for different values of A. Since in the real implementation A=8, we choose the transmission range r such that each inner node of the grid has 4 or 8 direct neighbours, respectively. We say a node is *randomly silent*, if it stays silent in its TX slot with probability $p = \frac{1}{16}$. We adopt this value in our model and use it for all experiments. The experiments focus on the latency of message dissemination versus the required energy consumption. The confidence level of all simulations is set to 0.95 and the relative confidence interval is 0.1.

Latency vs. energy consumption. A TDMA protocol is by design very predictable with respect to energy consumption: since only in the active slots the energy drain of the radio is non-negligible, and the number of active slots per frame is constant, also the power consumption is close to constant, depending linearly on the number of active slots. Energy consumption is therefore constant over time. However, the question is how energy-efficiently the network fulfils its task. We thus focus on the latency of message dissemination and the energy consumed by that. We consider the average time required and the total average energy consumed until a message is delivered to all network nodes. We say a node is *infected* if it has received a message. Initially, only one node is infected, the initiator. To get insight into the effect of the position in the network of the message initiator, we consider a corner node, a middle node at the border, and a center node (cf. Figure 9). Again, the simulations are run for different values of r and A to investigate the influence of these parameters on gMAC's energy consumption.

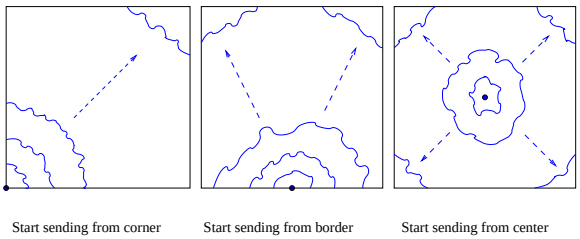


Fig. 9 Three different initiator positions

We run all aforementioned experiments for three network settings:

1. A static network without *randomly silent* nodes (for short *grid*),
2. A static network with *randomly silent* nodes (for short *grid+p*),
3. A network with node mobility but no *randomly silent* nodes (for short *grid+m*), so that we can obtain a clear comparison between static and mobile scenarios without influence of *randomly silent* nodes.

Since we want to investigate the influence of local changing of node position on the network, we model the mobility by rotating a fixed row (the fifth row) in the grid one position to the right. The node shifted out is shifted in on the other side.

The row is rotated one position every 100 frames for the collision experiment, so that the network has enough time to stabilise after each shift. Since the average time required to deliver a message to all nodes is less than 30 frames, we rotate every 1 or 3 frames to investigate the influence of the moving rate on the latency.

5 Energy Efficiency of gMAC

We consider a static network first. Figure 10(a) shows the experimental results for transmission range r=1.1 (4 neighbours), and A ranging from 4 to 7. The message initiator is positioned in the corner. The *circle-lines* show the energy consumption (right y-axis) versus the number of frames, and the black, curved lines (left y-axis) show the ratio of infected nodes (i.e., nodes that have received a message) versus the number of frames.

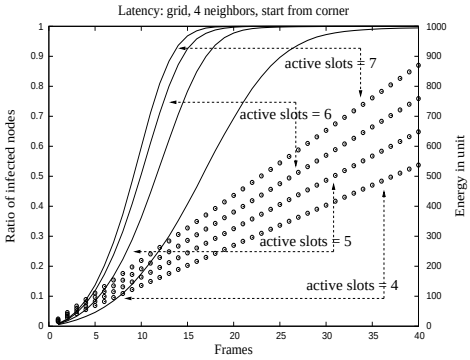
The results confirm that for fixed A, there is a linear dependency between the energy consumption and the number of frames, which is characteristic for TDMA protocols. The slope depends on A; the larger A, the steeper the energy curves. For the message dissemination, it can be observed that after a short warm-up phase, the fraction of infected nodes drastically grows, after which this slowly progresses to 1. For increasing A, the percentage of infected nodes converges to more quickly to one, i.e., message dissemination is faster.

The larger A, the lower the message latency becomes, but as a pay-off, the energy consumptions increases with larger A. In order to get insight into the trade-off between message dissemination and energy consumption, Figure 10(b) depicts an energy-percentage diagram, which shows the percentage of infected nodes versus the total energy needed to infect all nodes. One clearly sees that A=4 and A=7 are not economical and in the considered scenario, a network with 5 or 6 active slots provides the best result in terms of energy efficiency. Performing the experiments for $A = 8,9$ and 10 reveals that these settings are less energy-efficient than for $A = 7$. For $A = 10$, e.g., the network tends to be collision-free, but requires twice as much energy as for $A = 5$ without offering a doubled propagation speed. We performed the experiment for three different initial sending positions and different transmission ranges. All of them exhibit a pattern similar to Figure 10(b). The optimal values of A are summarised in Table 1.

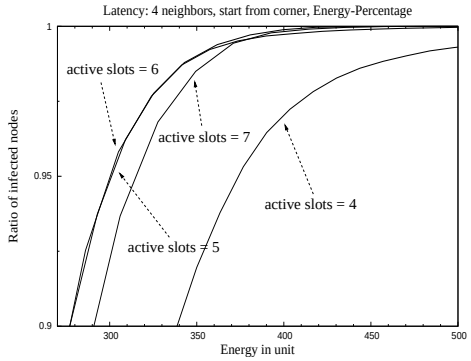
Range \ Position	corner	border	center
4 neighbours	6	5	5
8 neighbours	8	9	8

Table 1 Optimal A values

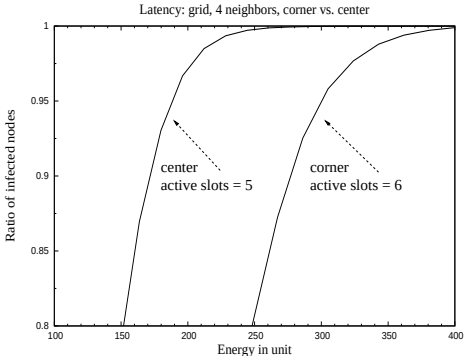
Figure 10(c) shows another effect of changing the initial sending position. We put the most energy-efficient results from a network with 4 neighbours and initial



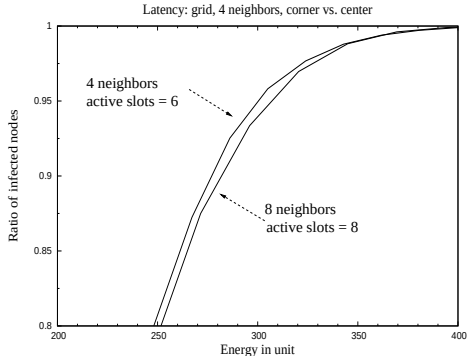
(a) grid, 4 neighbours, start sending from corner



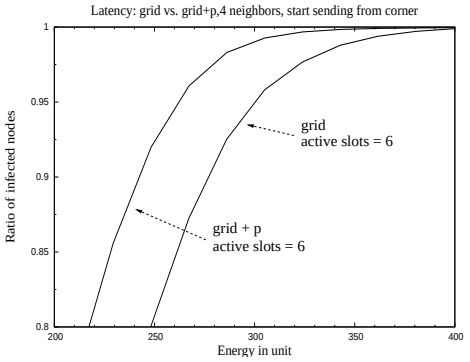
(b) Energy-percentage: grid, 4 neighbours, start sending from corner



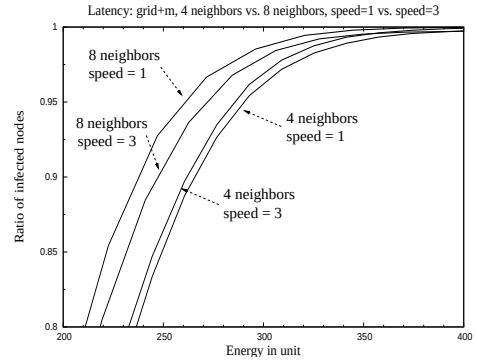
(c) corner vs. center, grid, both 4 neighbours



(d) 4 neighbours vs. 8 neighbours: grid, both start sending from corner



(e) *grid* vs. *grid+p*, both have 4 neighbours and start sending from corner



(f) different transmission ranges and shifting rates

Fig. 10 Latency vs. energy consumption

sending position at the corner or the center in one graph. Obviously, starting from the center needs only two third energy of that starting from the corner. It does not come as a surprise that message dissemination from the center is more efficient than from a corner, since the number of hops to the nodes farthest away is smaller. However, when we consider the influence of network density on latency, we can see that with a fixed initiator, a network with 4 neighbours or 8 neighbours exhibits almost the same performance (*cf.* Figure 10(d)). This means, although a denser network can propagate messages faster (a result which we have not shown here), it takes still as much energy as in a less dense network to deliver a message to the whole network.

Random silence. The results for *grid+p* show a similar behaviour, hence we will not present them here. Interesting is however the comparison between *grid* and *grid+p*. In Figure 10(e), we see the most economical results of *grid* and *grid+p*, both with 4 neighbours and the same initial sending position. The superiority of *grid+p* is quite clear, since roughly 15% energy can be saved if nodes are *randomly silent*. This is not self-evident, since for the used radio, receiving costs actually more energy than sending. We believe that the 15% drop in energy consumption is because *grid+p* has in general more opportunities to receive messages, which accelerates information dissemination.

Node mobility. For a network with mobility, we consider first the case of start sending from the corner. There are two options for transmission range (4 neighbours or 8 neighbours) and for the speed of shifting: every frame, or every third frame. As before, we combine the best results from each of these combinations in one graph (Figure 10(f)) to compare them. Recall that in the simple *grid* network, the density does not have a significant influence on the latency (see Figure 10(d)). However, in *grid+m*, if the other parameters are identical, the difference between a network with 4 neighbours and 8 neighbours cannot be neglected (compare the left-most curve with the right-most one, or the two middle curves). The influence of the speed of shifting is not very significant (compare the left-most two curves or the right-most two curves), and it is difficult to judge which speed overcome the others, for instance, speed=3 performs better than speed=1 for neighbours=4 while the trend is reversed for neighbours=8. This is due to the way we modelled mobility. In our mobility scenario, it takes about 15 frames to deliver messages to the whole network, and a shifting of every 1 frame or every 3 frames cannot have much influence on the result. Under other mobility models, different results will be obtained.

6 Further Work

In [7], the experiments described in this chapter have been repeated. However, main objective there is the comparison of two radio models: the unit disk model, and the physical Signal-to-Interference-plus-Noise-Ratio (SINR) model of [6]. In the SINR model, a node i can receive a message from another node j, if the signal strength

of j at the location of i exceeds the background noise and the sum of all signals of other sending nodes by at least a factor $\beta > 1$.

Two different protocols are compared, each in combination with the two different radio models. The first protocol is gMAC, as described above. The second one is a variant of *Slotted ALOHA*, where no collision detection is performed and the send slot is chosen randomly at the beginning of each frame. Both protocols have been modelled in MoDeST.

The results of the analysis show that, when using the unit disk model, the gMAC protocol appears superior to slotted ALOHA, with a reasonably chosen parameters. With the SINR radio model, however, it turns out that gMAC gives only slightly better results than slotted ALOHA, and might even perform worse in certain cases. This latter result has been independently obtained by CHESS (by measuring their existing protocols) and led to a redesign of gMAC based on slotted ALOHA. This new protocol, *distributed slotted ALOHA (dsA)*, keeps track of the number of neighbors that have been received in successive frames, and varies the number of active slots (more neighbors, more slots).

The SINR radio model suggest a modification of this protocol, in particular the fact that the sending signal of one node is noise to another sending node. This prompts the question whether in a crowded cluster of nodes the sending nodes should not reduce their signal strength in order to disturb fewer nodes farther away. Whereas this hardly decreases the energy consumption of individual nodes, it might increase the overall network throughput.

At the time of writing of this chapter, the distributed slotted ALOHA protocol with and withoute adaptive signal strength is being evaluated with MoDeST/Möbius.

7 Conclusions

In this chapter we have demonstrated how MoDeST can be used to model network protocols for simulation purposes. This is only one application The gMAC protocol, aimed for gossiping-based applications in sensor networks. Our analysis reveals that randomly deciding to refrain from using send slots significantly reduces energy consumption by about 15%. Node mobility does not affect the number of detected collisions. We determined the number of active slots that optimise the trade-off between latency and energy consumption. In the setting with 8 neighbour nodes, our experimental results confirm the optimality of CHESS's current node implementation (i.e., A=8).

The analysis of the MoDeST model has been facilitated with the Möbius analysis environment, by using the tool MoToR as intermediary [2, 3, 4, 8].

References

1. H. Bohnenkamp, P.R. D'Argenio, H. Hermanns, and J.-P. Katoen. Modest: A compositional modeling formalism for real-time and stochastic systems. *IEEE Trans. on Software Engineering*, 32(10):812–830, 2006.
2. H. Bohnenkamp, H. Hermanns, and J.-P. Katoen. Motor: The MoDeST tool environment. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 500–504. Springer-Verlag, 2007.
3. Henrik Bohnenkamp, Tod Courtney, David Daly, Salem Derisavi, Holger Hermanns, Joost-Pieter Katoen, Vinh Vi Lam, and Bill Sanders. On integrating the Möbius and MoDeST modeling tools. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*. IEEE Computer Society, IEEE Computer Society Press, June 2003.
4. Henrik Bohnenkamp, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. The MoDeST modeling tool and its implementation. In *Computer Performance Evaluation—Modelling Techniques and Tools (TOOLS 2003)*, volume 2794 of *LNCS*. Springer-Verlag, 2003.
5. D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Trans. on Software Engineering*, 28(10):956–969, 2002.
6. P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Trans. Inf. Theory*, 46(2), March 2000.
7. Malte Kampschulte. Evaluation of radio models for the analysis of a gossiping mac protocol. Bachelor thesis, RWTH Aachen University, 2010.
8. Joost-Pieter Katoen, Henrik Bohnenkamp, Ric Klaren, and Holger Hermanns. Embedded software analysis with MOTOR. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
9. A Roy and N Sarma. Energy saving in mac layer of wireless sensor networks: a survey. In *National Workshop in Design and Analysis of Algorithm (NWDAA)*. Tezpur University, India, 2010.
10. Nordic Semiconductors. *nRF2401 Single-chip 2.4GHz Transceiver Data Sheet*, 2002.
11. Haidi Yue, Henrik Bohnenkamp, and Joost-Pieter Katoen. Analyzing energy consumption in a gossiping MAC protocol. In *Proc. MMB '01*, volume 5987 of *LNCS*, pages 107–119. Springer, 2010.

Probabilistic Analysis of Embedded Systems

Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen

Abstract This chapter provides a gentle introduction into compositional modeling of systems that involve nontrivial real-time and probabilistic aspects. It revolves around the language MODEST, a Modelling and Description language for Stochastic and Timed systems. The language supports the compositional description of reactive systems while covering both functional and non-functional system aspects such as quantified component failure rates and hard and soft real-time. A running example illustrates the language constructs. Afterwards, this example is used to highlight different avenues to analyse such models that have been implemented in a tool suite. Among them, we find probabilistic timed model checking as well as discrete event simulation.

1 Introduction

There is a growing awareness among embedded software designers that the classical computer science approach—to abstract from physical aspects—is too limited and too restricted for contemporary and upcoming design challenges [1, 13]. Instead, abstractions of software that leave out "non-functional" aspects such as cost, efficiency, and robustness need to be adapted to current needs.

Embedded software controls the core functionality of many systems. It is omnipresent: it controls telephone switches and satellites, drives the climate control in our offices, runs pacemakers, is at the heart of our power plants, and makes our

Arnd Hartmanns
Saarland University – Computer Science, e-mail: arnd@cs.uni-saarland.de

Holger Hermanns
Saarland University – Computer Science, e-mail: hermanns@cs.uni-saarland.de

Joost-Pieter Katoen
University of Twente, RWTH Aachen University – Computer Science, e-mail: katoen@cs.rwth-aachen.de

cars and TVs work. Whereas traditional software has a rather transformational nature mapping input data onto output data, embedded software is different in many respects. Most importantly, embedded software is subject to complex and permanent interactions with its—mostly physical—environment via sensors and actuators. Software in embedded systems does not typically terminate and interaction usually takes place with multiple concurrent processes at the same time. Reactions to the stimuli provided by the environment should be prompt (timeliness or responsiveness), i.e., the software has to "keep up" with the speed of the processes with which it interacts. As it executes on devices where several other activities go on, non-functional properties such as efficient usage of resources (e.g., power consumption) and robustness are important. High requirements are put on performance and dependability, since the embedded nature complicates tuning and maintenance.

Embedded software is an important motivation for the development of modelling techniques that, on the one hand, provide an easy migration path for design engineers and, on the other hand, support the description of quantitative system aspects. This has resulted in various extensions of light-weight formal notations such as SDL (System Description Language) and UML (Unified Modeling Language), and in the development of a whole range of more rigorous formalisms based on, for example, stochastic process algebras, or appropriate extensions of automata such as timed automata [2] and probabilistic automata [16]. Light-weight notations are typically closer to engineering techniques, but lack a formal semantics; rigorous formalisms do have such a formal semantics, but their learning curve is typically too steep from a practitioner's perspective and they mostly have a restricted expressiveness.

This paper surveys MODEST [3], a description language that has a rigid formal basis (i.e., semantics) and incorporates several ingredients from light-weight notations such as exception handling, modularization, atomic assignments, iteration, and simple data types. The paper also illustrates advanced tool support, all by means of a running example.

MODEST is a *compositional* specification formalism: the description of complex behaviour is obtained by combining the descriptions of simpler components. This provides an elegant way to specify concurrent computations, inherited from process algebra. MODEST is enhanced with convenient language ingredients like simple data-structures and a notion of exception handling. It is capable to express a rich class of non-homogeneous stochastic processes and is therefore most suitable to capture non-functional system aspects. MODEST may be viewed as an overarching notation for a wide spectrum of prominent models in computer science, ranging from labeled transition systems to timed automata [2, 4], probabilistic variants thereof [5], and stochastic processes such as Markov chains and (continuous-time and generalised) Markov decision processes [8, 10, 15, 16].

MODEST takes a *single-formalism, multi-solution* approach. Our view is to have a single system specification that addresses various aspects of the system under consideration. Analysis thus refers to the same system specification rather than to different (and potentially inconsistent) specifications of system perspectives like in UML. Analysis takes place by extracting simpler models from MODEST specifications that are tailored to the specific property of interest. For instance, to check reachability

properties, a possible strategy is to "distill" an automaton from the MODEST specification and feed it into an existing model checker such as SPIN [11] or CADP [7]. For probabilisitic timed models, we implement a translational model checking approach [9], using the PRISM tool as backend. On the other hand, to carry out an evaluation of the stochastic process underlying a MODEST specification, we support discrete-event simulation.

2 Modelling with MODEST

In this section, we will introduce the MODEST language syntax and its semantic basis by modelling a simple communication scenario. We will introduce the language features and constructs step-by-step, starting with a very basic functional model which we then extend to include timed, probabilistic and stochastic behaviour. While the focus of this section is on the language syntax and the types of behaviours than can be modelled, we will also give brief insights into the underlying semantics where useful for a deeper understanding.

2.1 Syntax and Semantics Basics

The scenario we are going to model in MODEST is a simple communication setting where a sender continuously tries to send messages to some receiver over an unreliable channel that may lose, but not reorder or create messages. We will refine our models in the following sections; let us start with a very basic functional description of sender, receiver and channel for now.

In the MODEST language, being inspired by classic process-algebraic languages such as CCS, CSP and LOTOS, everything is a process. Processes can perform actions, and in this way transition into a different process. One of the most basic processes in MODEST is `tau`, which can perform a single action named *tau*, transitioning into a process than cannot do anything, which we denote by ✓ if it expresses a situation corresponding to "successful termination". Processes can also be given names, allowing them to be reused in other places.

Our first example, the `Receiver` process shown in Figure 1, uses the sequential composition construct `;` to combine several basic processes (that each just perform an action) into a sequence of processes that each still perform a single action, but then transition into the next one. While the semantics of the `;` construct is intuitively clear, a formal definition of the semantics of all constructs of the MODEST language is necessary for formal verification. The result of applying this semantics [3] to a given process is an automaton whose states correspond to MODEST processes, and whose edges are labelled with actions and represent the transitions between processes. For now, the resulting automaton is just a labelled transition system (LTS), the simplest submodel supported by the MODEST language.

```
process Receiver()
{
   receive_data;
   send_ack;
   report_data;
   Receiver()
}
```

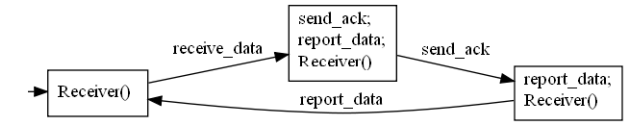


Fig. 1 The most basic model of a receiver

Figure 1 also shows the LTS corresponding to the `Receiver` process. The behaviour of the receiver is thus to—upon receiving some data from the communication channel—first send an acknowledgment back to the sender, then report the arrival of the message (presumably to some upper network layer), and finally start over again and wait for new data. In the following, we may sometimes omit location labels and certain parts of the edge labels for clarity; however, keep in mind that a location always corresponds to a MODEST process.

2.2 Nondeterminism

An important feature of many modelling formalisms, including MODEST, is to allow nondeterministic specifications. A nondeterministic choice is a choice between two different courses of action without any information about the likelihood of one of them or the circumstances that may lead to it. As such, it can be used to model the complete absence of knowledge about the actual behaviour of a system; it can be used to intentionally leave an implementation choice in a specification; and it allows an open model to react on stimulus from a yet unknown environment.

We use a nondeterministic choice for the latter purpose in our first model of the sender: To allow guaranteed delivery, the sender waits for an acknowledgment from the receiver that the message it just sent has arrived before moving on to the next one. Since the communication channels are lossy, the actual data or the receiver's acknowledgment may be lost, so the sender may have to repeat its transmission. We use a nondeterministic choice between receiving an acknowledgment and detecting that a message has been lost, which will eventually be resolved by the actual behaviour of the environment.

The MODEST model for the sender and the corresponding LTS is shown in Figure 2. The **`alt`** construct is used to specify the nondeterministic choice between receiving an acknowledgment and detecting message loss, the latter of which is encapsulated in a dedicated process `Timeout` that we specify as

```
alt { :: timeout_send  :: timeout_ack }
```

for this first model. Note that instead of calling `Sender()` after the receipt of an acknowledgment or timeout to start over again, we chose MODEST's loop construct, **`do`**, which causes some process to be repeated ad infinitum or until it issues the special **`break`** action.

```
process Sender()
{
   do {
      :: send_data;
         alt {
            :: receive_ack
            :: Timeout()
         }
   }
}
```

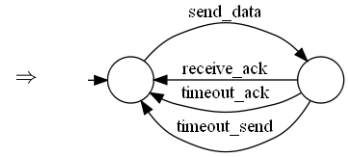


Fig. 2 The most basic model of a sender

```
process Channel()
{
   receive;
   alt {
      :: send
      :: drop
   };
   Channel()
}
```

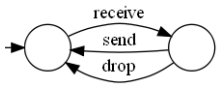


Fig. 3 A simple lossy communication channel

The only component of our communication scenario that still needs to be modelled is the channel. Its model, shown in Figure 3, again uses a nondeterministic choice. However, this time, it represents an "absence of knowledge" case: We do not know anything about the channel except for the fact that it *may* lose messages, so after receiving a message on one end, we just nondeterministically allow both possibilities—successful propagation of the message to the other end (`send`) or message loss (`drop`).

2.3 Processes Running in Parallel

Although we now have MODEST processes that represent all components of our communication scenario—sender, receiver and channel—we still need to obtain a single model for the whole system that also includes the interactions between the different components. Since these components usually represent distinct physical entities that mostly run independently from each other, possibly with different processing speeds, their composition is best represented by letting them run in parallel without further restrictions, allowing their actions to occur interleaved in any order. Only if the components actively interact with each other may we need to model a synchronisation between them.

In MODEST, this kind of parallel composition is implemented by the **`par`** construct: The parallel composition of n processes P_1 to P_n, **`par`** $\{ :: P_1 \ \ldots \ :: P_n \}$, allows the processes to perform their actions in any order, unless an action is shared by at least two processes. In that case, in order to perform such a shared action, all the processes that contain the action have to perform it at the same time, as a sin-

```
par {
   :: Sender()
   :: relabel {receive, send, drop} by {send_data, receive_data, timeout_send}
      Channel() // Channel from sender to receiver
   :: relabel {receive, send, drop} by {send_ack, receive_ack, timeout_ack}
      Channel() // Channel from receiver to sender
   :: Receiver()
}
```

Fig. 4 MODEST code for the parallel composition of sender, receiver and communication channels

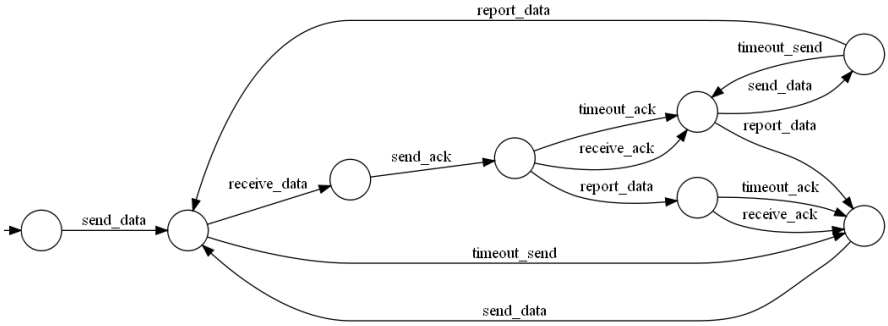


Fig. 5 LTS of the parallel composition of sender, receiver and communication channels

gle step. This synchronisation on shared actions allows us to model communication between processes.

Figure 4 shows the parallel composition of the sender, one instance of the `Channel` process to model the channel from sender to receiver, another instance to model the channel from receiver to sender, and the receiver itself. We use the **`relabel`** construct to rename actions in such a way that sender and receiver synchronise with the two channels as intended.

In our example, the LTS corresponding to the parallel composition (Figure 5) is still relatively small, but this is only because the individual processes synchronise very often. In theory, the size of the LTS of a parallel composition is only bounded by the product of the numbers of states of the individual processes, and may thus grow large very quickly.

2.4 Data Exchange

The basic model of the communication scenario we developed so far has one serious problem: The receiver will report every receipt of a message from the channel, even if it is just a retransmission after a lost acknowledgment. This is clearly undesirable behaviour, but in order to fix it, we need a way to distinguish different messages. A classic solution is that of the *Alternating Bit Protocol*: If we can guarantee that there is at most one message in transit at any time, it suffices to include a single bit in every message that is flipped between subsequent messages. The receiver stores the

```
bool channel_bit;

process Sender()
{
   bool bit;

   do {
      :: send_data {= channel_bit = bit =};
         alt {
            :: receive_ack {= bit = !bit =}
            :: Timeout()
         }
   }
}

process Receiver(bool last_bit)
{
   bool bit;

   receive_data {= bit = channel_bit =};
   urgent send_ack;
   alt {
      :: when(bit != last_bit) report_data
      :: when(bit == last_bit) tau
   };
   Receiver(bit)
}
```

Fig. 6 Transmission of an alternating bit between sender and receiver

value from the last reported message, and if a new message arrives with the same value, it clearly is a retransmission.

MODEST supports data in the form of global and process-local variables, including parameters to processes, which can be of Boolean, integer, bounded integer or real type, arrays thereof, or user-defined composite types. In order to transmit a single bit, a Boolean variable is sufficient. The necessary modifications to the model are shown in Figure 6: A global variable `channel_bit` is added that represents the bit of the message that is currently being transmitted, and sender and receiver get local variables or parameters to keep track of the current and previous bits.

Aside from the variable declarations, we see two new language features in the modified model, namely assignments and the **when** construct, or *guard*. Assignments are associated with an action and enclosed in brackets ({= ... =}). They are executed atomically when the action is performed; in particular, the textual order of the assignments in MODEST code inside a {= ... =} block does not matter. The variables can then be used to put constraints on when an action can actually be performed by using the **when** construct.

2.5 Time

Up to this point, the detection of message loss was implemented by the process

```
alt { :: timeout_send  :: timeout_ack }
```

```
const int TD; // maximum channel transmission delay
const int TS; // sender timeout

process Timeout()
{
   clock c;

   when(c >= TS) urgent(c >= TS) tau // timeout: retransmit
}

process Channel()
{
   clock c;

   receive {= c = 0 =};
   alt {
      :: urgent(c >= TD) send
      :: urgent tau // silently drop the message
   };
   Channel()
}
```

Fig. 7 Adding a transmission delay and realistic timeout detection

where the two `timeout_*` actions just synchronised with the `drop` actions that the channels performed when they lost a message. This is, of course, a significant abstraction from reality where message lost is usually detected by the passage of a certain amount of time, after which a *timeout* occurs. Fortunately, time is supported in MODEST as well, in a way that is almost identical to how time is modelled in Timed Automata (TA, see Chapters 2 and 3), so we can now make our model more realistic w.r.t. timeouts.

Just like TA, MODEST has clock variables, which can be also be used in expressions in guards and *deadlines*. Deadlines (or *urgency constraints*, written in MODEST as `urgent(`*d*`)` where *d* is an expression of Boolean type) are used to constrain the passage of time; they are the MODEST analogue to invariants in TA. In contrast to invariants, however, they are not associated with states, but with the actions possible in a process. If the deadline of any action possible in a process becomes true (we say that the action becomes *urgent*), time cannot pass any more and some edge has to be taken.

Figure 7 shows how to use these new constructs to put realistic timeouts into our model. The transmission of a message through a channel now takes up to `TD` time units (which is achieved by resetting clock `c` to zero and then prefixing the channel's `send` action with `urgent(c >= TD)`), and the `Timeout` process is modified such that it actually waits some time—precisely `TS` time units—before terminating and thereby causing a retransmission, leaving the bit unchanged. The corresponding automaton is shown in Figure 8—note that guards, deadlines and assignments are stored symbolically on the transitions.

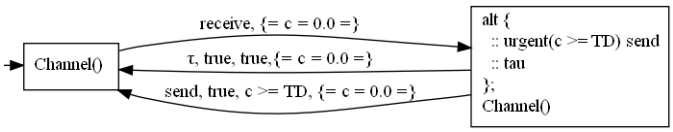


Fig. 8 Automaton for the channel with transmission delay

```
process Channel()
{
   clock c;

   receive palt {
      :98: urgent(c >= TD) send
      : 2: urgent tau // silently drop the message
   };
   Channel()
}
```

Fig. 9 A timed probabilistic lossy channel

2.6 Probabilistic Choices

Although our model of the communication scenario has become significantly more realistic through the addition of "real" timeouts, there still is one more problem with the current channel model: The decision whether or not to lose a message is currently a nondeterministic one. While this is a good model for complete absence of information about the frequency of message losses, one possible resolution of this choice is to always drop the message, i.e., a completely non-functional channel. In order to avoid this situation, we either need so-called fairness assumptions which may enforce that eventually some message is not lost, or we just have to put information about the frequentness of message loss into the model.

Let us pursue the latter option for this example. Experiments may have shown that for a typical implementation of this kind of channel, a single message is lost with a probability of 2%, independent of any other conditions. This *probabilistic choice* between two options is available in MODEST via the **`palt`** construct, as shown in Figure 9: After receiving a message at one end of the channel, it is now lost with probability $\frac{2}{98+2} = 2\%$, and it successfully arrives at the other end (possibly with some delay) with probability 98%. The probabilities used in the **`palt`** construct are actually *weights*, so we could equivalently have written 49 and 1 instead of 98 and 2, and they are not restricted to constant values as in the example, but can be arbitrary expressions.

For the previous examples, the semantics of our model was an automaton with variables and edges that consisted of an action, a guard, a deadline, and assignments. When we add the **`palt`** construct, however, the model becomes slightly more involved because we need to represent the probabilistic branching in the automaton. As shown in Figure 10, edges can now target multiple locations, with a probability given for each branch. (Formally, the edges now relate a source location with a probability distribution over target locations and assignments.) This is also why

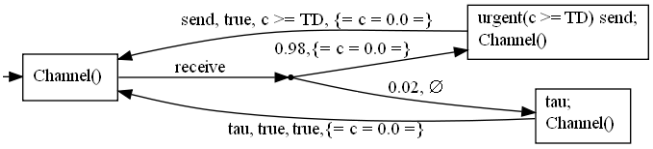


Fig. 10 Automaton for the channel with transmission delay and probabilities

the `palt` construct has to be prefixed with an action—it needs an existing edge to which it can add probabilistic branching.

2.7 Random Timing

The model built so far for our communication scenario could already be considered complete; however, let us add one more refinement by explicitly modelling the arrival of new messages for the sender. We will add a queue of messages that periodically grows and out of which the `Sender` process takes a message as soon as the transmission of the previous one is successfully completed.

The time between the arrivals of "customers", in our case messages, to a queue is usually assumed to be exponentially distributed with some rate parameter λ. To model these delays, we can take advantage of the possibility to draw samples from random variables distributed according to a number of predefined distributions in MODEST: for example, the statement

```
{= delay = Exponential(3.14) =},
```

assigns a value to the variable `delay` (which is of type real) that is randomly chosen according to the exponential distribution with rate 3.14. We can then use this variable in guards and urgency constraints to achieve a delay that is of exponentially distributed length, as long as we resample before using it again:

```
urgent(c >= delay) when(c >= delay) ...
```

This now allows us to implement the queue of messages as shown in Figure 11. We have also modified the sender by adding a `get_data` action that synchronises with the `Queue` process, taking out one message.

2.8 More Syntax

The full communication scenario model (Figure 11) concludes our tour of the MODEST language. We have seen that MODEST can be used to build models of real-time systems with probabilistic or stochastic behaviour, including how its process-algebraic nature can be used to specify a complex system in terms of its natural components, which themselves remain small and easy to understand.

```
action get_data;
action receive_data, send_ack, report_data;
action send_data, receive_ack;
action receive, send;

const int TD; // maximum channel transmission delay
const int TS; // sender timeout
const real AR; // data arrival rate

bool channel_bit;

process Timeout() { ... }

process Sender(bool bit)
{
   get_data;
   do {
      :: urgent send_data {= channel_bit = bit =};
         alt {
            :: receive_ack; urgent break
            :: Timeout()
         }
   };
   Sender(!bit)
}

process Receiver(bool last_bit) { ... }

process Channel() { ... }

process Queue()
{
   clock c;
   int items;
   real delay;

   do {
      :: urgent(c >= delay) when(c >= delay)
         {= items += 1, delay = Exponential(AR), c = 0 =}
      :: when(items > 0) urgent(items > 0)
         get_data; urgent {= items -= 1 =}
   }
}

par {
   :: Queue()
   :: Sender(false)
   :: relabel {receive, send} by {send_data, receive_data}
      Channel() // Channel from sender to receiver
   :: relabel {receive, send} by {send_ack, receive_ack}
      Channel() // Channel from receiver to sender
   :: Receiver(true)
}
```

Fig. 11 The full model

MODEST is an expressive language, and although we essentially covered the types of behaviours that are expressible in MODEST, we were not able to present all of its syntactic features. Most notably, it also support exceptions as known from modern programming languages that can be thrown at some point in a model and be caught at another, but also several useful shorthands that, for example, allow the use of invariants as known from timed automata instead of deadlines.

Submodel	Probability distributions?	Clocks/Time?	Nondeterminism?
STA	arbitrary	arbitrary	yes
GSMP	arbitrary	arbitrary	no
PTA	finite	integer bounds	yes
TA	none	integer bounds	yes
PA/MDP	finite	no	yes
LTS	none	no	yes
CTMDP	exponential + finite	exponential delays	yes
CTMC	exponential + finite	exponential delays	no
DTMC	finite	no	no

Table 1 MODEST submodels

3 Analysing MODEST Models

The ultimate goal of modelling any kind of system is to be able to analyse the model, and in particular verify the presence or absence of certain good or bad properties. Due to the enormous expressiveness of MODEST, no currently known analysis technique can be used for arbitrary MODEST models. However, certain submodels of MODEST (Table 1) can easily be identified, e.g. by the absence of certain language constructs, and efficient analysis techniques targeted to several of these submodels exist:

Two very general submodels underlying MODEST are probabilistic timed automata (PTA, [5]) and generalized semi-Markov processes (GSMP, [8]). The STA corresponding to a MODEST model is a PTA if only probability distribution with finite support set occur. The complete model of our running example from the previous section is thus not a PTA because it uses the exponential distribution, but the one developed up to Section 2.6 is. We will show how to analyse this model in the first part of this section. On the other hand, regardless of the probability distribution appearing, a MODEST model corresponds to a GSMP, provided it is fully deterministic. Unfortunately, this condition is difficult to assure — in particular, the parallel composition is not closed under determinism: fully deterministic processes running in parallel may behave nondeterministically.

3.1 Model-Checking PTA Models

For the formal verification of MODEST models that correspond to PTA, a set of properties to be checked has to be defined. As PTA combine probabilistic and real-time behaviour, we can refer to both probabilities and time in these properties. Some classes of probabilistic timed properties that can efficiently be verified are

- *probabilistic reachability properties*: “What is the probability of ever reaching an error state?”,

- *probabilistic time-bounded reachability properties*: “What is the probability of reaching an error state within *n* time units?”, and
- *expected-time reachability properties*: “What is the expected time until an error state is reached?”.

Because PTA can contain nondeterminism, the answers to all of these properties depend on how the nondeterminism is resolved. All of them thus exist in a *maximum* and *minimum* variant: If asked for the maximum (minimum) probability of reaching an error state, all possible ways to resolve nondeterministic choices are considered and the highest (lowest) probability is returned.

Let us now analyse the model of the communication scenario developed up to Section 2.6 in terms of correctness and performance. The most basic correctness criterion for such a communication protocol is that the probability of eventually succeeding, which we may, for example, detect by the receipt of an acknowledgment, is 1. This can be specified inside the model as

```
property success = P(<> did(receive_ack)) >= 1.0;
```

For $\texttt{TD} = 1$ and $\texttt{TS} = 4$, this property is indeed satisfied. Now that we are confident that the protocol used is correct, we can study performance aspects. A requirement may for example be that, in the worst case, the probability of completing a transmission within 7 time units is close to 100 %. For the same values for TD and TS, we will see that it is actually 99.843 % using

```
property Pmin(<> did(receive_ack) && time <= 7.0);
```

Lastly, we can also find out what the actual expected time until successful transmission is. Using the following properties—

```
property success_time_min = Tmin(did(receive_ack));
property success_time_max = Tmax(did(receive_ack));
```

—we see that it lies between 1.649 and 2.165 time units, depending on how the nondeterministic choice of the actual transmission delay in the channel is resolved.

3.2 Discrete-Event Simulation for GSMP Models

Models using probability distributions with infinite support, such as the exponential distribution, can be analysed using discrete-event simulation. Since discrete event simulation relies on the execution and evaluation of large batches of *concrete* traces of a model, this model either needs to be deterministic (i.e., a GSMP), or the nondeterminism has to be resolved in some way specified by the user in order to obtain a GSMP.

There are many different methods to resolve nondeterminism, and the particular method employed may skew the analysis results in unexpected and sometimes counterintuitive ways. For example, two distinct possibilities to resolve nondeterminism over time – such as for the transmission delay in our running example – are to always choose the earliest or the latest possible point in time. For our example,

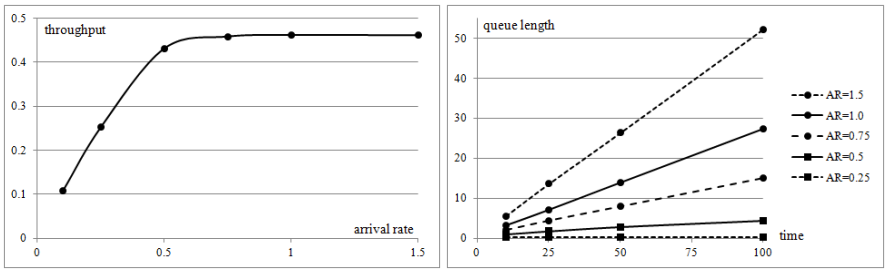


Fig. 12 Simulation results for the communication scenario

the results we obtain this way for the expected time to success do coincide with the actual lowest and highest values, but it is easy to construct a model where, for example, the inverse is the case. Understanding that the method to resolve nondeterminism can influence the results and taking this into account when interpreting the results is thus crucial.

When analysing the final model of our communication scenario via discrete-event simulation, we can still check the properties presented for the PTA model above, but also new measures such as the actual throughput of the system in terms of successfully transmitted messages per time unit and the average length of the queue over time. The former can be achieved by simply counting the number of times that the `report_data` action is performed via MODEST's support for transition rewards:

```
increment count for {report_data} by {1} Receiver(true)
```

At the end of a simulation run, we then observe the value of `count/time`. To measure the average queue length, we need to use a rate reward that grows (linearly) over time at the speed of the current queue length. We can specify this in MODEST by introducing a new variable `items_reward` of type real and setting its derivative to be the queue length:

```
der(items_reward) = items;
```

Again, we let the simulator observe the value of `items_reward/time` at the end of each simulation run.

Discrete-event simulation is usually used to perform a large number of simulation runs and then compute a statistical evaluation of the collected results. To obtain numbers for the measures introduced above, we collect 1000 random traces for different model time lengths with uniformly random resolution of nondeterminism, but a deterministic transmission delay of 1 time unit and $\mathsf{TS} = 4$; the results are plotted in Figure 12. We see that the throughput of the system is limited to about 0.46 messages per time unit due to the communication delays and the lengths of the timeouts in case a message is lost; as expected, the queue length appears to grow without bounds for arrival rates close to or larger than that value; slowly so for $\mathsf{AR} = 0.5$, but already significantly for $\mathsf{AR} = 0.75$.

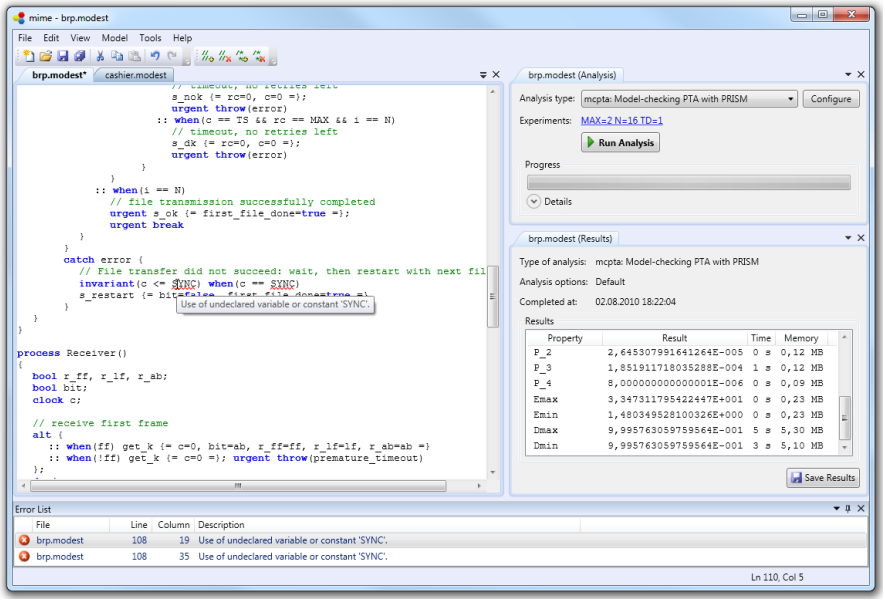


Fig. 13 mime, the integrated modelling environment for MODEST

3.3 Tool Support

The analysis of MODEST models is today supported by two sets of tools: The newly developed *Modest Toolset* and the original *Modest Tool Environment*, MOTOR.

3.3.1 The Modest Toolset

The MODEST Toolset, developed at Saarland University, currently consists of three tools: *mcpta*, which allows model-checking MODEST models of PTA [9]; *modes*, a discrete-event simulator for MODEST, and *mime*, an integrated modelling environment that combines a MODEST editor with syntax and error highlighting with direct access to the model analysis capabilities of mcpta and modes (Figure 13). The MODEST Toolset is cross-platform and can be downloaded at

`www.modestchecker.net`

mcpta transforms MODEST models corresponding to PTA into probabilistic, but untimed models and hands these over to the PRISM probabilistic model-checker [14] for analysis. This process is fully automated, and all classes of properties introduced in Section 3.1 are supported.

modes' focus is on discrete-event simulation with a sound treatment of nondeterminism. Its default behaviour is therefore to reject nondeterministic models, but it allows the user to override this behaviour by explicitly choosing one out of a set

of predefined resolution methods. Additionally, modes can be configured to detect and ignore certain kinds of *spurious* nondeterminism, i.e. choices that do not actually influence the final result. When this option is used, modes can analyse (certain) nondeterministic models while the user can rest assured that the values obtained are unaffected by any particular resolution of nondeterminism.

3.3.2 MoTor and Möbius

MOTOR [12] is the original set of tools designed to interface MODEST with different existing analysis backends such as CADP [7] for functional verification and MÖBIUS [6] for discrete-event simulation. Today, the MÖBIUS backend is mostly used for high-performance and distributed simulation of MODEST models; see the following chapter for an extensive case study. In contrast to modes, nondeterministic choices over different actions are always resolved in a uniformly probabilistic way, while a *maximal progress* semantics is used for nondeterministic delays (that is, as soon as an action is possible, it is taken, even when the model would allow more time to pass).

MOTOR is available from the University of Twente at

`fmt.cs.utwente.nl/tools/motor/`

3.3.3 Analysing other Submodels

While not yet supported by the tools introduced above, several other submodels of MODEST are easy to analyse with well-established model-checking tools once the MODEST code is translated into the respective tool's formalism: For example, the timed automata subset could be analysed using UPPAAL, while the one corresponding to continuous-time Markov chains (CTMC) could also be translated to PRISM. Since Markov decision processes (MDP) and discrete-time Markov chains (DTMC) are special cases of PTA, mcpta can already be used in combination with PRISM to analyse these submodels.

4 Summary

Summary/conclusion **HH: TODO HH:**.

References

1. IEEE Computer, special issue on embedded systems, 2000.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

3. Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.
4. Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.
5. Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the IEEE-1394 root contention protocol with KRONOS and PRISM. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.
6. Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Software Eng.*, 28(10):956–969, 2002.
7. Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.
8. Peter W. Glynn. A GSMP formalism for discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 14–23. 1989.
9. Arnd Hartmanns and Holger Hermanns. A Modest approach to checking probabilistic timed automata. In *QEST '09: Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, pages 187–196, Washington, DC, USA, 2009. IEEE Computer Society.
10. Holger Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.
11. Gerard J. Holzmann. Software analysis and model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.
12. Joost-Pieter Katoen, Henrik C. Bohnenkamp, Ric Klaren, and Holger Hermanns. Embedded software analysis with MOTOR. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 268–294. Springer, 2004.
13. Edward A. Lee. Embedded software. *Advances in Computers*, 56:56–97, 2002.
14. David Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
15. M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Inc., New York, 1994. A Wiley-Interscience Publication.
16. Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

Model-Based Testing

Brian Nielsen and Jan Tretmans

Abstract Model-based testing is one of the promising technologies to increase the efficiency and effectiveness of testing. In model-based testing, a model specifies the required behaviour of a system, and test cases are automatically generated from this model. This chapter gives an introduction to model-based testing. First, the concepts and aspects of model-based testing are discussed in general. Then these concepts are elaborated for simple, state-based models expressed as labeled transition systems, and subsequently for the more complex model of Timed Automata.

1 Introduction

Testing plays an important role in the demand for improved quality of systems and software. Testing, however, is often a manual and laborious process without effective automation, which makes it error-prone, time consuming, and very costly. Estimates are that testing takes 30-50% of the total software development effort. This leads to the quest for more effective and more efficient testing.

Model-based testing is a promising new technology that can contribute to increasing the efficiency and effectiveness of the testing process. In model-based testing, a model is the starting point for testing. This model expresses precisely and completely what the *system under test* (SUT) should do, and should not do and, consequently, it is a good basis for systematically generating test cases. Model-based testing makes it possible to generate a set of test cases, including test oracles, completely automatically from a model of required SUT behaviour. In this way, model-based testing allows for test automation that goes well beyond the mere automatic

Brian Nielsen
Aalborg University, Aalborg, Denmark, e-mail: bnielsen@cs.aau.dk

Jan Tretmans
Embedded Systems Institute, Eindhoven, The Netherlands, e-mail: jan.tretmans@esi.nl

execution of manually crafted test scripts, which is the current state of practice. And if the model is valid, i.e., expresses the system requirements accurately, then all these algorithmically generated tests are provably valid, too.

Classification of Model-Based Testing

There are many different kinds of testing depending, for example, on the quality aspects being tested, whether the specification or the code is the starting point of testing, and whether we have access to the internal details of the SUT or only to its external interfaces. Likewise there are different kinds of model-based testing. In this chapter we consider *specification-based*, *black-box* testing of *functionality*.

Functionality testing involves checking whether the system correctly does what it should do in terms of input/output behaviour, i.e., correct and timely responses to given stimuli. Other quality characteristics that might be tested with other methods are, e.g., performance, usability, reliability, robustness, or security properties.

The starting point for our testing is the *specification*, which prescribes what the SUT should, and should not do. Our specification is given in the form of a behavioural model with which the behaviour of the SUT must comply. Moreover, the testing is *black-box*, which means that the SUT is seen as a black box without internal detail, which can only be accessed and observed through its external interfaces, as opposed to white-box testing, where also the internal structure of the SUT, i.e., its code, is used for testing.

Finally, our approach is *rigorous* in the sense that the models of the SUT are given in a well-defined notation, and they define unambiguously what correct and incorrect SUT behaviour is.

Overview

This chapter discusses rigorous, specification-based, black-box, model-based testing of functionality. First, Section 2 presents general concepts, ingredients, and phases of model-based testing. Then Section 3 elaborates these for simple, state-based models expressed as labeled transition systems. Section 4 uses the full power of Timed Automata, introduced in Chapters **??** and **??**, to express models, and shows how the model-based testing tool UPPAAL-TRON is used to test real-time properties. Section 5 concludes with benefits, open issues, and perspectives of model-based testing. This chapter prepares for Chapters **??** and **??**, where case studies of model-based testing are discussed.

2 Model-Based Testing

In model-based testing there is a *system under test* (SUT), there is a *model* that prescribes how the SUT shall behave, and there is the question whether the behaviour of the SUT *complies with* the behaviour expressed in this model. To check compliance *test cases* are constructed from the model through *test generation* and *selection*. *Test execution*, and *analysis* consisting of comparing actual test outcomes with expected ones according to the model, lead to a *verdict* whether the SUT indeed complies with the model.

This section discusses these concepts of model-based testing in general; the next two sections will elaborate them for particular model-based testing approaches using labeled transition systems and Timed Automata, respectively.

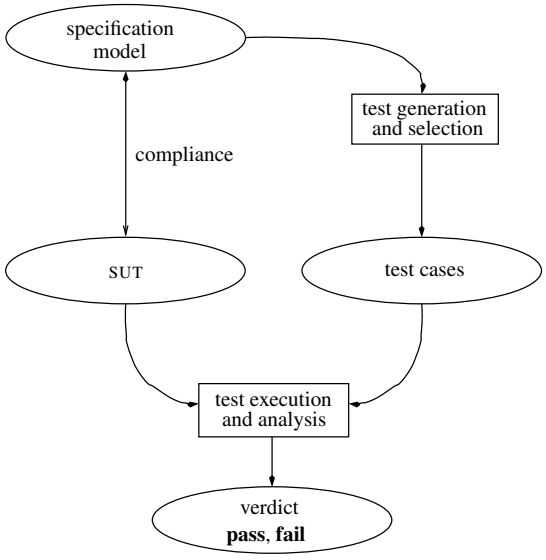


Fig. 1: The process and concepts of model-based testing.

System Under Test

Since the system to be tested (SUT) is treated as a black box exhibiting behaviour, a tester can only control and observe the SUT via its external interfaces, where stimuli and inputs can be provided and responses and outputs can be observed. Consequently, identifying these test interfaces of the SUT, in terms of, e.g., ports, programming interfaces, message exchanges, and communication lines, is an important first step for (model-based) testing. The occurrence of input and output actions on these

interfaces, together with their inter-dependencies, ordering, and timing, constitutes the behaviour of the SUT. It is this behaviour of the SUT that will be tested.

Model

The second main ingredient for model-based testing is the *model*. A model is an abstraction of reality, a description focusing on some aspects of the system, while leaving out many details. A model for model-based testing focuses on behaviour. It specifies which input and output actions occur, how outputs depend on inputs, and how they are ordered and timed. Various notations for models exist such as different forms of (finite) state machines and state charts. The next section uses a very basic, state-based notation, called labeled transition systems, to express models; Section 4 uses Timed Automata.

Compliance

The goal of model-based testing is to check whether the actual behaviour of the SUT complies with the behaviour expressed in the model. To relate an SUT to a model, the first, static step is to map (or *adapt*) the real interfaces, inputs, and outputs of the SUT, to the abstract descriptions of the interfaces, inputs, and outputs in the model, e.g., to map the concrete socket connection $\langle 192.168.1.1, 7890 \rangle$ to the abstract port p used in the model, and to map the concrete message with bit pattern 01010101 to the abstract message `Initialize`.

The second, dynamic step of relating a model to an SUT is stating precisely when an SUT correctly implements the behaviour described in the model. An *implementation relation*, or *conformance relation*, defines the conditions under which the behaviour of an SUT complies with the behaviour prescribed in its model. Such a relation is necessary because a model in itself does not completely define which SUTs are correct. A model expresses behaviour, but it does not define whether the SUT *must exactly implement* all behaviours described in the model, whether it *may leave out* some behaviours, or whether it *may add* some additional, non-specified behaviours. An implementation relation typically answers such questions. Implementation relations for labeled transition systems and Timed Automata are further discussed in Sections 3 and 4.

Test Generation

A *test case* specifies the experiment that is performed on the SUT. It specifies the inputs, or stimuli, to be supplied to the SUT, the outputs, or responses, expected from the SUT, and the ordering and timing of these inputs and outputs. In model-based testing, test cases are algorithmically generated from the model.

The generated test cases should exactly detect those behaviours that are not correct with respect to the model according to the implementation relation. A collection of test cases, called a *test suite*, is *sound* if all correct SUTs pass the test suite, i.e., there are no false alarms. The other way around, if no erroneous SUT passes a test suite, the test suite is called *exhaustive*, i.e., there are no failure escapes.

Soundness is a minimal requirement for test suites. Exhaustive test suites do not exist in practice, because detecting all potential failures would require an infinite number of infinitely long test cases for any non-trivial SUT: "Program testing can be used to show the presence of bugs, but never to show their absence!" [2]. Yet, for reasoning about model-based test-generation algorithms, both soundness and exhaustiveness are important concepts. A theoretically exhaustive test generation algorithm will eventually detect all possible errors if the time of testing is unbounded. Practically, this means that every error has a non-zero probability of being detected, i.e., there are no errors that are fully undetectable.

Test Selection

A sound and exhaustive test generation algorithm can generate many more test cases than can ever be executed. Even testing the addition of two 32-bit integers, which could easily be automated by writing a test generation algorithm that enumerates all $2^{32} \times 2^{32} = 1.8\ 10^{19}$ possible test cases, would require $584,542$ years of test execution if one test case would take 1 μsec.

Practical test generation algorithms use *test selection criteria* to generate a feasible and executable selection of sound, but not exhaustive test cases. The aim is to select test cases in such a way that they provide a high chance of detecting failures, and give confidence that an SUT that passes the test cases is indeed complying, within given constraints of testing time and effort.

Selection criteria, also referred to as test adequacy criteria, are based on heuristics, experience, gut feeling, and expert domain knowledge, so human influence is prominent. Structural selection criteria for model-based testing express when a model is considered sufficiently *covered* by test cases. Examples are state- and transition coverage for state-based models. Selection criteria may also be specific for a particular model or domain, such as a domain expert having knowledge about particular critical behaviours.

For the addition of two 32-bit integers the classical *equivalence partitioning* (EP) and *boundary value analysis* (BVA) test-selection criteria are commonly used [7]. EP divides the input and output domains into classes of values that behave in the same way. BVA states that values on the boundaries of these classes have higher test value. For addition, typical EP input classes are positive and negative integers, and output classes are valid and overflow outcomes. For BVA, typical additional test values are -1, 0, 1, -2^{31}, and $2^{31}-1$. Taking all combinations of these values leads to a reduced test suite of about 50 test cases covering addition following the selection criteria EP and BVA.

Test Execution and Analysis

We use the term *test execution* for applying a test case to an SUT, resulting in some observations. To execute a test case, the abstract actions of the test case must invoke the concrete interfaces of the SUT, and the concrete observations made on the SUT must be interpreted in terms of the abstract model actions. This is called *adaptation*, and the component in the test execution environment taking care of this is usually called the *adapter*.

After test execution the test outcomes must be analysed to investigate whether they are as expected, or not. If the actual outcome complies with the expected outcome according to the model, then this is indicated with the verdict *pass*. If the outcome is different the verdict is *fail*. A third verdict *inconclusive* is used to indicate that the outcome is correct but different from what was desired.

3 Model-Based Testing with Labeled Transition Systems

One of the theories for model-based testing is the **ioco**-testing approach, where models are expressed as labeled transition systems, and compliance between the SUT and the model is expressed using the **ioco**-implementation relation. This approach provides a well-defined foundation for model-based testing, and it has proved to be a good basis for several practical model-based test generation tools and their application. In this section we introduce the **ioco**-testing approach using examples. For a complete treatment we refer to [8].

3.1 Labeled Transition Systems

A *labeled transition system* is a structure consisting of states representing the states of the system, and labeled transitions between states modeling the actions that a system can perform, analogous to a timed automaton but without explicit time; see Chapters **??** and **??**. Typically, the actions represent the interactions of the system with its environment, such as inputs and outputs. Inputs are the stimuli provided to the system; they are under control of the system's environment. Outputs are the system's responses to these stimuli; they are under control of the system itself. We will decorate inputs with '?' and outputs with '!'. The observable behavior of a system is captured by the sequences of actions that a system is able to perform. Such sequences of inputs and outputs are called *traces*.

Example 1. Figure 2 presents seven labeled transition systems all of them modeling coffee machines. Machine s has two states: s_0, the initial state, and s_1. Upon pressing the button, which is modeled by the abstract input action *but*?, s takes the transition from s_0 to s_1. This is written as $s_0 \xrightarrow{but?} s_1$. In s_1 there are two possible outputs: either

coffee is produced, modeled as *cof*!, or tea is output, modeled as *tea*!. Both bring the machine back into its initial state s_0. The observable behaviour of s is expressed by its set of traces, of which there are infinitely many:

ε (the empty trace)	*but*?*cof*!*but*?
but?	*but*?*cof*!*but*?*tea*!
but?*cof*!	*but*?*tea*!*but*?*cof*!*but*?*tea*!
but?*tea*!	...

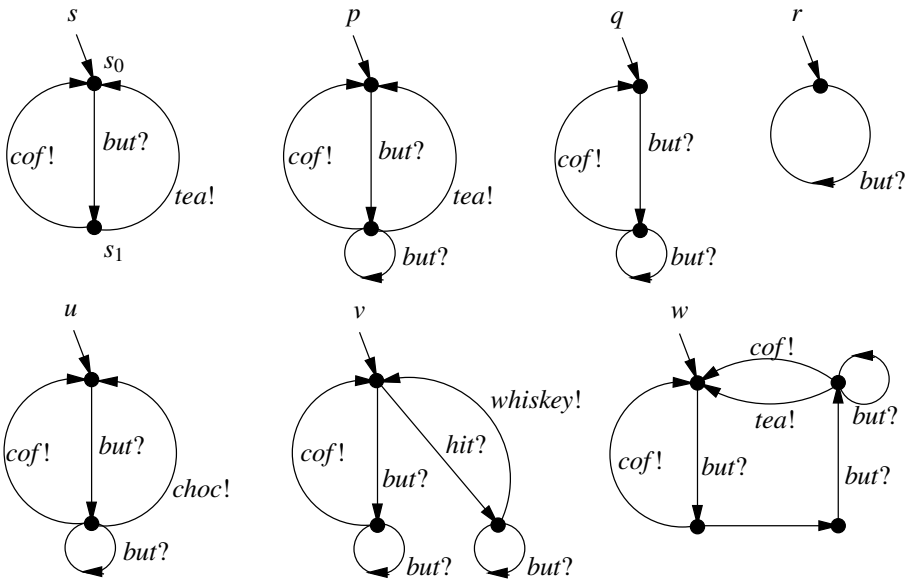


Fig. 2: Labeled transition systems.

In the **ioco**-approach a specification is expressed as a labeled transition system with given sets of inputs and outputs. The SUT, on the other hand, is a black-box performing behaviour that is observable at its interfaces. It is a software program, a physical system, or a combination of these. Yet, we can view the behaviour of the SUT as a labeled transition system that communicates with its environment using the same inputs and outputs as its specification. This allows to reason about the SUT's behaviour as if it were a labeled transition system.

Moreover, SUTs have the property that inputs can always be performed (although 'performing' may mean 'silently ignoring them'). We call a transition system where all inputs can be performed in all states, *input-enabled*. Together, this means that we can reason about the behaviour of SUTs as input-enabled labeled transition systems.

Example 2. In Figure 2, s is not input-enabled in state s_1, but p is: *but*? is accepted in s_0, and in s_1 without state change. Also the other systems in Figure 2 are input-enabled, and thus can be viewed as representing SUTs.

The examples in Figure 2 are very simple systems. Realistic systems may have millions or billions of states. Representing them as graphs is then not an option. An implicit representation in the form of a language of which the semantics can be expressed as labeled transition system is then used. The principles remain the same, only the representation changes. One such a language is Timed Automata, which, by using parallel automata, clocks, and variables, is able to represent very large transition systems in a concise way; this will be illustrated in Section 4.

Nondeterminism

Not all actions that a system can perform are observable. A system may perform an internal computation leading to an internal transition, or the communication between two system components may result in a transition that is not visible to the system's environment. An internal transition is written by omitting the label from the transition.

Internal transitions occur autonomously and invisibly within the system. This implies that the tester cannot precisely know, based on observing inputs and outputs, in which state the system currently is, and how the system will proceed with its behaviour: the system is *nondeterministic*. Also two transitions with the same label from the same state lead to nondeterminism.

In addition, a system can have *output-nondeterminism* when a state has two outgoing transitions with different output labels. Since outputs cannot be controlled by the system's environment, the choice which output transition to take is autonomously made by the system. This means that a tester cannot predict which of the outputs will occur, yet, after the output has occurred the state is known again.

Nondeterminism may occur both in specifications and in SUTs, and though it may seem inconvenient and leads to some complexity, it is useful in various ways. In a specification, nondeterminism is used to express freedom of implementation, that is, the specifier leaves different options open and does not uniquely prescribe the required behaviour, either because this is his deliberate choice, or because he has insufficient knowledge to specify all details. Other reasons of nondeterministic behaviour, both in models and in SUTs, are internal computations, invisible communications, restricted access to interfaces, and abstraction. Nondeterminism implies that the tester cannot uniquely predict in which state the SUT is, and which outputs it shall provide.

Example 3. Coffee machine w in Figure 2 has an unobservable transition, which has the effect that after pushing the button *but*? we do not know in which state the system is, and whether *cof*! will be produced or not. Output-nondeterminism occurs, for example, in s: after *but*? the machine produces either coffee *cof*! or tea

tea!. Actually, *s* specifies a hot-drinks machine leaving the choice between various hot drinks open.

Nondeterminism in a model leads to different possible behaviours that are all correct. If later test cases are generated from the model then these test cases must accept these different, correct behaviours. Moreover, if an SUT behaves nondeterministically then executing the same test case twice may lead to different outcomes. This means that test execution must be repeated a number of times in order to get confidence that all possible SUT outcomes have actually been observed.

Example 4. A particular communication link may split large messages in smaller chunks depending on the link's available resources. A specifier will not know the details of the splitting algorithm nor the status of available resources to specify precisely how messages are split into chunks. What can and shall be required and specified is that the concatenation of all chunks must equal the original message. Moreover, testing the link twice with the same input message may result in different, yet correct chunks.

Quiescence

A transition expresses that an action, input or output, can be performed. In addition, it is also important to be explicit about the actions which cannot be performed. In particular, a state that cannot perform any output action must be distinguished from a state that can perform at a least one output action. A state that cannot perform any output or internal action cannot progress autonomously; it has to wait until someone provides it with an input before it can continue. Such a state is called *quiescent*.

An observer or tester looking at a system in a quiescent state does not see any output occurring. This observation of seeing nothing can itself be considered as an event. It is called *quiescence*, and it is denoted by the Greek letter δ (*delta*). Quiescence is an observation that can be made on a system, e.g., by setting a sufficiently large time-out value and observing that the system does not produce any output before expiration of the timer. Quiescence δ can be considered as just another 'output'-action, representing the absence of any 'real' output. It does not occur explicitly in a labeled transition system, but only implicitly by absence of 'real' outputs. Yet, quiescence actions can be composed with normal actions to express behaviour, for example, in traces.

Example 5. Coffee machine *s* in Figure 2 is quiescent in its initial state s_0. Machine *r* is always quiescent. After *but*? machine *w* may produce coffee *cof*!, but *w* may also be quiescent: *w* can perform the trace *but*?*cof*! but also the trace *but*?δ*but*?*tea*!δ: we write $w \xRightarrow{but?\,\delta\,but?\,tea!\,\delta}$. It expresses that after pushing the button there is no output, i.e., δ, is observed. If then the next stimulus *but*? is supplied *tea*! is observed, after which the machine is waiting again for the next input.

3.2 The Implementation Relation ioco

An implementation relation precisely defines when an SUT conforms to a specification model. The implementation relation **ioco**, abbreviated from input-output conformance, is such a relation for labeled transition systems. An SUT i is **ioco**-conforming to specification s if all outputs that i can produce, including quiescence, can also be produced by s in a comparable a state. In other words, i **ioco** s if after a sequence of actions of s, i never produces an output that is unexpected in s.

Example 6. Consider s in Figure 2 as the specification. It specifies that after pushing button *but*? either coffee or tea shall be produced. This is exactly what p does, so p **ioco** s. That p can produce *cof*! after *but*?*but*? is not important because *but*?*but*? is not a sequence of actions of s, so s does not specify anything for what shall happen after *but*?*but*?.

Also q **ioco** s because q does not produce any unexpected output: *cof*! is also an output of s after *but*?. In other words, q makes an implementation choice, viz. producing *cof*!, whereas s had left this choice open through the output-nondeterminism between *cof*! and *tea*!.

On the other hand, u **ioco** s, because u can produce *choc*! after *but*? which is not foreseen in s. Also r **ioco** s because r is quiescent after *but*? which is not expected in s: s always produces at least one of the outputs *cof*! or *tea*!. Also w can be quiescent after *but*?, so w **ioco** s.

The system v extends the functionality of s; v features producing *whiskey*! after being *hit*?. Yet, v **ioco** s, because s does not specify what shall happen after *hit*?. It is not a sequence of actions of s, and the SUT is free to do anything after *hit*?. We say that s is *underspecified* for input *hit*?.

A formal definition of **ioco** is given as follows. Let i and s be labeled transition systems with input actions in L_I and output actions in L_U, and let i be input-enabled. Then **ioco** is defined by

$$i \textbf{ ioco } s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in Straces(s) : \ out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

where

- The *traces* of s, i.e., sequences of actions, including quiescence δ that s can perform, are: $Straces(s) =_{\text{def}} \{ \sigma \in (L_I \cup L_U \cup \{\delta\})^* \mid \exists s : \ s \stackrel{\sigma}{\Longrightarrow} s' \}$
- A *generalized transition* $q \stackrel{\sigma}{\Longrightarrow} q'$ is a sequence of transitions from state q to state q', where $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$, i.e., σ is a, possibly empty, sequence of inputs, outputs, and quiescence. The sequence of transitions may contain internal transitions, but these are not visible in σ.
- A state q of i or s is *quiescent*, denoted by $\delta(q)$, if no output or internal transitions are possible from q: $\forall x \in L_U : \ q \stackrel{x}{\not\longrightarrow}$ and $q \not\rightarrow$
- The set of *reachable states* of s, with initial state s_0, is: $s \textbf{ after } \sigma =_{\text{def}} \{ s' \mid s_0 \stackrel{\sigma}{\Longrightarrow} s' \}$
- The set of possible *outputs*, possibly including quiescence δ, of state q is: $out(q) =_{\text{def}} \{ x \in L_U \mid \exists q' : q \stackrel{x}{\longrightarrow} q' \} \cup \{ \delta \mid \delta(q) \}$
- The set of possible *outputs* of all states of s reachable after trace σ is: $out(s \textbf{ after } \sigma) =_{\text{def}} \bigcup \{ out(q) \mid q \in s \textbf{ after } \sigma \}$

Nondeterminism is reflected in **ioco** by having more than one element in the set of possible outputs $out(s \textbf{ after } \sigma)$. A system i may implement any selection of these possible outputs as long as $out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$.

Underspecification occurs if for some inputs the specification does not prescribe at all which outputs an SUT may produce. This is reflected in **ioco** by requiring $out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$ only for traces $\sigma \in Straces(s)$. This means that for $\sigma \notin Straces(s)$ an SUT is free to perform any behaviour.

Example 7. In Figure 2, we have that $out(s \textbf{ after } but?) = \{cof!, tea!\}$, so s is output-nondeterministic.

We have p **ioco** s because $out(p \textbf{ after } but?) = out(s \textbf{ after } but?) = \{cof!, tea!\}$, and also q **ioco** s because $out(q \textbf{ after } but?) = \{cof!\} \subseteq \{cof!, tea!\}$.

But u **ioco** s because $choc! \in out(u \textbf{ after } but?)$ but $choc! \notin out(s \textbf{ after } but?)$. Also r **ioco** s because $out(r \textbf{ after } but?) = \{\delta\}$ but $\delta \notin out(s \textbf{ after } but?)$. For w, $out(w \textbf{ after } but?) = \{cof!, \delta\} \not\subseteq out(s \textbf{ after } but?)$, so that w **ioco** s.

Underspecification in s occurs for input $hit?$: $hit? \notin Straces(s)$, so v **ioco** s , although $out(v \textbf{ after } hit?) = \{whiskey!\} \not\subseteq out(s \textbf{ after } hit?) = \emptyset$.

Also $but?\,but?$ is underspecified in s. Thus, although $out(p \textbf{ after } but?\,but?) = \{cof!, tea!\} \not\subseteq \emptyset = out(s \textbf{ after } but?\,but?)$, we do have p **ioco** s.

3.3 Testing for Labeled Transition Systems

Testing involves the generation of test cases, the execution of these test cases on an SUT, and the analysis of the test outcomes. We first look at the structure of test cases for testing labeled transition systems, and how these test cases are executed. Then we show how these test cases can be automatically generated from a labeled transition system specification, and finally soundness and exhaustiveness are considered.

Test Cases

A test case specifies which inputs the tester will supply to the SUT, and which outputs it expects from the SUT, including the special output quiescence if no 'real' output occurs. Test cases for testing labeled transition systems are labeled transition systems themselves, but with reversed roles for inputs and outputs compared with the SUT. To assign a verdict, the terminating states of a test-case transition system are labeled **pass** or **fail**. If test execution ends in such a state the corresponding *verdict* is assigned: **pass** if all responses comply with the specification, and **fail** if an unexpected output occurs.

Even if test cases themselves are deterministic, possible nondeterministic behaviour of the SUT may lead to different outcomes when the same test case is repeated with the same SUT. An SUT passes a test case if all its test runs lead to a **pass** verdict; otherwise the SUT fails. An implementation passes a test suite, i.e., a set of test cases, if it passes all test cases in the test suite.

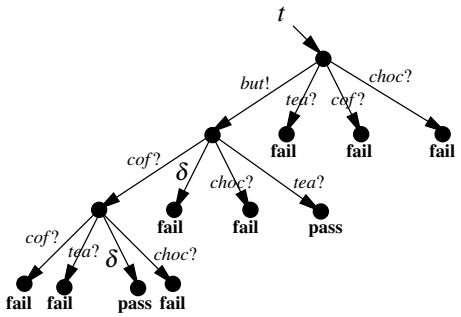


Fig. 3: Test case as a labeled transition system.

Example 8. In test case t shown in Figure 3, the tester initially presses *but*!. If the SUT produces any output before *but*! is pressed this leads to **fail**. After *but*! the tester expects some output from the SUT: *tea*? leads to the verdict **pass**, *choc*? leads to **fail**, and also the absence of output, i.e., the observation of δ, leads to **fail**. After *cof*? has been observed the tester does not expect any more real outputs: only quiescence then leads to **pass**.

If t is executed with p of Figure 2, because of output-nondeterminism in p, there are two possible test runs. Either the sequence *but*?*tea*! is observed leading to the verdict **pass**, or *but*?*cof*!δ is observed also leading to **pass**; we say that p **passes** t. On the other hand, applying t to u leads to a test run which fails: *but*?*choc*! ends in a test-case state labeled **fail**. Thus, we say that u **fails** t.

If t is executed with the nondeterministic SUT w we have two test runs: *but*?*cof*!δ leading to **pass**, and *but*?δ ending in **fail**, so w **fails** t.

Test Generation

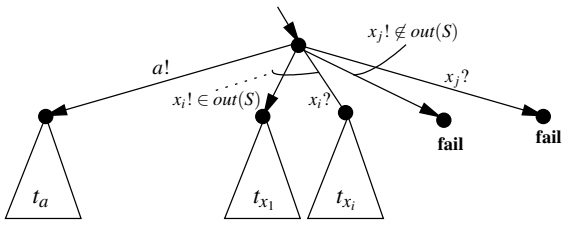
Now all ingredients are there to present an algorithm that generates test cases from a labeled transition system specification. The algorithm is recursive in the sense that the first transition of the test case is derived from the states in which the specification can be initially, after which the remaining part of the test case is recursively derived from the specification states reachable after the initial states via the first test-case transition. In each recursive step there are different choices leading to different test cases: (*1*) the test case can be terminated with the verdict **pass**; (*2*) the test case can continue with any input enabled in the specification; or (*3*) the test case can wait for an output from the SUT and check it, or conclude that the SUT is quiescent.

Algorithm 1. *Let s be a specification model, and let S be the set of states reachable via internal transitions from the initial state of s. A test case t is obtained from S as follows:*

1.

The single-state test case **pass** is always a valid test case.

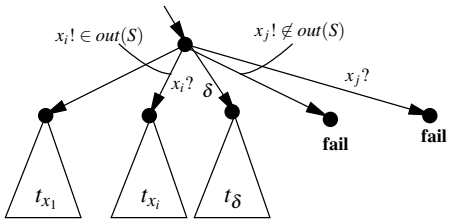
2.



*where $a?$ is an input (output of the test case) such that at least one transition with $a?$ is enabled in some state of S. The test case t_a is obtained by recursively applying the algorithm for the set of states S' reachable from S via $a?$: $S' = S$ **after** $a?$. Analogously, each t_{x_i} is obtained by recursively applying the algorithm for $S_{x_i} = S$ **after** x_i.*

This test case attempts to supply input $a?$ to the SUT, but if the SUT is faster and produces an output $x!$ before $a?$ could be supplied, then the tester checks whether this output is correct, $x_i! \in out(S)$, or not.

3.



This test case awaits and checks the next output $x_i!$ from the SUT. If $x_i!$ is correct, i.e., $x_i! \in out(S)$, then the algorithm continues recursively; otherwise the test fails. If no output arrives then quiescence δ has been observed, and the corresponding test-case transition is taken.

Example 9. Test case t in Figure 3 is obtained from s using Algorithm 1. From Example 8 we have that p **passes** t and this is consistent with p **ioco** s, which was shown in Example 6. We had u **io$\not$o** s and test case t can detect this: u **fails** t. From w **fails** t it can be concluded that w **io$\not$o** s.

Soundness and Exhaustiveness

Test generation Algorithm 1 can be shown to be correct. Firstly, all generated test cases are sound, i.e., they detect only non-**ioco** correct SUTs. Secondly, the collection of all different generated test cases is exhaustive, i.e., each non-**ioco** correct SUT is detected by a generated test case. But, as discussed in Section 2, such an exhaustive collection is infinite, or at least too large to be ever executed, so that test selection criteria are necessary to select from the exhaustive test suite. Test selection is not addressed in Algorithm 1, but most model-based test tools allow different strategies for it, the most straightforward one being random test selection. Other strategies include the manual specification of test purposes or test directives, and the use of structural coverage criteria like covering all states or all transitions of the labeled transition system specification.

4 Testing Real-Time Systems

The general concepts of model-based testing discussed in Section 2, and the more specific **ioco**-approach presented in Section 3, are also applicable to systems in which real-time plays an important role, such as embedded systems. By real-time we mean the system's ability to react timely to input events and produce responses (outputs) within specified (hard) deadlines, and the system's possibility to have different responses depending on the arrival time of input events. We introduce real-time in system modeling, in the implementation relation, in test generation, and in test test execution. Moreover, we introduce the real-time model-based test tool UPPAAL-TRON.

4.1 Environment and System Modeling

Adding time to a state-transition model can be accomplished by introducing clocks. Firing a transition and remaining in a state can be made dependent on constraints over clocks, and clocks can be reset in transitions. This is exactly the modeling formalism of Timed Automata that was introduced in Chapters **??** and **??**. Thus, we will use Timed Automata for describing models for timed model-based testing.

In addition to user interaction, an embedded system interacts closely with its environment which typically consists of the controlled physical equipment (the plant) accessible via sensors and actuators, other technical interfaces, or devices accessible via communication networks using dedicated protocols. In physical reality not all timed behaviour that can be modeled in Timed Automata can also occur. The switching time of a valve, the acceleration of an engine, or the time needed for manual operation of a light switch are constrained by physical laws. It makes no sense to generate test cases that contradict such environmental or physical constraints, be-

cause they can never be executed. To take such constraints into account, an explicit model of the environment is added to the system model. The model-based test generation tool can then restrict to generating test cases that make sense from a physical point of view.

Hence, we advocate an approach where both the required behaviour of the SUT and the assumed behaviour of its *environment* are modeled and used in test generation. This offers several advantages. First, the test generation tool can synthesize only relevant and realistic scenarios for the given type of environment, which, in turn, reduces the number of generated tests and improves the quality of the test suite. Second, the engineer can guide the test generator to specific situations of interest, even to specific test purposes. Third, a separate environment model offers a form of compositionality that eases system testing under different assumptions and use patterns.

On the one hand, the environment model shall only capture the behaviour that is interesting, and that can actually occur, in order not to generate too many or unrealistic test cases. On the other hand, it is important that all behaviour that can occur is included, i.e., the modeled environment behaviour shall be an over-approximation of the actual environment behaviour. Normally, environment models can be fairly simple. If no particular assumptions can be made or are desirable, a *universal environment* model, i.e., a model allowing all possible behaviour (see below), can always be used without loss of testing power, but probably at the expense of generating some uninteresting or infeasible test cases.

Cooling Controller Example

In this section we use a Cooling Controller as example, which is a simplified version of a real-life industrial refrigeration controller [5]. The objective of the Cooling Controller is to keep the temperature in a room within a given range by turning a compressor on and off. More specifically, the controller must (*1*) turn on the cooling device within an allowed reaction time when the room temperature becomes high; (*2*) turn it off when the temperature becomes low; and (*3*) sound an alarm when the temperature is high for a too long period.

The model for the Cooling Controller consists of 7 concurrent automata partitioned into environment and system components as shown in Figure 4. The components in the environment represent the *RoomTemperature* variations, the *Compressor*, an *AlarmHorn*, and the *User*. The SUT model consists of the *Compressor Controller*, the *Alarm Monitor*, and the *Adapter*. The room regularly informs the system about the current room temperature, modeled as three actions representing the major temperature ranges in the system: *High?*, *Med?*, and *Low?*. The controller reacts by switching the compressor using *CompressorOn!* or *CompressorOff!*. The alarm monitor notifies the user about a persistent high temperature via the *SoundAlarm!* action. The user may stop the alarm by pressing a *ClearAlarm?* key on the controller.

Figure 5 shows a timed automaton specification of the *Compressor Controller*. Initially, it is in the state *Off*, also called the *Off*-location. When it receives a

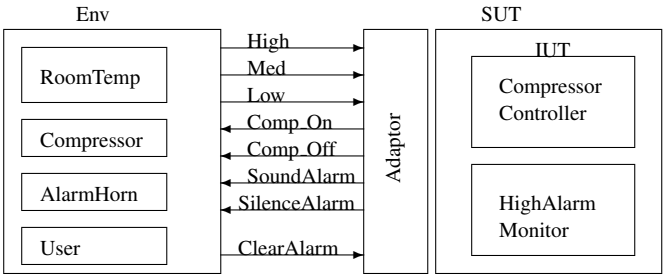


Fig. 4: System Diagram of Cooling Controller Model.

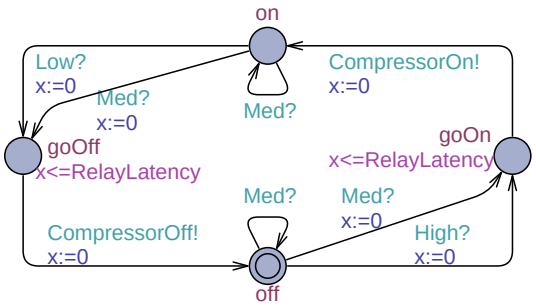


Fig. 5: Compressor Controller Model.

High? temperature indication it switches on the compressor relay within a reaction time of *RelayLatency*, which is controlled by clock x and the clock invariant `x<=RelayLatency` in location *goOn*. Dually, in the *On*-location it will transit towards the *goOff* location.

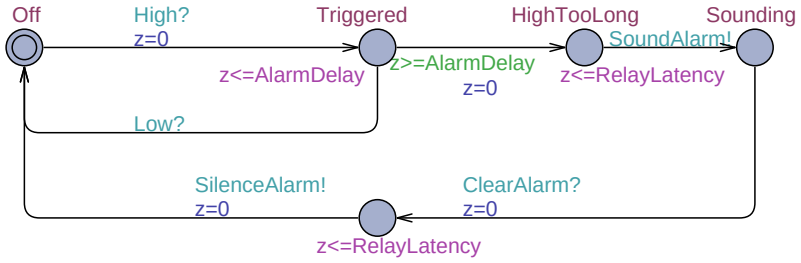
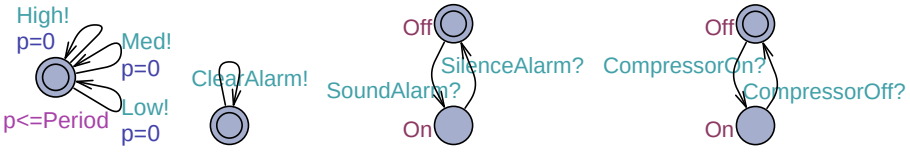


Fig. 6: Alarm Monitor Model.

Clock invariants are used to force the automaton to leave the *goOn* and *goOff* locations before the reaction time has elapsed, implying that the outputs, *CompressorOn!* and *CompressorOff!* respectively, are produced at some time before the required reaction time. The specification does not prescribe a precise moment when the output shall occur. An SUT may choose any moment as long as it is before the required reaction time. This is a case of timed output-nondeterminism; cf. Section 3.1. Similarly, when the room temperature is medium, the cooling is allowed to be either on or off, which is expressed by the nondeterministic response to the *Med?* input in locations *On* and *Off*. In both cases, it is the intention to give implementation freedom to the manufacturer concerning exact functionality, speed, timing tolerances, etc. This demonstrates an important feature of our testing framework, namely that models for testing may be kept loose and fairly abstract. It is not necessary to model a concrete implementation in all details, but only the requirements to be tested. The task of constructing models for testing is thus typically simpler than creating models for code generation where all details of the implementation must be contained in the model.

Figure 6 shows the *Alarm Monitor*. When it detects a *High?* temperature it moves from the *Off* location to the *Triggered* location, where it remains for *AlarmDelay* time units, controlled by the clock *z*, unless it observes a *Low?* temperature in the mean time, after which the high temperature situation is ignored. If the high temperature remains for *AlarmDelay* time units, the automaton moves to the *HighTooLong* location with an internal transition. Here it sounds the alarm, allowing for a small timing tolerance up to *AlarmSoundLatency*. It remains *Sounding* until the user clears the alarm by pressing the *ClearAlarm?* key.



(a) Temperature. (b) User. (c) Horn. (d) Compressor.

Fig. 7: Universal Environment Model for Cooling Controller

Figure 7 shows the components of the environment model. Figure 7a shows the universal, completely unconstrained environment allowing any (timed) sequence of temperature variations, including changing directly from high to low (and vice versa) without passing through the medium range, and with any speed. Similarly, the user may clear the alarm at any time and speed, whether sounding or not. The remaining components just track the on/off states of the horn and the compressor, respectively. The temperature variations are not realistic, as the physical temperature evolves slowly and continuously. Hence, one may prefer an environment model with more restricted, realistic behaviour. Figure 8 provides an alternative environment where the temperature always changes through the medium range and with a

speed bounded by *MinTempDelay*. Thus our framework enables the test engineer to use different environment models of varying strengths. Taken further, this can be used to focus testing on restricted, important scenarios, or on specific test purposes.

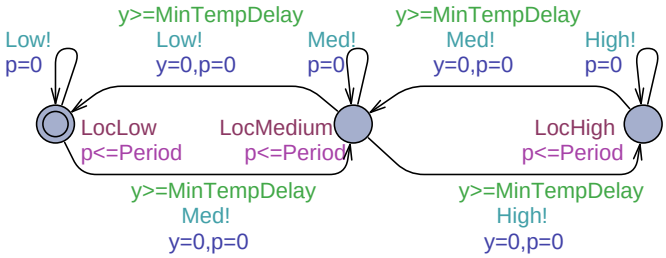


Fig. 8: Alternative, more realistic Environment Model

4.2 A Timed Implementation Relation

The implementation relation **ioco** uses labeled transition systems with inputs and outputs. In Timed Automata, in addition to inputs and outputs, there is also time that is observable: a tester can observe the passing of time, or can decide not to provide an input for a certain period of time. This means that time passing, with specific durations, occurs as an action in traces and observations. An issue with time, from a testing perspective, is that it is neither input nor output, but a bit of both: a system and its environment synchronize on time, and both can decide to let time pass, or to perform an action first. Time might be called 'semi-controllable' and 'semi-observable' by the system as well as its environment.

Quiescence was introduced as the absence of outputs, now and in the (unbounded) future. Once we have time in our tests, absence of outputs is always for a particular period of time, which is just observing passing of time without an action occurring. This means that in timed testing the concept of quiescence is no longer necessary, nor possible.

Adding timed actions, timed observations, removing quiescence, and introducing an explicit environment model leads to the implementation relation $\mathbf{rtioco}_e$, the e-relativized timed input/output conformance relation:

$$i\ \mathbf{rtioco}_e\ s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in \mathit{TimedTraces}(e) :\ \mathit{out}(\,(i,e)\,\mathbf{after}\,\sigma\,) \subseteq \mathit{out}(\,(s,e)\,\mathbf{after}\,\sigma\,)$$

Here, (i,e) denotes the combined state of the implementation i and the environment e, and dually, (s,e) is the combined state of the specification s and e. An SUT i conforms to a model s in an environment e according to $\mathbf{rtioco}_e$ if for all possible timed traces σ of the environment e, the observations of i in context e after σ are

included in the observations of s in context e after σ. A timed trace is a sequence of inputs, outputs, and explicit time durations. An observation in *out* is an output that can occur now, or it is the passing of some time duration. Inclusion of observations means that (*1*) i may only produce outputs which are at this particular moment allowed by s, i.e., i is not allowed to produce an output at a time when that is not allowed by s; and (*2*) i may let time pass only if s can let time pass, i.e., it is not allowed to omit producing an output when an output is required by s.

Timed testing now consists of selecting and executing timed inputs and proposed delays from the model, applying them to the SUT, observing the resulting outputs and SUT delays, and checking with the specification model that these observations are allowed.

Example 10. Consider the following alternating sequence of inputs and delays from the automaton in Figure 5, where delays are represented by numbers indicating the amount of time that passes between two events: 2 Med? 150 High? 100. Such a trace may serve as *test-input sequence*: inputs are supplied by the tester to the SUT, analogous to the untimed case in Section 3.3, and time delays are periods in which the tester does nothing and waits for outputs from the SUT. Then it can happen that the SUT produces an output *CompressorOn!* during the last delay period after 5 time units. The resulting observed input/output/delay trace is 2 Med? 150 High? 5 CompressorOn! 95.

If *RelayLatency* equals 20 this observation is allowed according to the model in Figure 5, since $5 \in out((s,e)\ \textbf{after}\ 2\,Med?\,150\,High?) = \{0..20, CompressorOn!\}$.

If, however, the trace 2 Med? 150 High? 25 CompressorOn! is observed, it would not be conforming, because the model does not allow that 25 time units elapse before the compressor is switched on: $25 \notin \{0..20, CompressorOn!\}$.

4.3 The SUT *and System Adaptation*

To execute generated tests, the events of the model must me translated into concrete physical stimuli on the interface of the SUT, and vice versa for SUT outputs. In model-based testing, this was called *adaptation*, and the component implementing it is called the *adapter*; cf. Section 2.

For demonstration purposes we have created a Cooling Controller SUT as a Java application. The interface consists of a number of methods for injecting a new temperature reading, clearing alarm, and setting alarm and compressor status. The adaptation consists of a TCP connection and a small piece of interface code; see Figure 9. This translates the abstract model input events *Low?*, *Med?*, and *High?* represented as messages into direct API calls of `handleNewTemp(x)` with concrete representative temperatures x. The reverse, i.e., mapping actual outputs into model events is not shown due to brevity.

The communication between our model-based test tool UPPAAL-TRON and the SUT takes place via a TCP socket connection. This implies that the communication

is bidirectional and buffered. Events may travel concurrently in both directions, and consequently such concurrent input and output events may appear to be re-ordered. Although this situation may appear exotic, it does happen in practice during intensive on-line tests, and it may lead to incorrect verdicts. Moreover, there are communication delays that the tester must compensate for to be able to give a correct verdict when performing real-time testing. Thus, the SUT that the tester actually sees is the combined behaviour of the Cooling Controller SUT *composed with* the socket connection (or adapter in general). The way to deal with this is to provide an abstract model of the adapter behaviour, too, so that tests are generated for the composition of original SUT and adapter, i.e., the SUT that the tester actually sees.

```
1   while (true) {
      lock.lock(); // lock operations on input buffer
3     while (inputBuffer.isEmpty())
          cond.await();
5     msg = inputBuffer.poll().intValue();
      lock.unlock();// allow buffer to be filled again
7     if (msg == inputTempHigh) {
          controller.handleNewTemp(10);
9     } else if (msg == inputTempMed) {
          controller.handleNewTemp(5);
11    }
       else if (msg == inputTempLow) {
13        controller.handleNewTemp(0);
      }
15    else if (msg == inputClearAlarm) {
          controller.handleClearAlarm();
17    } else {
          System.err.println("IOHandler: UNKNOWN");
19    }
    }
```

Fig. 9: Fragment of the adaptor code.

4.4 Timed Test Generation and Execution

Execution of timed test cases is more difficult than untimed test execution. Timed test execution requires synchronization between the SUT and the tester. They must have the same vision on the progress of time. Moreover, the test execution engine itself is a (soft) real-time program that must execute fast enough to keep pace with real-time, if test cases are executed following real, physical time. Alternatively, it is sometimes possible to execute tests in simulated time, i.e., time is simulated by the increasing value of a variable, yet, also in this case synchronization is important. Chapter **??** will discuss a case of real-time testing in simulated time.

UPPAAL-TRON

UPPAAL-TRON [4, 5] is a model-based test generation and execution tool that supports both real-time and simulated time testing. It executes test events *on-line*, i.e., the events are immediately executed when they are generated; see also Section 5. Input events are generated by performing a guided random exploration of the environment model while outputs are checked against the specification model, where the e-relativized timed input/output conformance relation $\mathbf{rtioco}_e$ outlined in Section 4.2 is used to judge the SUT behaviour.

UPPAAL-TRON is integrated with the UPPAAL model-checker, and uses the same Timed Automata syntax and semantics as presented in Chapters **??** and **??**, the only difference being that it must be possible to partition the model into an environment and a system part. It is thus possible to simulate and verify properties of the model prior to testing, to ensure that the right behaviour is specified and tested.

The algorithm behind UPPAAL-TRON operates in a similar fashion as the one for labeled transition systems in Algorithm 1. It continually computes the possible set of states S that the (combined system and environment) model can occupy. It uses this set to compute the possible set of inputs that the tester can offer, and the set of allowed system outputs. The main loop runs until the testing time is over, or non-conformance is detected. It randomly chooses between offering an input to the SUT, observing the SUT for outputs, and delay to let time pass. If an output occurs, it checks if this is allowed in the current state-set; else it declares non-conformance. After each input, output, or delay action the set of states S is updated. Whereas in Algorithm 1 this updating of S is relatively easy because S is a finite set of states, UPPAAL-TRON needs more sophisticated data structures and algorithms to deal with infinite intervals of real-time. It uses and extends the compact symbolic data structures and state-exploration algorithms implemented in the UPPAAL engine to efficiently compute, update, and manipulate S.

The output of UPPAAL-TRON executed against an erroneous implementation of the Cooling Controller, where the deadline for switching on the alarm is exceeded, is shown in Figure 10. The verdict is on line 8 together with an explanation of the cause. Additional diagnostic and resource usage information are also provided.

```
    Options for input   : High()@(1147;1171), ClearAlarm()@(1147;1171)
2   Options for output  : SoundAlarm()@(1165;1171)
    Options for internal: c_ClearAlarm@(1147;1148), -@(1165;1166)
4   Options for delay   : until 1247)
    Last time-window    : (1246;1247)
6   Max. system delay   : until 1171)
  Last time-window is beyond maximum allowed delay.
8 TEST FAILED: IUT failed to produce output in time.
  Time elapsed: 1246 tu = 12.460001s
10 Time     left: 998754 tu = 9987.539999s
  Random  seed: 1297759459
```

Fig. 10: Sample output of UPPAAL-TRON when a test fails.

5 Concluding Remarks

In this chapter we have shown the concepts and principles of model-based testing, elaborated for labeled transition systems and for Timed Automata. In this section we conclude with automation and tools, other approaches, benefits, issues, and perspectives of model-based testing.

Automation and Tools

Although model-based testing can be applied manually, the main benefits are obtained when a model-based test generation tool is used, and when the generated test cases are also executed automatically. There are two ways of combining automatic test generation with test execution. The straightforward way is to generate test cases from a model, store them, and subsequently execute them on the SUT. This is called *off-line*, or *batch* model-based testing. The second way consist of weaving test generation into test execution, and is called *on-line*, or *on-the-fly* model-based testing. The first action of the test case is generated and immediately executed, and responses from the SUT are on-line compared with the model without explicitly generating a test case. On-line model-based testing has the advantages that no complete test cases need be generated and stored before test execution can start, information gathered during test execution can be used in the remaining part of test generation, and for state-based models there is no need to first generate a complete state space so that also infinite-state models can be used. Disadvantages are that there is no explicit test case that can be manually inspected, for repeatability test cases must be (automatically) regenerated, and since the complete state space is not generated, some coverage criteria, like state coverage, are not directly applicable.

Since model-based test generation tools generate so called abstract test cases, i.e., test cases on the same abstraction level as the model, test execution involves implementing these abstract test cases. This includes mapping the abstract input actions of the test case onto the concrete interfaces of the SUT, interpreting the concrete outputs from the SUT in terms of the abstract actions of the test case, and (physically) connecting the (model-based) test execution tool to the SUT. This was called *adaptation* in Section 2, and it is performed by an *adapter*. Adapter development can be laborious, in particular for embedded systems where SUT interfaces are not always easily accessible. Moreover, it is mostly SUT specific and difficult to automate. Yet, adapter development is not specific for model-based testing, but occurs in any kind of automated test execution.

Since model-based testing is currently a vibrant area with new approaches and tools appearing every month, both commercial and academic, we will not try to give an overview of existing tools. Tools that implement the **ioco**-approach are, among others, TORX [9], JTORX [1], and TORXAKIS [6]. JTORX will be used in Chapter **??** to test a software bus. UPPAAL-TRON [4] was used in Section 4 for real-time testing following the $\mathbf{rtioco}_e$ implementation relation; it will also be used in Chap-

ter **??** for testing a wireless-sensor-network node. These four tools follow the on-line approach: test cases are executed while they are generated.

Other Approaches

As stated in Section 1, this chapter considered specification-based, black-box testing of functionality, and we did that with state-based models that specify both inputs and expected outputs. Other approaches to model-based testing exist, and some of them are mentioned here.

Firstly, there are many other notations and languages to model the behaviour of systems: different varieties of finite, infinite, and extended state machines, state charts, functional languages, algebraic and process-algebraic languages, guarded commands, pre/post-conditions, (temporal) logic, formal grammars, UML, OCL, etc. For many of these notations model-based testing approaches have been investigated and developed.

Secondly, monitoring approaches, also called passive testing, only observe the behaviour of the SUT. Consequently, such models contain only outputs. The triggers or inputs must be supplied separately, usually by regular user behaviour.

Thirdly, models with only inputs need a separate *oracle* to judge the correctness of outputs. Such models are used in, e.g., performance-, load-, or operational-profile testing. In the latter the input model is derived from usage scenarios specifying how typical users use the system.

Benefits

Model-based testing makes it possible to automatically generate large numbers of long tests. This allows testing that is more thorough, faster, cheaper, and more efficient than manual testing. Moreover, it is easily repeated after modifications in the model or in the system: instead of manually analysing an existing test suite for regression and confirmation tests, a new test suite is automatically generated from an adapted model.

Model-based testing requires the development of a model. Although this may seem an extra effort, practice shows that this modeling activity in itself leads to improved understanding of the system, and to earlier detection of imprecise, incomplete, or ambiguous requirements. Moreover, the availability of models makes it possible to apply other model-based analysis techniques, such as simulation and model checking.

Validity of test cases, i.e., do test cases really test what they should test (soundness and exhaustiveness), is a cumbersome issue in manual testing, even more so when the system or its specification evolves. In model-based testing validity is precisely defined and established, and a test-generation algorithm can be proved to produce only valid test cases. The model-based testing practitioner, however, need

not be aware of such proofs, since they are established once by specialists and implemented in tools, so that the practitioner can concentrate on the SUT and its model.

Finally, the model-based testing approaches described in this chapter can deal with typical embedded software aspects such as real-time properties, concurrency, nondeterminism, abstraction and implementation choices, underspecified and partial specifications, and assumptions on the (physical) environment,

Issues

A straightforward model-based testing process consists of (*1*) constructing a model; (*2*) choosing and developing a tool environment including model-based test generation, test execution, and an adapter; and (*3*) generating and executing test cases, either on-line or off-line. Such a process is not always directly applicable, in particular, the construction of a valid model can be difficult and troublesome, if specification and requirement documents are incomplete, ambiguous, imprecise, or obsolete. Model-based testing can then be used in a less sequential and more cyclic and 'agile' process, where the system and the model are concurrently developed, and continuously compared using model-based testing techniques; Chapter **??** will illustrate this.

One of the open issues in model-based testing is a well-founded, quantified notion of test selection; see Section 2. Test selection involves optimizing test coverage, while minimizing test effort. Although the use of models, in principle, provides a firm basis for quantifying coverage, most model-based test tools use more ad-hoc approaches usually inspired by traditional software testing. The question how to define, then measure, and eventually control the coverage of an automatically generated test suite, is challenging. Even more challenging is the question how to relate such a test coverage measure to a measure of quality of tested products. After all, product quality is the ultimate reason to do testing.

Perspectives

Model-based testing currently attracts a lot of attention. Research institutes investigate it, companies are trying it, conferences and workshops are being organized, scientific journals as well as professional and more popular magazines publish about it, and tool vendors emerge [10, 3]. We expect that model-based testing has the potential to make the testing process more efficient and more effective, if used with care and supported by the right tools, and that the extra effort of making models is more than compensated by earlier detection of errors, and by cheaper, faster, and better testing, and in the end higher quality products. The next chapters **??** and **??** describe the application of model-based testing to a wireless-sensor-network node and to the server component of a software bus.

References

1. Belinfante, A.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: Esparza, J., et al. (eds.) TACAS 2010. LNCS 6015, pp. 266–270. Springer (2010)
2. Dijkstra, E.: Notes On Structured Programming (1969)
3. Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.: Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology. Software Testing, Verification and Reliability 21(1), 55–71 (2011)
4. Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing Real-Time Systems Using UPPAAL. In: Hierons, R., et al. (eds.) Formal Methods and Testing. LNCS 4949, pp. 77–117. Springer-Verlag (2008)
5. Larsen, K., Mikucionis, M., Nielsen, B., Skou, A.: Testing Real-Time Embedded Software using UPPAAL-TRON: An Industrial Case Study. In: Wolf, W. (ed.) EMSOFT 2005. pp. 299–306. ACM (2005)
6. Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., Wichers Schreur, R.: Model-Based Testing of Electronic Passports. In: Alpuente, M., et al. (eds.) FMICS 2009. LNCS 5825, pp. 207–209. Springer-Verlag (2009)
7. Myers, G.: The Art of Software Testing. John Wiley & Sons Inc. (1979)
8. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R., et al. (eds.) Formal Methods and Testing. LNCS 4949, pp. 1–38. Springer-Verlag (2008)
9. Tretmans, J., Brinksma, E.: TORX : Automated Model Based Testing. In: Hartman, A., et al. (eds.) First European Conference on Model-Driven Software Engineering. Imbuss, Möhrendorf, Germany (2003)
10. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann (2007)

Experiences with Formal Engineering: Model-based Specification, Implementation and Testing of a Software Bus at Neopost

Marten Sijtema and Mariëlle Stoelinga and Axel Belinfante and Lawrence Marinelli

Abstract We report on the actual industrial use of formal methods during the development a software bus. At Neopost Inc, we developed the server component of a software bus, called the *XBus*, using formal methods during the design, validation and testing phase: We modeled our design of the XBus in the process algebra mCLR2, validated the design using the mCRL2-simulator, and fully automatically tested our implementation with the model-based test tool JTorX. This resulted in a well-tested software bus with a maintainable architecture. Writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time. Moreover, the errors found with model-based testing would have been hard to find using conventional test methods. Thus, we show that formal engineering can be feasible, beneficial and cost-effective.

1 Introduction

Formal engineering, that is, the use of formal methods during the design, implementation and testing of software systems is gaining momentum. Various large companies use formal methods as a part of their development cycle; and several papers report on the use of Formal Methods during ad hoc projects.

Marten Sijtema
University of Twente, Address of Institute, e-mail: marten@systematic.nl

Mariëlle Stoelinga
University of Twente, Address of Institute e-mail: marielle@cs.utwente.nl

Axel Belinfante
University of Twente, Address of Institute e-mail: axel.belinfante@cs.utwente.nl

Lawrence Marinelli
Neopost, Address of Institute e-mail: cwl.martinelle@neopost.com

Formal methods include a rich palette of mathematically rigorous modeling, analysis and testing techniques, including formal specification, model checking, theorem proving, extended static checking, run-time verification, model-based testing, and much more. The central claim made by the field of Formal Methods is that, while it requires an initial investment to develop rigorous models and perform rigorous analysis methods, these pay off in the long run in terms of better, and more maintainable code. While experiences with formal engineering have been a success in large and safety-critical projects, we investigate this claim for a more modest and non-safety critical project, namely the development of a software bus.

Developing the XBus. In this paper, we report on our experiences with formal methods during the development of the XBus at Neopost Inc. Neopost is one of the largest companies producing supplies and services for the mailing and shipping industry, and the XBus is a software bus that supports communication between mailing devices and software clients. We have developed the XBus using the classical V-model, using formal methods during the design and testing phase.

An important step in the design phase was the creation of a behavioral model of the XBus, written in the process algebra mCRL2 [13, 2]. This model pins down the behavior of the XBus in a mathematically precise way, and this precision greatly increased the understanding of the XBus protocol, which made the implementation phase a lot easier.

After implementing the protocol, we tested the implementation against the mCRL2 model, using JTorX. JTorX [6, 1] is a model-based testing tool (partly) developed during the Quasimodo project [4], and is capable of automatic test generation, execution and evaluation. During the design phase, we already catered for model-based testing, and designed for testability: we took care that at the model boundaries, we could observe meaningful messages. Moreover, we made sure that the boundaries of the MCRL2 model matched the boundaries in the architecture. Also, to use model-driven test technology required us to write an adapter. This is a piece of software that translates the protocol messages from the MCRL2 model into physical messages in the implementation. Again, our design for testability greatly facilitated the development of the adapter.

Our findings. We ran JTorX against the implementation and the MCRL2 model (which is completely automatic) and found five subtle bugs that were not discovered using unit testing, since these involved the order in which protocol messages should occur. After repairing these, we ran JTorX several times for more than 24 hours, without finding any errors.

Since writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time, we conclude that the formal engineering approach has been very successful: with limited overhead, we have created a reliable software bus with a maintainable architecture. Therefore, we clearly show that formal engineering is not only beneficial for large, complex and/or safety-critical systems, but also for more modest projects.

Remainder of this chapter. The remainder of this chapter is organised as follows. Section 2 provides the context of this project. Then, Section 3 describes the course of this project, by discussing the activities involved in each phase of the development of the XBus. Section 4 reflects on the lessons learnt in this project and finally, we present conclusion in Section 5.

2 Background

2.1 The XBus and its context

Neopost, Inc. Neopost Incorporated [3] is one of the main manufacturers of equipment and supplies for the mailing industry. Neopost produces both physical machines, like franking machines and mail inserters, as well as software to control these machines. Neopost is a multinational company headquartered in Paris, France and has departments all over the world. Its software division, called Neopost Software & Integrated solutions (NSIS) is located in Austin, Texas, USA. This is where our project took place.

Shipping and Franking mail. Typically, the workflow of shipping and franking is as follows. To send a batch of mail, one first puts the mail into a folding machine, which folds all letters, then an inserter inserts all letters into envelopes[1] and finally, the mail goes into a franking machine, which puts appropriate postage on the envelopes, and keeps track of the expenses.

Thus, to ship a batch of mail, one has to set up this process, selecting which folder, inserter and franking machine to use and configure each of these machines, setting the mail's size, weight, priority, and the carrier to use. These configurations can be set manually, using the machine's built-in displays and buttons. More conveniently, however, is to configure the mailing process via one of the desktop applications Neopost provides.

The XBus. To connect a desktop application to the various machines, a software bus, called the XBus, has been developed. The XBus communicates over TCP and allows clients to discover other clients, advertise provided services, query for services provided by other clients and subscribe to services. Also XBus clients can send self-defined messages across the bus. An XBus connecting two desktop applications, a log generator and a franking machine is shown in Figure 1.

When this project started, an older version of the XBus existed, called the XBus version 1.0. Goal of our project was to re-implement the XBus while maintaining backward compatibility, i.e. the XBus 2.0 must support XBus 1.0 clients. Key requirements for the new XBus are improved maintainability and testability.

[1] Alternatively, a combined folder/inserting system can be used

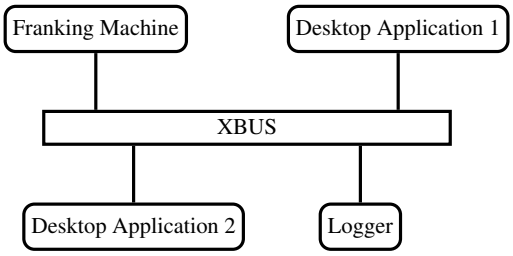


Fig. 1 Desktop applications, logger and a franking machine are connected to the central XBus.

2.2 Model-based testing

Model-based testing (MBT, a.k.a. model-driven testing) is an innovative testing methodology that provides methods for automatic test generation, execution and evaluation. Model-based testing requires a formal model *m*, usually a transition system of the system-under-test (SUT). This model *m* pins down exactly which system behaviors are correct and which are not.

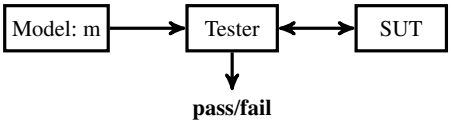


Fig. 2 Model-based testing.

The concept of model-based testing is visualized in Fig. 2: Tests are derived from *m* are applied to the system-under-test. Based on observations made during test executions, a verdict (pass or fail) about the correctness of the system implementation is given.

Each test case consists of a number of test steps. Each test step either applies a stimulus (i.e. an input to the SUT), or obtains an observation (i.e. a response from the SUT). In the latter case, we check whether the response was expected, that is, if it was predicted by the model *m*. If case of an unexpected observation, the test case ends with verdict **fail**. Otherwise, the test case may either continue with a next test step, or it may end with a verdict **pass**.

These techniques have been implemented in the model-based test tool JTorX. JTorX [6, 1] was developed during the Quasimodo project, and is a re-implementation of its predecessor TorX [5, 16]. TorX was one of the first model-based testing tools in the field, and JTorX greatly improves and enhances TorX with more efficiency, better user interface and more functionality.

In JTorX the test derivation and test execution functionalities are tightly coupled: test cases and test steps are derived on demand (lazily) during test execution. This is why we do not explicitly show the test cases in Fig. 2.

MBT provides a rigorous underpinning of the test process: assuming that the model correctly reflects the desired system behavior, all test cases derived from the model can be shown correct, i.e., they yield the correct verdict when executed against any implementation. Rich and well-developed MBT theories exist for control-dominated applications, and have been extended to test real-time properties [10, 14, 9], data-intensive systems [11], object-oriented systems [12], and systems with measure imprecisions [8]. Several of these extensions have been developed during the Quasimodo-project as well.

2.3 MCRL2

mCRL2 [13, 2] is a formal modeling language for describing concurrent systems developed by the Eindhoven University of Technology. It is based on the process algebra ACP [7], and extends ACP with rich data types extended and higher-order functions. The mCRL2-toolset facilitates simulation, analysis and visualization of behavior; model-based testing against mCRL2 models is supported by the model-based test tool JTorX. mCRL2 specifications start with a definition of the required data types. The behavior of the system is declared via process equations of the form $X(x_1 : D_1, x_2 : D_2, x_n : D_n) = t$, where x_i is a variable of type D_i and t is a process term. Process terms are built from potentially parameterized actions and the operators alternative composition, sum, sequential composition, conditional choice (if-then-else), parallel composition, and encapsulation, renaming, and abstraction. Actions represent basic events (like sending a message or printing a file) which are used for synchronization between parallel processes. mCRL specifications can be model checked via the CADP model checker, by generating the state space in `.aut` format, they can be proven correct using eg the theorem prover PVS, and they can be tested against with JTorX.

3 Development of the XBus

We developed the XBus implementation using the classical V-model [15], see Fig. 3. Below, we describe the activities we carried out in each phase of the V-model. Each section below corresponds to an activity in the V-model.

In our approach we have three testing phases: unit testing, integration testing and acceptance testing. Below we only discuss the first two of these; acceptance testing was done in the ususal way.

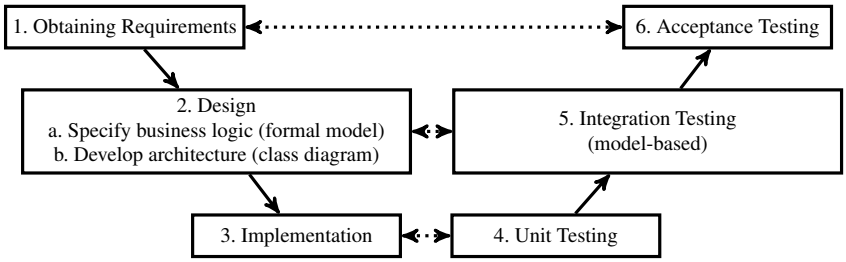


Fig. 3 The V-model that was used for development of XBus.

3.1 XBus Requirements

We have obtained the functional and nonfunctional requirements by studying the documentation from the XBus version 1.0 (a four page text document) and by interviewing the manager of the XBus development team.

The functional requirements express that the XBus is a centralized software application which can be regarded as a network router. Clients can discover other clients, advertise services, and query for services that are provided by other clients. Also, they can subscribe to services, and send self-defined messages to each other. Below, we summarize the functional requirements. As said before, important nonfunctional requirements are testability, maintainability and backwards compatibility with the XBus 1.0.

Functional requirements The functional requirements are as follows.

1. XBus messages are formatted in XML, following the same Schema as the XBus 1.0.
2. Clients connecting to XBus perform a handshake with the XBus server. The handshake consist of a $\mathsf{Conn}_{\mathsf{req}}$— $\mathsf{Conn}_{\mathsf{ack}}$— $\mathsf{Conn}_{\mathsf{auth}}$ sequence.
3. Newly connected clients are assigned unique id's.
4. Clients can subscribe to be notified when a client connects or disconnects.
5. Clients can send messages to other clients with self-defined, custom, data.
6. Clients can subscribe to receive messages of types they request in a Sub message.
7. Clients are able to advertise services they provide, using the $\mathsf{Serv}_{\mathsf{ann}}$ message.
8. Clients can inquire about services, by specifying a list of service names in a $\mathsf{Serv}_{\mathsf{inq}}$ message. Service providers that provide a subset of the inquired services will respond to this client with the $\mathsf{Serv}_{\mathsf{rsp}}$ message.
9. Clients can send *private* messages, which are only delivered to a specified destination.
10. Clients can send *local* messages, are delivered to the specified address, as well as to clients subscribed to the specified message type.

Protocol messages. XBus protocol messages are the following.

$\mathsf{Conn}_{\mathsf{req}}$ (implicit) implied by a client establishing a TCP connection with XBus

$\mathsf{Conn}_{\mathsf{ack}}$	sent from XBus to a client just after the client establishes a TCP connection with the XBus, as part of the handshake.
$\mathsf{Conn}_{\mathsf{auth}}$	sent from a client to the XBus to complete the handshake.
$(\mathsf{Un})\mathsf{Sub}$	sent from client to XBus to (un)subscribe a set of message types.
$\mathsf{Notif}_{\mathsf{conn}}$	sent from XBus to clients that subscribed connect notifications.
$\mathsf{Notif}_{\mathsf{disc}}$	sent from XBus to clients that subscribed disconnect notifications.
$\mathsf{Serv}_{\mathsf{ann}}$	sent (just after connecting) from a client c to XBus, which broadcasts it to all other connected clients, to advertise the services provided by c.
$\mathsf{Serv}_{\mathsf{inq}}$	sent (just after connecting) from client to XBus, which broadcasts it to all other connected clients, to ask what services they provide.
$\mathsf{Serv}_{\mathsf{rsp}}$	sent from a client via XBus to another client, as response to $\mathsf{Serv}_{\mathsf{inq}}$, to tell the inquirer what service the responding client provides

3.2 XBus Design

The design phase encompassed two activities: first, we made an architectural design, given by the UML class diagram in Figure 4. Then, we created a formal model in mCRL2, describing the protocol behavior.

Architectural Design. The architecture of the XBus is given in Figure 4, following a standard client-server architecture. Thus, the XBus has a client side, implemented by the XBusGenericClient, and a server side, implemented by the XBusManager. The latter handles incoming protocol messages and sends the required responses. Both the server and the client use the communications package, which implements communication over TCP.

We have catered for model-based testing already in the design: the XBusManager has a subclass JTorXTestableXBusManager. As we elaborate in Section 3.5, the JTorXTestableXBusManager class overrides the sendmessage from the XBusManager, allowing JTorX to have more control over the state of the XBus server.

Modeling strategy When creating the model, the first step is to define *what* and *what not* to model, to determine the abstraction level and boundaries of the model.

Included in the model The messages that come into the server, their handling and their response messages are modeled, as follows. After a message is received, the server will handle it. This means that the server will send a response, relay the message, broadcast a message, and/or modify internal state, depending on the type of message that arrived. Furthermore, the server keeps track of the client's state by keeping an internal list of client objects, just as in the Engine package in the architecture.

Excluded from the model In the ConnectionManager class a queue data structure is used as a buffer for incoming messages. This queue is *not* included in the model. The moment a message is popped from the queue it is considered received and thus

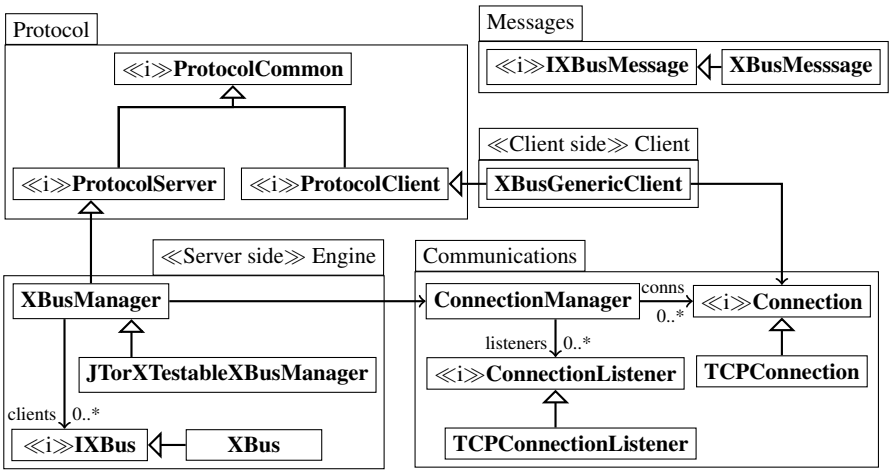


Fig. 4 High level architecture of the XBus system. It contains a server side package, and a client side package. Furthermore, it has functionality for TCP connections and XBus messages. Both server and client implement the Protocol interfaces. All interfaces are indicated with ≪i≫.

'in the model', and as soon as a message is sent to a client it leaves the model. So the queue is just on the outside of the boundary of what is model and what is not. This is reflected in the architecture in the Engine package. Here, the queue is also at the outside of the boundary of the Engine package.

Thus, we do *not* model internal components like TCP-sockets, queues, programmatic events, etc. The correctness of these internal components will be verified by unit-tests. We will use the model discussed here to simulate and test the XBus protocol (i.e. the business logic).

The XBus model We modeled the desired behavior of the XBus as an mCRL2 process. We chose for mCRL2 because of its powerful data types, which makes the modeling of the messages and its parameters convenient.

Behaviour. The behaviour of the XBus server is modeled as a single process that – for all kinds of incoming messages that it may receive – accepts a message, processes it (which may involve an update of its state), and sends a response (where appropriate), after which it is ready to accept the next message.

The language mCRL2 allows modeling of systems with multiple, parallel, processes, but this is not needed here. Having multiple, concurrent processes would make the system as well as the model more complicated, which would make them harder to maintain and test. One might choose to use multiple processes when performance is expected to be a problem, but that is not an issue here. In a large mailing room there may be 20 clients at the same time, a number with which the single-process server can easily cope.

Data. All the data that the server keeps track of is kept in one data object: a list of clients. This is modeled as a list of data structures, that for each client contain the following items:

- an integer that represents the identity of the client;
- the connection status of the client, which is an enumeration of: disconnected, awaitingAuthentication, connected;
- the subscriptions of the client, which is a list of enumerations: connectNotification, disconnectNotification, applicationNotification;
- the services that the client provides, which is a list of integers.

The client objects are manipulated by functions. These functions can be called from the server process, and can modify data structures defined in the data part.

Model size The entire model consists of 6 pages of text, including comments. Approximately half of it concern the specification of data types and functions over them; the other half is the behavioural specification.

Model validation During the construction of the model, we exhaustively used the simulator from the mCRL2 toolkit. This was done for two reasons. First, to get a better understanding of the working of the whole system, and to validate the design—did we design the system correctly?— already before the implementation phase is started. This was particularly useful to improve the understanding of the XBus protocol, of which only a (non-formal) English text description was available, which contained several ambiguities. Second, to validate the model, to be able to use it with the confidence that it faithfully represents the design during the model-based testing during integration testing. Due to time constraints model-checking was not done. It would have allowed validation of (basic) properties like the absence of deadlocks, as well as checking whether the model satisfies the functional requirements.

3.3 Implementation

Once we had sufficient confidence in the quality of the design—to a large extent due to modeling and simulation— it was implemented. The programming language used was C#.NET—use of .NET is company policy.

3.4 Unit Testing

As mentioned in the description of the approach we intended to use model-based testing for the business logic, i.e. to test the interaction between XBus and its clients. We chose to focus with unit-testing on those classes of XBus that are unrelated to the business logic: the classes of the Communications package and the Messages package, to cover as much of the XBus implementation as possible,

For the Communications package unit tests were written to test the ability to start a TCP listener and to connect to a TCP listener, to test the administration of connections, and to test transfer of data.

For the Messages package unit tests were written to test construction, parsing and validation of messages. The latter was tested using both correct and incorrect messages.

3.5 Integration Testing: Model-based

Test Architecture. We now look in more detail at integration testing of XBus, which we do model-based using JTorX. We want to test whether the XBus interacts correctly with its environment, i.e. with XBus clients. We do this by letting the tester play the role of 3 XBus clients as visualized in Fig. 5.

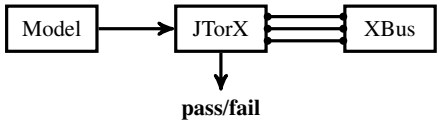


Fig. 5 Testing XBus with JTorX playing the role of 3 clients.

Now we have to decide how to connect test tool JTorX to XBus server. We briefly discuss two alternatives that we did not choose, and then in more detail the one we chose.

A first solution (see Fig.6) could be to let JTorX interact with three instances of the XBusGenericClient, each of which is connected to XBus via TCP. The main advantage of this solution is that no additional (testing) interface has to be added to XBus, because the interaction via the XBusGenericClient instances as (to XBus) identical to the interaction during deployment. The disadvantage of this solution is that the testing environment contains elements that are not part of the model (we explicitly excluded them, see Sect. 3.2).

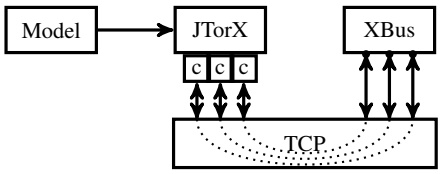


Fig. 6 First non-chosen solution: JTorX connected to XBus via generic clients (c) over TCP.

A second solution (not depicted) could be to extend the XBus server with two additional interfaces: one that allows JTorX to add messages to XBus' queue of incoming messages (to provide stimuli), and one that, instead of sending a message to an XBus client, informs JTorX of the intention to send the message. The advantage

of this solution is that the actual test architecture very closely resembles the model. However, it has the disadvantage that the XBus implementation has to be extended in multiple places.

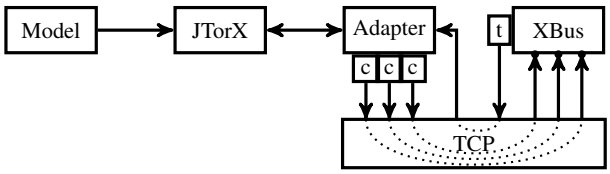


Fig. 7 Chosen solution: JTorX providing stimuli to XBus via generic clients (c) over TCP, and observing responses via test interface (t), also connected via TCP.

The chosen solution (see Fig. 7) combines the best of the two that we discussed above. We provide stimuli to the XBus using three XBusGenericClient instances that connect to the XBus via TCP. We observe the responses from the XBus not via the XBusGenericClient, but via a direct (testing) interface that has been added to XBus. This interface is provided by the JTorXTestableXBusManager in the Engine package, see Fig. 4. JTorXTestableXBusManager overrides the function that XBus uses to send a message to a specified client, and instead logs the message name and relevant parameters in the textual format that JTorX expects. Additional glue code —the adapter— provides the connection between JTorX and the XBusGenericClient instances on the one hand, and between JTorX and XBus test interface on the other hand. From JTorX the adapter receives requests to apply stimuli, and from the XBus test interface it receives observed responses. The adapter forwards the received responses to JTorX without additional processing. For each received request to apply a stimulus the adapter uses XBusGenericClient methods to construct a corresponding XBusMessage message and send it to the XBus server (except for the $\mathsf{Conn}_{\mathsf{req}}$ message, for which XBusGenericClient only has to open a connection to XBus).

Running JTorX Once we had the model, the XBus implementation to test, and the means to connect JTorX to it, testing was started. Figure 8 shows the settings in JTorX. These include the location of the model file, the way in which the adapter and the XBus are accessed, and an indication of which messages are input (from the XBus server perspective) and which are output.

Bugs Found Using JTorX The most interesting part of testing is finding bugs. In this case, not only because it allows improving the software, but also because finding bugs can be seen as an indication that model based testing is actually helping us. We found 5 bugs of which we think that they are hard to find without a tool like JTorX. Some of them are quite subtle:

1. The $\mathsf{Notif}_{\mathsf{disc}}$ message was sent to unsubscribed clients. This was due to an if-statement that had a wrong branching expression.

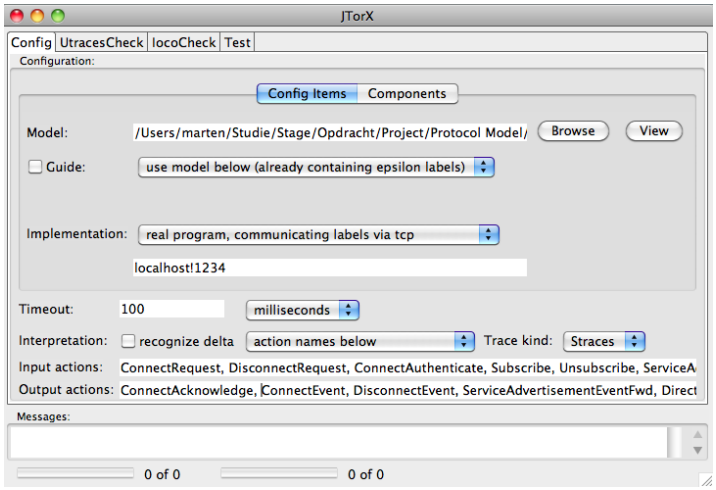


Fig. 8 Screen shot of configuration pane of JTorX, set up to test XBus. JTorX will connect to (the adapter that provides access to) the system under test via TCP on the local machine, at port 1234. The bottom two input fields list the input and output messages.

2. The $\mathsf{Serv}_{\mathsf{ann}}$ message was sent (also) to unauthorized clients. Clients that were still in the handshake process with the server, and thus not fully authenticated, received the $\mathsf{Serv}_{\mathsf{ann}}$ message. To trigger this bug one client has to advertise its service while another client is still connecting. Because this only occurs with a certain interleaving of messages, it might be more difficult to detect with manual testing.
3. The message subscription administration did not behave correctly. It was possible for a client to subscribe to one item, but *not* to two or more. This was due to subtle a bug in the operation that added the subscription to the list of a client.
4. The same bug also occurred with the list of provided services. It was implemented in the same way as the message subscription administration.
5. There was a subtle flaw in the method that handles Unsub messages. The code that extracts subscriptions from these messages (such that the found subscriptions can then be removed from the list of subscriptions of the corresponding client) contained a subtle typing error: two terms in an expression were interchanged.

4 Lessons Learnt

This section reflects on the process of designing, implementing and testing XBus. All in all the end result was satisfying, the approach has proven to be successful.

Putting it in a Time Perspective So how long did it take to create the artefacts for model-based testing, namely model, the test interface and the adapter? Programming

and simulating the model took 2 weeks, or 80 hours. The test interface was created in a few hours, since it was designed to be loosely coupled to the engine. It was a matter of a few dozens lines of code.

The adapter was created in two days, or 16 hours. In total the project took 14 weeks, or 560 hours to complete. Creating the artefacts needed for automated testing thus took about 17% of the total time.

The Modeling Process Writing a model takes a significant amount of time, but also forces the developer to think about the system behaviour thoroughly. Moreover, it is really helpful to be able to simulate a protocol, before implementing anything. Making and simulating a model gives a deep understanding of the system, in an early stage of development, from which the architectural design profits.

Automated Testing with JTorX Write an adapter can sometimes be a large project, but in this case it was relatively straightforward. This can be attributed to having an architectural design that closely resembles the formal model, having a one-to-one mapping between the actual XBus messages and their model representation. Therefore, such adapters can in principle be generated automatically, thus greatly facilitation the model-based testing process.

5 Conclusions and Future Research

We conclude that model-based testing using JTorX was a success: with a relatively limited effort, we found five subtle bugs. We needed 17% of the time to develop the artefacts needed for model-based testing, and given the errors found, we consider that time well spent. Moreover, for future versions of the XBus, JTorX can be used for automatic regression tests: by adapting the MCRL2 model to new functionality, one can detect automatically if there are new bugs introduced.

We also conclude that making the formal model together with the architectural design had a positive effect on the quality of the design. Moreover, the resulting close resemblance between model and design simplified the construction of the adapter.

Although construction of the adapter was relatively straightforward, it would have been even easier if (parts of) the adapter could have been generated, e.g. using the model, which is an important topic for future research.

References

[1] (2009) JTorX webpage. URL http://fmt.ewi.utwente.nl/tools/jtorx/
[2] (2009) mCRL2 toolkit webpage. URL http://www.mcrl2.org
[3] (2009) Neopost Inc. webpage. URL http://www.neopost.com
[4] (2011) Quasimodo webpage. URL http://www.quasimodo.aau.dk/

[5] Belinfante A, Feenstra J, de Vries RG, Tretmans J, Goga N, Feijs L, Mauw S, Heerink L (1999) Formal test automation: A simple experiment. In: 12^{th} Int. Workshop on Testing of Communicating Systems, Kluwer, pp 179–196

[6] Belinfante AFE (2010) JTorX: A tool for on-line model-driven test derivation and execution. In: Esparza J, Majumdar R (eds) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010), Springer Verlag, LNCS, vol 6015, pp 266–270

[7] Bergstra JA, Klop JW (1985) Algebra of communicating processes. In: de Bakker JW, Hazewinkel M, Lenstra JK (eds) Proceedings of the CWI Symposium on Mathematics and Computer Science, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

[8] Bohnenkamp HC, Stoelinga MIA (2008) Quantitative testing. In: de Alfaro L, Palsberg J (eds) Proceedings of the 7th ACM International conference on Embedded software, ACM, New York, pp 227–236

[9] Brandán Briones L (2007) Theories for model-based testing: Real-time and coverage. PhD thesis, University of Twente

[10] David A, Larsen KG, Li S, Nielsen B (2009) Timed testing under partial observability. In: ICST, IEEE Computer Society, pp 61–70

[11] Frantzen L, Tretmans J, Willemse TAC (2006) A symbolic framework for model-based testing. In: Havelund K, Núñez M, Rosu G, Wolff B (eds) Formal Approaches to Software Testing and Runtime Verification, Springer, Lecture Notes in Computer Science, vol 4262, pp 40–54

[12] Grieskamp W, Qu X, Wei X, Kicillof N, Cohen MB (2009) Interaction coverage meets path coverage by SMT constraint solving. In: Núñez M, Baker P, Merayo MG (eds) Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Springer, Lecture Notes in Computer Science, vol 5826, pp 97–112

[13] Groote JF, et al (2008) The mCRL2 toolset. In: Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)

[14] Larsen KG, Mikucionis M, Nielsen B (2004) Online testing of real-time systems using uppaal: Status and future work. In: Brinksma E, Grieskamp W, Tretmans J (eds) Perspectives of Model-Based Testing, Dagstuhl Seminar Proceedings, vol 04371

[15] Rook PE (January 1986) Controlling software projects. IEE Software Engineering Journal 1(1):7–16

[16] Tretmans J, Brinksma H (2003) TorX: Automated model-based testing. In: Hartman A, Dussa-Ziegler K (eds) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp 31–43