An Online Model-Checking Framework for Timed Automata

Applying Formal Verification to Medical Cyber-Physical Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Jonas Rinast

aus

Lübeck

2015

1. Gutachter: Frau Prof. Dr. Sibylle Schupp

2. Gutachter: Herr Prof. Dr. Dieter Gollmann

3. Gutachter: Herr Prof. Dr. René Rydhof Hansen

Tag der mündlichen Prüfung: 04.09.2015

Acknowledgments

The present dissertation has been carried out at the Institute for Software Systems and the Institute for Security in Distributed Applications at the Hamburg University of Technology.

At this point I would like to express my best thanks to my main advisor Prof. Dr. Sibylle Schupp, head of the Institute for Software Systems, for all her support and expertise during the planning, development and implementation of the research. Many insightful discussions were held that helped spark new ideas and guide my research.

I would also like to thank my second advisor Prof. Dr. Dieter Gollmann, head of the Institute for Security in Distributed Applications, for providing thoughts from different perspectives that helped distinguish the important from the unimportant and keep my research focused.

Next, I would like to thank Prof. Dr. Alexander Schlaefer, head of the Institute for Medical Technology, for supporting the final medical case study and supplying the medical data used in it.

Special thanks goes to Xintao Ma and Axel Neuser, two students whose Bachelor and Master theses I supervised, and whose results contributed to my dissertation and advanced my research.

In addition, I want to thank all members of the Ambient Assistance for Recovery (AA4R) initiative of the Hamburg University of Technology for criticizing my research during several lengthy workshop sessions and thus improving my final work.

Of course all the various institute members that I not only had research-focused discussions with but also relaxing smalltalk also deserve a thanks for creating a pleasant working environment.

Leaving the academic field, I wholeheartedly thank my long-time girlfriend, Gabi, who always cheered me up when I felt demotivated and comforted me after long days with my dissertation.

Lastly, I would like to thank my parents for always supporting me in what I do and teaching me valuable life lessons that ultimately made this dissertation possible.

– „THANKS"

Abstract

Over the last years the number of cyber-physical systems that provide critical functionality has increased continuously. Ensuring the correct behavior of those systems is essential to prevent potentially catastrophic malfunctions. This dissertation is about online model checking, a dynamic variant of model checking that can provide safety guarantees even if accurate long-term modeling of a system is infeasible, which is often the case for cyber-physical systems. In particular the main contributions of this dissertation are in analyzing and automating the online model-checking approach with the model checker UPPAAL.

As the first contribution, I present the transfer of an online model-checking case study that uses a laser tracheotomy scenario from a formalization for the hybrid model checker PHAVer to an UPPAAL-SMC formalization. This case study focuses on examining whether UPPAAL can be used as the verification engine in an online model-checking context. Experiments show that the employment of UPPAAL is feasible and that easing the development process of online model-checking applications is desirable. Therefore, the second contribution of this dissertation is a framework for the development of online model-checking applications with UPPAAL. The framework provides components that help to implement a complete online model-checking application. Features include a data acquisition and processing pipeline, automatic model reconstruction and adaptation, and seamless integration of UPPAAL for verifications. The most notable feature is the automatic state space reconstruction, which empowers the user to easily define model and state adaptations to adjust the model to real-world observations. An algorithm for automatically modifying a model such that it has a particular initial state, which is required in this state space reconstruction and adaptation step, is the third contribution. Furthermore, as online model checking imposes real-time deadlines on the runtime of the state space adaptation I present two optimizations for the reconstruction algorithm. One is based on projections in a graph representation of the state space and the other one is based on use-definition chains, an adaptation of a static data flow analysis technique. Both optimizations remove unnecessary transformations from a transformation sequence and can be used in other contexts as only a general transformation system is assumed. For both optimizations I provide formal correctness arguments and proofs. The last contribution is an online model-checking case study on robotic radiosurgery developed with the framework. The case study evaluates the capabilities of the framework while developing models for automatic compensation of breathing movements. The study shows that the framework provides all means necessary to specify such an application and that it is capable of meeting the real-time deadlines of online model checking.

Contents

1 Introduction

Technological advances in all domains lead to the development of more and more complex systems that support humans in various ways. With the increasing complexity of the systems understanding all the processes inside these systems becomes more and more difficult. Unforeseen interactions between components of a system can cause complications and compromise system integrity.

The research field of *formal methods* comprises means to analyze such systems in a formal way. Sound mathematically methods enable us to reason about properties of systems. These properties can then be used to guarantee correct behavior. One such formal method is the well-researched *model-checking* approach. Classic model checking systematically searches the state space of a previously developed system model for property violations. It then returns a verdict on the property's truth that can then be used to ensure the system's correctness.

One big class of systems that requires such thorough analysis is the class of *cyber-physical systems*. Cyber-physical systems are systems that directly interact with the real world, e.g., cars, medical devices, or airplanes. Therefore, these systems always include virtual and physical components, which need to be connected in safe ways, and the correct interaction between them must be ensured. When applying the model-checking approach to such real-time systems an accurate model of the system must be created during its development and, even though multiple modeling formalisms and implementing tools exist, often the formalism of *timed automata* is the formalism of choice as timed automata not only provide means to model discrete transition events but also the characteristic timing of the transitions.

However, the modeling of a cyber-physical system can be very difficult. System parts may behave non-deterministically and accurate models may not be available. For example, in the medical domain recent research focused on closed-loop treatment for patients. In such cases a model that represent the patient's physiological reactions to the treatment is necessary. But accurate long-term prediction of the human body is infeasible most of the time except for special cases. Thus, the model-checking approach is not suitable to analyze such systems because the guarantees obtained with it would not apply to the corresponding real-world system because of the model inaccuracies.

Recently a variant of model checking, the online model-checking approach, was proposed as a method to still obtain guarantees on such systems where accurate long-term models are difficult to obtain. Instead of statically proving the safety of a system before the system is deployed, online model checking (OMC) is a dynamic technique that monitors the system while it is running. Periodically OMC performs a classic model-checking analysis to obtain guarantees on the current system state and its near future. When the analyses are scheduled such that the validity scopes of the guarantees overlap, one obtains safety guarantees for the complete execution of the system. As a result if a problem arises a grace period that allows timely countermeasures is always available to ensure safe operation. The benefit of this approach is that at the verification seams the simulated state or the model itself can be modified.

When done correctly such adjustments can ensure that the used model represents the real-world system correctly even though the model is of approximate nature only.

In this dissertation I investigate the online model-checking approach in the context of medical cyber-physical systems with the UPPAAL model-checking tool. UPPAAL is a state-of-the-art model-checking tool for timed automata that is still improved continuously and that has already been used in the industry successfully. The main contribution of this dissertation is a framework for the development of online model-checking applications with UPPAAL. The framework lets the user define an OMC application by specifying the properties to verify, the verification intervals, the adjustments for the simulation state and model, and sensors to obtain data from the real world. It is available for download at `http://www.tuhh.de/sts/research/projects/online-model-checking.html`. The model for the system itself is still created with the graphical user interface (GUI) of UPPAAL. This approach allows the reuse of older UPPAAL models and decouples system modeling from the OMC implementation. When everything is specified the framework works fully automatic: after the start of the system the UPPAAL model is simulated synchronously to the real-world system and after every verification interval an adjustment process followed by a verification is initiated. During the adjustment process the sensor data is evaluated as specified by the user and state and model adjustments are performed. This phase requires the reconstruction of the previous state space to obtain seamless transitions and to perform the modifications. This *state space reconstruction* is the main theoretical contribution of this dissertation. I developed two optimization algorithms for the reduction of transformations in a generic transformation system that can be used for this automatic reconstruction by specializing the general algorithms to the UPPAAL transition system. The optimization of the reconstruction is necessary as OMC constitutes a real-time system where guarantees must be obtained within their verification intervals. If deadlines are exceeded safe behavior can no longer be ensured. In the following verification step the new model is subject to a model-checking analysis with the user-defined properties. If the verification fails at any point an alarm is raised and counter-measures can be performed.

Moreover, in addition to the OMC framework the contributions of this dissertation include two case studies of online model checking in the medical domain. One has been carried out before the development of the framework to assess its viability; the other one was performed with the framework for evaluation purposes. The preliminary case study ensures the safe operation of a laser scalpel during a laser tracheotomy, i.e., a surgery that opens a direct airway to the patient's windpipe to alleviate breathing. The case study models the patient, the ventilator unit, and the laser scalpel and it incorporates a supervisor component that enforces a safe usage protocol for the scalpel. Windpipe oxygen and blood oxygen parameters are predicted, monitored, adjusted, and evaluated. The case study shows that a useful online model-checking application can be implemented with UPPAAL and that critical safety properties can be analyzed in spite of the approximate nature of the prediction models. In consequence the OMC framework was developed and an evaluation case study was conducted. That case study is positioned in the field of cancer treatment with irradiation. More specifically,

the case study is assessing the safe operation of performing radiation therapy on lung cancers with robotic radiosurgery devices, which is currently explored in medical research. When using robotic radiosurgery devices for such treatment the most critical safety requirement is that the irradiation beam is correctly targeting the tumor cells. In the case of lung cancers accurate targeting during treatment is difficult as the patient is not only moving because of regular respiration but also due to breathing artifacts like coughing. The OMC case study benefits this system by providing a formal, explicit prediction model for the breathing process of the patient capable of detecting abnormal breathing artifacts. As a result the case study not only provides a base for a thorough safety analysis of robotic radiosurgery applications but also shows that the developed OMC framework is suitable for the development of complex OMC applications.

In conclusion, this dissertation provides a base for further research in the field of online model checking in an attempt to facilitate thorough safety analyses of critical cyber-physical systems whose modeling is challenging. The variation of the classic model-checking approach, which is applicable to systems whose models are inaccurate for long-term predictions, advances the field of formal methods into a new territory where system correctness is still crucial but classic static approaches fail to provide meaningful assessments. The developed framework may help exploring this field and lead to safer systems in general in the long term.

The remaining dissertation is structured as follows: the next chapter, Chapter 2, sets the context of this dissertation in detail by presenting related publications. It focuses on the modeling aspect of systems, the verification of systems, model checking and online model checking, optimization techniques for implementation purposes, and case studies that have been performed. The publications are briefly introduced and their relations to this dissertation are made clear.

Chapter 3 then introduces the theoretical foundation required for understanding this dissertation. Starting with formal modeling first simple finite automata are introduced, which are then extended to the timed automata formalism and the hybrid automata formalism. Then, the model-checking approach is presented by giving algorithms and logic specifications. Statistical model checking is also briefly introduced as the UPPAAL tool also allows statistical analyses. Next, the UPPAAL tool itself with its modeling language and its extension for networks of timed automata is presented. Formal definition of the semantics are given. The statistical component of UPPAAL, UPPAAL-SMC, is also covered briefly. Lastly, the chapter presents the online model-checking approach. OMC is motivated and the process is explained in detail.

Afterwards, Chapter 4 presents the preliminary laser tracheotomy case study used to assess online model checking with UPPAAL. The medical requirements for a safe surgery are shown and corresponding safety properties are derived. The UPPAAL models for all system components are shown and the results obtained by the experiments performed are analyzed.

The following Chapter 5 showcases the developed online model-checking framework. Its architecture, features, and components are shown and examples of the development

of an application with it are given. Also, the accompanying GUI that gives insight into a running OMC application is shown and an example application is presented.

The main theoretical contributions of this dissertation are the topic of Chapter 6: the adaptation of simulation states and the underlying models to the real-world system. Different approaches to adaptation are introduced and accompanying challenges are pointed out. Then, for the actual implementation of the state space reconstruction a formal definition of a general transformation system is given and the two reduction algorithms are presented and their correctness is assessed. The first algorithm is based on a graph representation of the state space and projections are exploited to find shortcuts for the optimization. The second algorithm uses use-definition chains, a way to trace modifications by transformations, for the removal of unused transformations for its optimization. In a next step, the algorithms for the general transformation system are formally specialized to work with UPPAAL's timed automata semantics. Then, after a short section on synthesizing the adapted models from the reduction data, the results of the experiments I performed to assessed the reduction algorithms are presented and I argue that the proposed algorithms suffice for applications in the online model-checking context.

Chapter 7 then focuses on the evaluation case study in the robotic radiosurgery context. At first, the system is introduced, the UPPAAL models are shown, and the online model-checking application is specified. Then, the experiment setup is defined and the results of the performed experiments are shown and analyzed.

At last, Chapter 8 summarizes the dissertation, discusses the work conducted, and proposes further research topics. Topics that may be interesting for future research include different approaches to the adjustment of state spaces, the development of even better reconstruction algorithms, techniques to couple the model simulation to the real-world system, or relatively simple extensions of the framework for usability purposes like a builder GUI for the specification of OMC applications.

2 Context and Related Work

Generally, this dissertation is motivated by ensuring the safety of patients in the medical domain. From the point of view of formal methods the medical domain is challenging as components of such systems, e.g., a patient, often exhibit behavior that is difficult to predict. Therefore, the assurance of safety in a general sense is a very broad, difficult goal and in the literature many different areas are explored to achieve this goal. A common denominator for most approaches though is that they employ some kind of model to represent some process.

The subject of the modeling process though is quite varied even when only focusing on the medical domain. Subjects include models for organizing the handling of patients, e.g., in the form of protocols [77] or workflows [34], as well as behavioral models for medical components. These behavioral models can further be divided into models that represent an actual technological device like an infusion pump [33] and models for predicting and understanding human physiology, e.g., in the form of a model of the human heart [76, 77]. Moreover, the behavioral models also include models that combine models of multiple components for an analysis of complete medical systems like monitoring systems for patient rehabilitation [96]. This dissertation focuses on the last class of models, which are used to analyze complete medical systems. These models pose additional obstacles due to the interconnection of devices and the dynamic nature of components. An example of such a system that this dissertation is concerned with can be found in a notable case study by King et al. that analyzes a closed-loop control system for treating patients automatically with a patient-controlled analgesia pump (PCA pump) [57].

When discussing the modeling of such systems two basic questions need to answered: first, one needs to decide what the underlying formalization for the model should be and, second, how the model should actually be obtained.

For the formalization choice the formal methods field provides several viable options: it can employ mathematical language, domain-specific language, or formal logic. For example, mathematical models exist for implantable pacemakers [22], and domain-specific semantics for sensor networks have been developed [37]. However, most of the time previously defined, well-studied formalizations are used for system modeling to make use of advances in the formal methods field. Such formalizations include event-based methods [98] as well as automata-based approaches and multi-model techniques [12, 87]. Common automata-based formalizations are timed automata for the modeling of real-time systems with timing constraints [6, 21] and hybrid automata for systems with dynamic and discrete behavior [45]. In this dissertation I focus on the timed automata formalism as timed automata are a well-known formalization with good tool support and timing is often crucial for safety in the medical domain. Furthermore, timed automata models can often be changed to hybrid models without much effort if a medical system requires the additional modeling functionality.

When choosing how to create a model the obvious way that comes to mind is creating the model from scratch if the system is understood well enough to make

such a creation feasible. However, this is not the only approach. Pattern-based model creation allows the reuse of established architectures that may benefit the model accuracy or performance. For example, Yun et al. propose a hierarchical structure for cyber-physical systems (CPS), i.e., systems with a physical and a virtual component like medical systems, to keep the state space small [101] and Al-Nayeem et al. employ an architectural pattern to tackle multi-rate CPS efficiently [3]. If for some reason the construction of the model by hand is impossible it may be possible to learn a model by observation of the system. Aarts et al., for instance, derive automatically a model for biometric passports by employing a complexity-reducing abstraction technique [1]. Other model synthesis approaches focus on finding optimal models because classic inference techniques may sometimes produce unsatisfactory models. For example, Bohlin et al. propose a technique to improve model synthesis attempts by imposing a structure on the underlying model [23]. In this dissertation the challenge of optimal model synthesis also arises. In general, accurate long-term models for the reactions of patients to treatment are not available due to missing information and knowledge of human physiology. In an attempt to still obtain valid models I observe the real-world system and synthesize an appropriate model. For the synthesis I employ a minimization algorithm that is similar to an algorithm proposed by Alur et al. that uses a projection-based optimization approach [5]. The research field of model repair is closely related to my synthesis approach and is relevant to this thesis as it may provide valuable insight in potential model adjustments that modify a model to match the real-world system. In model repair potential adjustments include the modification of transition probabilities [30] as well as restricting control to sequences that satisfy certain properties [11].

Several overview studies identify the development of safe software as a crucial research topic for system development in the future [2, 31, 40]. To ensure safety one can generally distinguish between dynamic and static approaches, which both have advantages and disadvantages with regards to safety analysis. Thus, for this dissertation the general question arises on what kind of approach is appropriate.

A wide-spread dynamic approach in the software engineering context is testing. Testing supplies the system with predefined inputs and compares system behavior to the expected one. It generally is employed before the software is released and its dynamic nature stems from the fact that the program is executed to obtain the test result. How the test cases are obtained and how they are executed however varies greatly between applications. For example, in contrast to simple unit tests for program modules also CPS can be subject to testing. For instance, Jiang et al. show how model-based testing can be used to test implantable pacemakers in a closed-loop system [55]. Moreover, the UPPAAL variant UPPAAL COVER by Hessel et al. supports model-based testing of real-time systems by generating test cases from an UPPAAL model with input and output channels [46]. If desired UPPAAL COVER can also be used in a different kind of dynamic context where it checks the conformance of the system while it is running with the help of the model, which yields a hybrid testing approach. Such dynamic approaches, which run concurrently to the system in question, most of the time use some kind of monitoring system to identify problems of the running

system. For example, Bak et al. propose to encapsulate controllers of cyber-physical systems in safety-assuring sandbox modules to keep the operation of the controllers in safe regions [10]. Other dynamic approaches propose raising alarms on-the-fly. For example, Goldfain et al. [43] and Ko et al. [58] evaluate the conditions of patients that are monitored and raise alarms if their conditions deteriorate.

In contrast to the dynamic approaches that execute the system static safety assurances are generally obtained by an analysis of the system specification. Static assurances may be obtained by simulation often in conjunction with the determination of safe parameters for the system. Arney et al., for example, simulate a closed-loop PCA pump system in Matlab/Simulink to obtain deadlines for the safe infusion of drugs [8]. The most thorough and complete safety assurances however are obtained with a formal verification of the system. Thus, Arney et al. also verify an UPPAAL model with the obtained deadlines to ensure that the pump is controlled correctly in the same case study [8]. Several other case studies have employed verification techniques to guarantee safe operation in the medical domain: Ganesan et al. extract structures from the code base of the control software of a blood pressure maintaining device and upon verification discovered that the code base had several dead functions compromising its safety [42]. Hooman et al. verify an x-ray system with the compositional model checker ASD:Suite and identify invalid communication between components that may risk safety [50]. Furthermore, Jiang et al. formally guarantee correct stimuli of an implanted dual chamber pacemaker when the heart exhibits abnormal conditions showing along the way that open-loop testing can not identify all problems [56]. Chen et al. work on the verification of pace makers as well: they combine model checking in PRISM with simulation techniques in MATLAB to show that a pacemaker corrects Bradycardia (slow heart beat) without inducing Tachycardia (fast heart beat) [29].

In general, the literature shows that safety assurances with static verification techniques are of higher quality and are able to identify more problems than pure testing and simulation approaches. In this dissertation I thus focus on the formal verification of medical systems using model checking. Model checking has been explored in the literature in many different directions. The general approach of symbolic model checking for real-time systems was introduced by Alur et al. [4]. Larsen, Pettersson, and Yi extended the approach for timed systems and also dealt with questions regarding model composition [64, 65]. Larsen et al. also show that symbolic model checking may provide diagnostic traces for debugging purposes during modeling [66]. A thorough analysis of symbolic model checking with timed automata can be found in Pettersson's PhD thesis [81].

Statistical model checking takes a different direction in that it allows the modeling and verification of probabilistic systems. Legay et al. provide an overview of the technique and algorithms [68]. More complete information on statistical model checking can be found in Wang's PhD thesis [95]. Recently, UPPAAL was extended to also provide statistical model-checking capabilities. David et al. and Bulychev et al. introduce this extension [27, 33]. Bulychev et al. also expand on the extension to check Weighted Metric Temporal Logic [26].

Model-checking variants focusing on solving specialized problems have also been

proposed. Asarin and Maler, for example, use the notion of a timed game to reach a certain model configuration as soon as possible [9] and Alur et al. consider the optimal-reachability problem where they try to minimize a cost function while traversing a timed automaton [7]. Larsen et al. also deal with such a minimization of a cost function during traversal and introduce an extension for Linearly Priced Timed Automata (LPTA) and implement it in UPPAAL for this problem [61].

In a recent position paper Calinescu, Ghezzi, Kwiatkowska, and Mirandola identify the combination of static and dynamic approaches to ensure safe operation of systems as one of the main challenges to advance the development of safe software and systems [28]. In this dissertation I explore the application of static verification techniques in a dynamic monitoring context as up to now most dynamic approaches do not employ formal methods. Koushanfar et al., for example, use an approach with function minimization and statistical methods to identify faulty sensors in sensor networks [59]. However, recently, research has started to incorporate model checking into dynamic safety assurance methods resulting in the notion of online model checking. Sauter et al., e.g., propose a hybrid model-checking approach that involves an offline and an online phase to increase performance of complex models [88] and Qi et al. employ a monitoring system to check distributed C++ web services for safety and liveness properties [82]. Furthermore, Li et al. perform online model checking to ensure the safety in a laser tracheotomy scenario using the hybrid model checker PHAVer [70, 71]. I use this case study as a base for the preliminary case study conducted in this dissertation. Moreover, the online model-checking approach is also proposed by Zhao and Rammig to reduce the model state space for verification [102]. Lastly, Bu et al. extended the model checker BACH to facilitate online model checking in the prototype BACHOL [25].

A common feature of most publications for model checking is that they involve some software tool that actually carries out the experiments or that contains the proposed optimization or feature. One such tool already mentioned multiple times is UPPAAL, a model checker for timed automata, and this is also the tool that I use in this dissertation. Its first implementation was created in 1994 by Yi, Petterson, and Daniels [99]. Over the years continually improvements have been made to UPPAAL and a summary of most features is given by Larsen et al. [67]. For an introduction to UPPAAL and the main contributions in version 4.0 see the relatively recent tutorial paper by Behrmann et al. [15]. Details on the implementation can be found in another paper by Behrmann et al. [13] or in the PhD thesis of Bengtsson [18].

Over the years several other model-checking tools have also been developed. Most differ in functionality, usability, and performance. A tool also based on the theory of timed automata is the Kronos tool introduced by Yovine in 1997 [100]. The PRISM model checker is a model checker specializing in probabilistic systems and information about the most recent release is provided by Kwiatkowska et al. [60]. The BACH tool is a model checker for compositional linear hybrid systems that is founded on the theory of constraint solving. Details on the BACH 2 release can be found in the publication of Bu et al. [24]. At last, the model checker SPIN focuses on proving the correctness of process interactions. Holzmann provides an introductory paper to

the tool [49]. The choice of focusing on the well-developed UPPAAL tool in this dissertation can be attributed to the fact that many optimizations and spin-off tools for it have already been developed successfully and its development is still ongoing, which promises support in the future and acceptance in the research community.

UPPAAL's relevance as a verification tool can also be seen by its usage for several case studies that analyze systems like a gearbox [72] or protocols like the Collision Avoidance Protocol [53], the startup protocol for the Time Division Multiple Access protocol [73], an audio control protocol [19], the biphase mark protocol [94], and the Web Services Atomic Transaction protocol [83].

When exploring the online verification research field one also finds the closely related field of runtime verification. Research on runtime verification has recently diverged into two dimensions from the original model-less monitoring technique for software execution paths [41, 91]. The first dimension is to incorporate explicit models into the runtime verification applications blurring the lines between runtime verification and online model checking [36, 47] even though runtime verification still only detects safety violations as soon as they happen in contrast to online model checking, which tries to predict such events. The other dimension is that runtime verification has spread into domains other than the pure software monitoring application. The class of cyber-physical systems, which includes the class of medical systems that motivate this dissertation, has especially gained interest: control-theoretic systems [39], robotic applications [51], and CPS safety in general [78] have been explored by the runtime verification community. An overview of the runtime verification field is provided by Leucker et al. [69].

Summarizing, the combination of static and dynamic safety assurance techniques has recently gained interest not only in the domain of formal methods but also in the software engineering and the CPS domain. The knowledge transfer between all involved fields promises advances in every field and therefore I explore online model checking with the UPPAAL tool in this dissertation, using cyber-physical systems from the medical domain as motivating examples for applications that may benefit from increased CPS safety in the future.

The way I implement the online model-checking approach in my framework for UPPAAL many implementation details of UPPAAL's model-checking algorithm, data structures, and optimizations are relevant as the implementation ties directly into UPPAAL's implementation and, for example, the data structures are reused. When employing online model checking for monitoring a system the system becomes a real-time system as OMC requires that guarantees are obtained within specific deadlines. Thus, in this dissertation performance of the individual components of the OMC framework was a main concern during the implementation. When trying to increase performance not only the size of every individual state is relevant but also the overall number of states. In the literature thus approaches to reduce the memory consumption per state and the size of the state space are proposed.

For the representation of states efficient data structures have been developed to reduce the overall memory consumption. Larsen et al. propose a compact data structure, difference bound matrices, for the efficient storage of timed automata time states [62].

Bengtsson improves upon this representation by incorporating a compression technique and excluding unnecessary states early [17]. Furthermore, Larsen et al. reduce the memory consumption by employing a static analysis to find minimal sets that cover all dynamic loops of an automata network [63].

For the reduction of the state space a number of state space reductions have been proposed over the years and incorporated into the UPPAAL tool. Many of them have some influence on the implementation of online model checking with UPPAAL: a first reduction is the elimination of unnecessary cycles in automata networks as proposed by Larsen et al. in their state representation paper [62, 63]. A second reduction is the partial order reduction introduced by Bengtsson [20]. This optimization lets clocks of individual processes vary independently and resynchronizes them when necessary. This approach eliminates implicit synchronization requirements. A third reduction present in UPPAAL is symmetry reduction as implemented by Hendriks [44]. Basically, the fact that parts of the model are identical can be made explicit and then the model checker only has to consider non-symmetric parts.

Many other optimizations, however, are not directly implemented in the model-checking engine but aim at reducing the state space by changing the input model. One such optimization is slicing as proposed by da Cruz [32]. Here a model is split into multiple parts such that all parts can be verified individually leading to a contract-based verification approach. Another approach involves reducing redundancies in a model by exploiting projections between different states. This reduction is highly relevant to my graph-based transformation reduction used for state space reconstruction in the online model-checking framework developed in this dissertation as it uses a similar projection approach. It was developed by Su et al. [92]. Another class of model optimizations employ abstraction and refinement strategies to adapt models to particular verification problems. Daws and Tripakis, for example, propose and implement in KRONOS a reachability preserving abstraction to overcome the state space explosion problem of model checking [35]. Janowska et al. compress paths in systems where the intermediate states are irrelevant to a particular property result to obtain an abstract model for that property [52]. This compression is similar to the implementation of the state space reconstruction algorithms I developed as I also remove intermediate states without influence on the final reconstructed state. Behrmann et al. derive lower and upper bounds for clocks to obtain a zone-based abstraction for clock values, which yields an optimization for dealing with difference bound matrices that I use for state representations [14]. Thacker et al. discuss an abstraction technique for cyber-physical systems where often only small parts of a system need to be accurately modeled to obtain satisfactory results [93]. The relevance to the dissertation here is that the resulting abstracted models often are approximate and could benefit from the OMC approach and the developed OMC framework. Duggirala and Mitra incorporate a refinement step in addition to abstraction into their approach to verify the stability of CPS models: guided by counter-examples they abstract unnecessary parts of the model and refine relevant parts until the verification of stability becomes feasible [38]. This approach demonstrates a potential adaptation strategy for OMC model synthesis to obtain approximate models of sufficient accuracy. At last, Jiang et

al. use an abstraction and refinement approach to verify a closed-loop medical device system involving pacemakers with UPPAAL and generate necessary controller code from the models, which also demonstrates modeling for an OMC application in the medical domain [54]. All these optimization techniques can be considered when implementing an online model-checking framework for UPPAAL as the model adaptation step not only modifies parts of the state but it can also incorporate abstraction and refinement approaches to keep online model checking within its real-time deadlines and to guarantee the necessary accuracy of the models.

Summarizing this chapter, one can say that already a significant amount of research has been conducted on model checking in general and safety assurances for CPS. However, the online model-checking variant that I facilitate in this dissertation is still an upcoming research field and few publications exist that directly focus on it. A few recent publications originating from the related field of runtime verification employ an approach similar to OMC and even fewer publications close to OMC have a background in formal methods. Furthermore, the only tool support for online model checking currently available is the BACHOL extension for the BACH model checker for hybrid systems that uses constraint solving. My framework for UPPAAL thus is novel in the sense that it enables online model checking using timed automata as the underlying formalism. With the research field presented, we can now continue to the next chapter where the theoretical foundation of this dissertation is laid out and formal definition for all employed constructs are given.

3 Theoretical Foundation

In this chapter the theoretical foundation for the rest of the dissertation is laid out. Standard formalizations and analysis approaches are presented and the model-checking tool used, UPPAAL, is introduced.

At first, finite automata are introduced as a modeling formalism in Section 3.1 and two extended variants, timed automata and hybrid automata, are formalized. Next, the model-checking approach is presented in Section 3.2 and two variants, the symbolic and the statistical approach, are explained. Then, Section 3.3 introduces UPPAAL, the model-checking tool used in this dissertation. Special attention is given to features of UPPAAL not present in standard automata formalizations. Lastly, the online model-checking approach is motivated and broken down into its essential parts in Section 3.4.

3.1 Finite Automata

Finite automata, also known as finite state machines (FSM), are a modeling formalism that represents systems with transitions between states of sytem components. Finite automata can easily be visualized using a directed graph. Figure 3.1 shows an example of a finite state machine of a simple heater. The example automaton has three locations: **Off**, **Heating**, and **Idle**, where **Off** is the initial location as indicated by the double circle. The edges annotated with the actions `SwitchOn` and `SwitchOff` represent external input that switches the heater on or off. The internal action ϵ indicates that an edge may be fired without external input. The model thus represents a heater that, when switched on, may alternate between the **Heating** and **Idle** locations. For the context of this dissertation finite automata are defined as follows:

Definition 1 (Finite Automaton (FA) / Finite State Machine (FSM))**.** *A finite automaton F is a tuple $F = (L, l_0, A, E)$ where L is the finite set of locations, $l_0 \in L$ is the initial location, A is the set of actions including the internal action ϵ, and $E \subseteq L \times A \times L$ is the set of edges.*

The example automaton can thus be defined by specifying

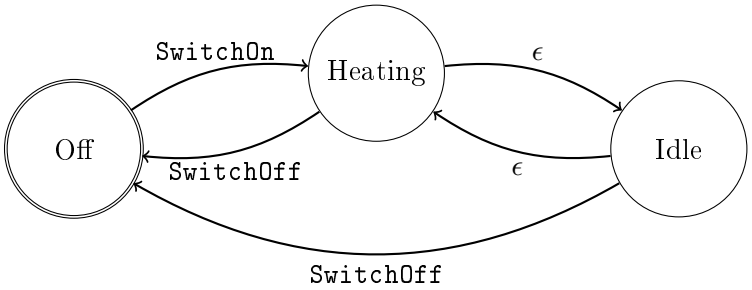


Figure 3.1: Example of a finite state machine for a heater

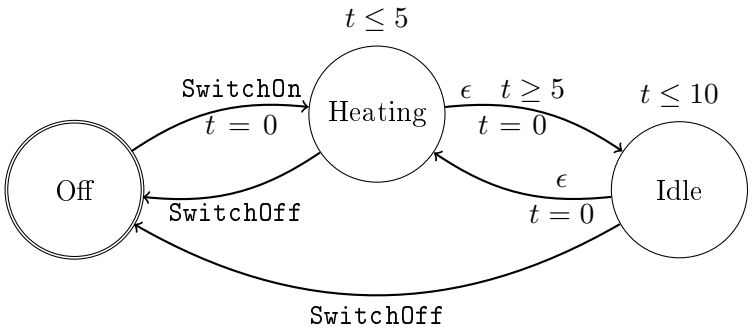


Figure 3.2: Example of a timed automaton for a heater

- $L = \{$ Off, Heating, Idle $\}$
- $l_0 =$ Off
- $A = \{$ SwitchOn, SwitchOff, $\epsilon \}$
- $E = \{$ (Off, SwitchOn, Heating), (Heating, SwitchOff, Off), (Heating, ϵ, Idle), (Idle, ϵ, Heating), (Idle, SwitchOff, Off) $\}$

The state $s \in S$ of a finite automaton is completely defined by a single location, i.e., the set of all states S is equal to the set of the locations L of the FSM. The standard notation for state transitions is $s \Rightarrow s'$, i.e., the state s transitions to the new state s'. The semantics of the transition system of a finite automaton is then given by the sole transition rule:

$$l \overset{a}{\Rightarrow} l' \text{ if } (l, a, l') \in E.$$

With finite state machines defined as a basis I now introduce timed automata in Subsection 3.1.1 and hybrid automata in Subsection 3.1.2 as more powerful modeling formalizations.

3.1.1 Timed Automata

Timed automata extend finite automata with real-valued clock variables to allow a specification of time. Figure 3.2 displays a timed automaton formalization of the heater example. The finite state machine model (see Fig. 3.1) is extended with two invariants, $t \leq 5$ and $t \leq 10$, on the locations **Heating** and **Idle**, the internal edges include new reset annotations, $t = 0$, and a new guard annotation, $t \geq 5$, and the `SwitchOn` edge was modified to include a clock reset, $t = 0$. The clock annotations give the following timing behavior of the heater:

- The heater stays exactly 5 time units in the **Heating** location unless it is switched off because every incoming edge sets t to zero and edge with the guard $t \geq 5$ may only be fired when t advanced to 5.

- The heater stays at maximum 10 time units in the **Idle** location because the invariant $t \leq 10$ disallows the clock from advancing any further.
- The heater may stay switched off indefinitely as there is no restricting invariant.

Summarizing, when the heater is turned on it has cyclic heating periods of 5 time units that are at most 10 time units apart. Formalizing timed automata I define a timed automaton as follows:

Definition 2 (Timed Automata (TA)). *A timed automaton T is a tuple $T = (L, l_0, C, A, E, I)$ where L is the finite set of locations, $l_0 \in L$ is the initial location, C is the set of clock variable symbols, A is the set of actions including the internal action ϵ, $E \subseteq L \times \mathcal{P}(C) \times A \times 2^C \times L$ is the set of edges, and $I : L \to \mathcal{P}(C)$ is the mapping assigning invariants to locations, where $\mathcal{P}(C)$ is a predicate over C.*

Note that the components of an edge (l, g, a, R, l'), also denoted by $l \xrightarrow{g,a,R} l'$, are g, the guard constraint, a, the action, and R, the reset set. Using this definition the example automaton is specified by

- $L = \{\, \text{Off}, \text{Heating}, \text{Idle} \,\}$
- $l_0 = \text{Off}$
- $C = \{\, t \,\}$
- $A = \{\, \text{SwitchOn}, \text{SwitchOff}, \epsilon \,\}$
- $E = \{\, (\text{Off}, \top, \text{SwitchOn}, C, \text{Heating}), (\text{Heating}, \top, \text{SwitchOff}, \emptyset, \text{Off}),$
  $(\text{Heating}, t \geq 5, \epsilon, C, \text{Idle}), (\text{Idle}, \top, \epsilon, C, \text{Heating}),$
  $(\text{Idle}, \top, \text{SwitchOff}, \emptyset, \text{Off}) \,\}$
- $I : I(\text{Off}) = \top$, $I(\text{Heating}) = t \leq 5$, $I(\text{Idle}) = t \leq 10$

where $\top$ indicates a tautology. The state $s \in S$ of a timed automaton is defined by its location and the valuations of the clock variables C, i.e., $s = (l, v)$, where $l \in L$ is a location and $v : C \to \mathbb{R}_0^+$ is a clock valuation function. Denoting constraint satisfiability by $\models$ the semantics of a timed automaton is then defined by two transition rules:

1. Action transition
$$(l, v) \overset{a}{\Rightarrow} (l', v')$$
if $\exists (l, g, a, R, l') \in E\, [v \models g \wedge v' \models I(l')]$ where $v'(x) = \begin{cases} 0 & \text{if } x \in R \\ v(x) & \text{otherwise} \end{cases}$.
2. Delay transition
$$(l, v) \overset{\delta}{\Rightarrow} (l, v')$$
if $v' \models I(l)$ where $v'(x) = v(x) + \delta$, $\delta \in \mathbb{R}^+$.

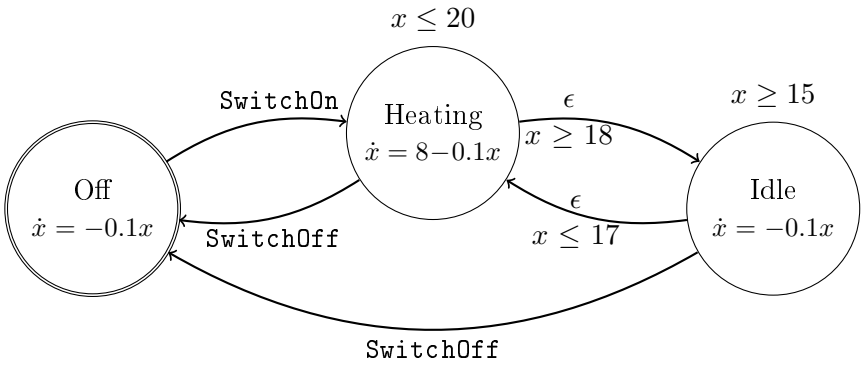


Figure 3.3: Example of a hybrid automaton for a heater

Note that under these rules action transitions are instantaneous; only delay transitions consume time. As an example, a trace in the example automaton is

$$(\text{Off}, t \mapsto 0) \xRightarrow{2.4} (\text{Off}, t \mapsto 2.4) \xRightarrow{\texttt{SwitchOn}} (\text{Heating}, t \mapsto 0) \xRightarrow{1.3} (\text{Heating}, t \mapsto 1.3)$$

if we assume an initial clock valuation function v_0 such that $\forall c \in C\,[v_0(c) = 0]$.

3.1.2 Hybrid Automata

Hybrid automata extend finite state machines with real-valued data variables such that not only discrete behavior can be modeled, but also continuous behavior. Figure 3.3 shows the heater example modeled as a hybrid automaton. All locations have been annotated with a differential equation that models how the temperature variable x evolves in the locations. Furthermore, like in the timed automaton case, invariant and guard annotations have been added to restrict when locations can be reached and when edges can be triggered. The behavior of the heater and the temperature can be described as follows:

- When the heater is switched off, i.e., when the automaton is either in the location **Off** or in the location **Idle**, the temperature decreases, as described by the differential equation $\dot{x} = -0.1x$, assuming $x > 0$.
- When the heater is switched on, i.e., the automaton is in the **Heating** location, the temperature increases, as described by the differential equation $\dot{x} = 8-0.1x$, assuming $x < 80$.
- The heater may suspend heating as soon as the temperature reaches 18 degrees and it must suspend it upon reaching 20 degrees.
- The heater may resume heating as soon as the temperature drops to 17 degrees and it must resume upon dropping to 15 degrees.

The formalization for hybrid automata is given in the following:

Definition 3 (Hybrid Automata (HA)). *A hybrid automaton H is a tuple $H = (L, l_0, X, A, E, F, I)$ where L is the set of locations, $l_0 \in L$ is the initial location, X is a set of symbols each representing a real-valued, continuous variable, A is the set of actions including the internal action ϵ, $E \subseteq L \times \mathcal{P}(X) \times A \times \mathcal{P}(X \cup X') \times L$ is the set of edges, $F : L \to \mathcal{P}(X \cup \dot{X})$ is a mapping assigning flows to locations, and $I : L \to \mathcal{P}(X)$ is a mapping assigning invariants to locations, where $\mathcal{P}(X)$ is a predicate over X, $\dot{X}$ is the set of dotted symbols from X representing the first derivatives of the variables and X' is the set of primed symbols from X representing conclusions of discrete change to the variables.*

Note that a flow defines how a variable valuation evolves, i.e., a flow describes a particular continuous change of a variable. In this hybrid automata definition a flow in a location is specified by the differential equation given by the predicate assigned to that location. Furthermore, the components of an edge (l, g, a, u, l'), also denoted by $l \xrightarrow{g,a,u} l'$, are g, the guard constraint, a, the action, and u, the update. Note that an update according to the definition is thus a predicate over $X \cup X'$. This predicate specifies how a variable valuation changes during a transition by defining how the post-transition valuations of the variables, the primed variable symbols, can be obtained from the pre-transition valuations, the unprimed variable symbols. The example automaton can then completely be defined by specifying

- $L = \{\, \text{Off}, \text{Heating}, \text{Idle} \,\}$
- $l_0 = \text{Off}$
- $X = \{\, x \,\}$
- $A = \{\, \text{SwitchOn}, \text{SwitchOff}, \epsilon \,\}$
- $E = \{\, (\text{Off}, \top, \text{SwitchOn}, x' = x, \text{Heating}), (\text{Heating}, \top, \text{SwitchOff}, x' = x, \text{Off}),$
  $(\text{Heating}, x \geq 18, \epsilon, x' = x, \text{Idle}), (\text{Idle}, x \leq 17, \epsilon, x' = x, \text{Heating}),$
  $(\text{Idle}, \top, \text{SwitchOff}, x' = x, \text{Off}) \,\}$
- $F : F(\text{Off}) = \dot{x} = -0.1x,\ F(\text{Heating}) = \dot{x} = 8 - 0.1x,\ F(\text{Idle}) = \dot{x} = -0.1x$
- $I : I(\text{Off}) = \top,\ I(\text{Heating}) = x \leq 20,\ I(\text{Idle}) = x \geq 15$

where $\top$ indicates a tautology. Analogously to timed automata, the state $s \in S$ of a hybrid automaton is defined by its location and its valuations of the continuous variables X, i.e., $s = (l, v)$, where $l \in L$ is the location and $v : X \to \mathbb{R}$ is the variable valuation function. Again denoting constraint satisfiability by $\models$ the semantics of a hybrid automaton is defined by two transition rules:

1. Action transition
$$(l, v) \overset{a}{\Rightarrow} (l', v')$$
if $\exists (l, g, a, u, l') \in E\, [v \models g \wedge v' \models I(l') \wedge u[v(x)/x]]$ where $u[v(x)/x]$ denotes the predicate obtained by replacing in u the occurrences of primed and unprimed variable symbols, $x \in X \cup X'$, by their valuations, $v(x)$.

2. Delay transition

$$(l, v) \overset{\delta}{\Rightarrow} (l, v')$$

if for all variables $x \in X$ there is a differentiable function $f : [0, \delta] \to \mathbb{R}$ with its first derivative $\dot{f} : [0, \delta] \to \mathbb{R}$ such that $f(0) = v(x)$ and $f(\delta) = v'(x)$, and $\forall \varepsilon \in [0, \delta]\, [f(\varepsilon) \models I(l) \wedge f(\varepsilon), \dot{f}(\varepsilon) \models F(l)]$, where $\delta \in \mathbb{R}^+$. The function f is called a witness for the transition $(l, v) \overset{\delta}{\Rightarrow} (l, v')$.

As in timed automata, action transitions are instantaneous and time only advances via delay transitions. If we assume an initial variable valuation function v_0 such that $\forall x \in X\, [v_0(x) = 0]$, an example state trace in the hybrid automaton from Fig. 3.3 is

$$(\text{Off}, x \mapsto 0) \overset{2.4}{\Longrightarrow} (\text{Off}, x \mapsto 0) \xrightarrow{\texttt{SwitchOn}} (\text{Heating}, x \mapsto 0) \overset{1.3}{\Longrightarrow} (\text{Heating}, x \mapsto 9.75\ldots)$$

with the witness $w_1(t) = 0$ for the first delay transition and $w_2(t) = 80(1 - e^{-0.1t})$ for the second delay transition.

Often the class of hybrid automata is reduced to the class of *linear hybrid automata* (LHA) as the restrictions imposed on linear hybrid automata, i.e., only permitting updates, invariants and flows that are conjunctions of linear predicates, make the handling of hybrid systems feasible. For example, model-checking algorithms for hybrid automata generally assume LHA.

Definition 4 (Linear Hybrid Automata (LHA))**.** *A hybrid automaton H is linear if all invariants of it are finite conjunctions of linear predicates over X, all its flows are finite conjunctions of linear predicates over $\dot{X}$, and all its updates are finite conjunctions of predicates over $X \cup X'$.*

Note that under these restrictions a linear differential equation may not be expressible. Thus, the notion of linearity of hybrid automata is not the same as linearity for differential equations.

3.2 Model Checking

Model checking is a verification technique that evaluates a formal model of a system with regards to previously specified properties. It yields guarantees on whether or not the properties are fulfilled by the model. In this dissertation I focus on model checking for real-time systems that are modeled with the timed automata formalization. There are two different algorithmic approaches to solving the model-checking problem, i.e., deciding whether $\mathcal{M} \models \Phi$ where $\mathcal{M}$ is a model and Φ is a property. Exhaustive model-checking approaches enumerate all states of the model until the satisfiability of Φ can be concluded. Symbolic model checking is such an exhaustive method and Subsection 3.2.1 presents the symbolic approach in detail. In contrast, the statistical model-checking approach is a non-exhaustive approach. It simulates the model behavior until enough confidence is gained to reason about Φ. The statistical technique is covered in Subsection 3.2.2.

3.2.1 Symbolic Model Checking

Symbolic model checking (MC) is an instance of those model-checking approaches that try to exhaustively enumerate all states of a model. Generally, symbolic model checking makes use of special data structures to represent states of the model such that many similar states can be processed at the same time. For example, instead of listing every possible valuation of a variable for certain applications it is sufficient to track the boundary values of the variable only. The intermediate values are analyzed implicitly. This aggregation of states leads to the notion of symbolic model checking: instead of exhaustively listing all possible values a symbol is introduced to represent many values simultaneously.

Algorithm 1 Naive algorithm for exhaustive reachability analysis [18]

```
1:  function REACHABILITY(s_0, S_f)
2:      W = { s_0 }
3:      P = ∅
4:      while W ≠ ∅ do
5:          s = s', s' ∈ W
6:          W = W \ { s }
7:          if s ∈ S_f then
8:              return true
9:          end if
10:         P = P ∪ { s }
11:         for all s' ∈ { s' | s ⇒ s' } do
12:             if s' ∉ P then
13:                 W = W ∪ { s' }
14:             end if
15:         end for
16:     end while
17:     return false
18: end function
```

A common model-checking analysis that can efficiently be performed with symbolic model-checking techniques is *reachability analysis*. Reachability analysis is used to verify the presence or absence of particular states in the model. Algorithm 1 shows a naive exhaustive forward reachability algorithm that makes no use of symbolic representation for states [18]. Its inputs are s_0, the initial state of the model, and a set of states S_f that the algorithm searches for. The algorithm maintains a set of waiting states W that still need to be checked and a set of states P that have already passed the check. As long as there are waiting states the algorithm takes one state at a time and checks if the state is in the target set S_f. If a state of the target set is found the algorithm returns success for the analysis, i.e., a state that was searched for is part of the model's state space. Otherwise the algorithm adds the checked state to the passed list, adds all its successor states to the waiting list, and continues the checks

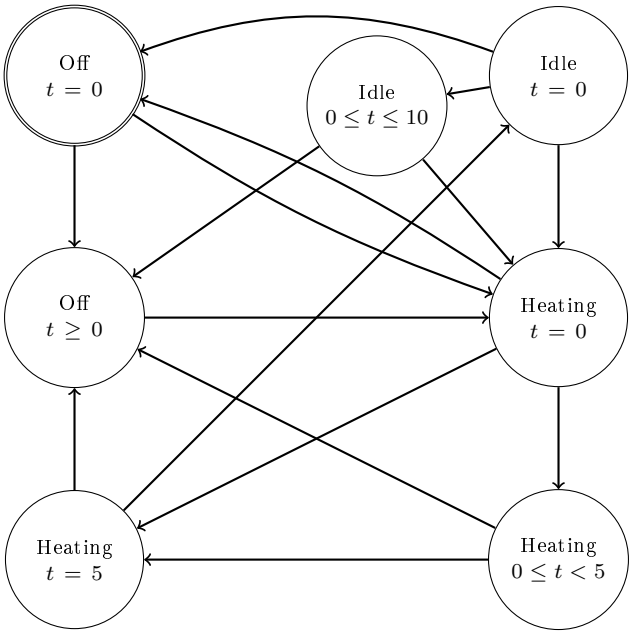


Figure 3.4: Zone graph of the heater example

with the next state from the waiting list. If the waiting list is empty the algorithm terminates and returns failure, i.e., no state that has been searched for is part of the model's state space.

If we apply this algorithm to the timed automata context the flaw of the naive approach becomes obvious: in timed automata a state is a pair (l, v) where l is the current location and $v : C \to \mathbb{R}_0^+$ is the current clock valuation function (see Subsection 3.1.1). Due to the real-valued nature of clocks the state space of a timed automaton is infinite and the model-checking approach in Alg. 1 will not work. The issue that the state space can not be explicitly enumerated can be solved for many model-checking problems, such as reachability analysis, by the construction of a *zone graph* as a symbolic state representation. The zone graph does not take particular clock values into account but aggregates them by defining zones of clock values with constraint expressions. For example, in the timed automata heater example (see Figure 3.2) in the **Heater** location only two clock zones are relevant for t: $t < 5$ and $t = 5$. Thus, values less than 5 can be aggregated to a single zone. Figure 3.4 shows the complete zone graph for the timed automaton of the heater example.

Unfortunately, zone graphs of timed automata may still be infinite, e.g., if there are two clocks and one of them is never reset. A normalization step like *k-normalization* reduces those infinite zone graphs to finite ones [19].[1] In k-normalization one aggregates all clock zones greater than the maximum constant in the timed automaton with the idea that the particular clock values are no longer of interest; only the fact that the values are greater than the biggest comparison value is relevant for the

[1] Note that k-normalization might be unsound for TA in certain cases.

model's behavior. The combination of the zone-graph approach with appropriate normalization guarantees termination of model-checking algorithms for timed automata. Algorithm 2 shows a reachability analysis based on zones [18]. As above the input is the initial state and a set of states to search for. This time, though, the search states (l, D) define the clock values to look for within an appropriate clock zone. Additionally, in line 12 an optimization is performed such that only zones that are not included in previously processed zones are checked.

Algorithm 2 Algorithm for reachability analysis with zones [18]

```
1:  function REACHABILITY(s_0, S_f)
2:      W = { s_0 }
3:      P = ∅
4:      while W ≠ ∅ do
5:          s = (l, D), (l, D) ∈ W
6:          W = W \ { s }
7:          for all (l_f, D_f) ∈ S_f do
8:              if l = l_f ∧ D ∩ D_f ≠ ∅ then
9:                  return true
10:             end if
11:         end for
12:         if ∀(l_p, D_p) ∈ P [l_p = l ⟹ D ⊈ D_p] then
13:             P = P ∪ { s }
14:             W = W ∪ { s' | s ⇒ s' }
15:         end if
16:     end while
17:     return false
18: end function
```

To efficiently obtain model-checking guarantees with timed automata it is necessary to process zones in an efficient manner. A common data structure for representing zones are *Difference Bound Matrices* (DBMs) [19]. Others are clock difference diagrams and its relatives [16]. In general, a zone for a set of clocks C is an expression given by the ⟨zone⟩ production in the grammar

$$\begin{aligned} \langle \text{comparator} \rangle &\models \leq \mid < \mid = \mid > \mid \geq \\ \langle \text{clockexpr} \rangle &\models c \mid c - c \\ \langle \text{clockterm} \rangle &\models \langle \text{clockexpr} \rangle \langle \text{comparator} \rangle n \\ \langle \text{zone} \rangle &\models \langle \text{zone} \rangle \wedge \langle \text{zone} \rangle \mid \langle \text{clockterm} \rangle \end{aligned}$$

where $c \in C$ is a clock symbol and $n \in \mathbb{N}$ is a natural number. By introducing an additional clock $\mathbf{0}$, the static zero clock, to the clock set C we obtain the extended clock set $C_0 = C \cup \{ \mathbf{0} \}$ that allows one to unify zone expressions. The grammar then

becomes

$$\begin{aligned} \langle \text{comparator} \rangle &\models < \mid \leq \\ \langle \text{clockexpr} \rangle &\models c \mid c - c \\ \langle \text{clockterm} \rangle &\models \langle \text{clockexpr} \rangle \, \langle \text{comparator} \rangle \, n \\ \langle \text{zone} \rangle &\models \langle \text{zone} \rangle \wedge \langle \text{zone} \rangle \mid \langle \text{clockterm} \rangle \end{aligned}$$

where $c \in C_0$ is a clock symbol and $n \in \mathbb{Z}$ is an integral number. This unification allows the specification of a zone by specifying the constraint values and operators $(n, \preceq)$ for every pair of clocks $x, y \in C_0$ if a constraint restricts that pair. A difference bound matrix stores these constraint value/operator pairs for all pairs of clocks. If a pair is not constrained the special symbol ∞ is used instead of a pair.

As an example, a clock constraint system with two clocks a and b and the constraints $a \in [2, 4)$, $b > 5$, and $b - a \geq 3$ is transformed to the canonical constraints $a - \mathbf{0} < 4$, $\mathbf{0} - a \leq -2$, $b - \mathbf{0} < \infty$, $\mathbf{0} - b < -5$, $a - b \leq -3$, and $b - a < \infty$. The resulting DBM is

$$\begin{array}{c|ccc} & \mathbf{0} & a & b \\ \hline \mathbf{0} & 0 & (-2, \leq) & (-5, <) \\ a & (4, <) & 0 & (-3, \leq) \\ b & \infty & \infty & 0 \end{array}$$

A difference bound matrix thus is a $|C_0| \times |C_0|$ matrix with entries from the set $\{ (n, \preceq) \mid n \in \mathbb{Z}, \preceq \in \{ <, \leq \} \} \cup \{ \infty \}$. The set of DBM entries is denoted by $\mathcal{K}$. An order on the entries is given by

$$\begin{aligned} (n, \preceq) &< \infty \\ (n_1, \preceq_1) &< (n_2, \preceq_2) \qquad \text{if } n_1 < n_2 \\ (n, <) &< (n, \leq) \end{aligned}$$

Furthermore, addition is defined by the following rules:

$$\begin{aligned} \infty + (n, \preceq) &= \infty \\ (m, \leq) + (n, \leq) &= (m + n, \leq) \\ (m, <) + (n, \preceq) &= (m + n, <) \end{aligned}$$

For the specification of verification properties for a model checker a formal logic is used. For timed systems the *Timed Computation Tree Logic* (TCTL) is a well-explored specification language. It was derived from *Computation Tree Logic* (CTL), which was originally developed for finite-state systems because it aimed at the verification of hardware [65]. CTL formulas Φ take the form given by

$$\langle \Phi \rangle \models \phi \mid \textbf{false} \mid \langle \Phi \rangle \rightarrow \langle \Phi \rangle \mid \exists \bigcirc \langle \Phi \rangle \mid \exists \, \langle \Phi \rangle \, \mathcal{U} \, \langle \Phi \rangle \mid \forall \, \langle \Phi \rangle \, \mathcal{U} \, \langle \Phi \rangle$$

where ϕ is an atomic proposition. These formulas allow the specification of particular execution paths, i.e., a sequence of states, in the system model. Their meaning is as follows:

$\exists \bigcirc \Phi$ There exists an execution path where the direct successor state satisfies Φ, i.e., if the current state is s then $\exists\, s' \in S\, [s \Rightarrow s' \wedge s' \models \Phi]$.

$\exists\, \Phi_1\, \mathcal{U}\, \Phi_2$ There exists an execution path where in some state Φ_2 holds and all previous states up to that state satisfy Φ_1, i.e., if the current state is s_0 then $\exists\, s_0 \Rightarrow s_1 \Rightarrow \ldots\, [\exists\, i\, [s_i \models \Phi_2 \wedge \forall\, 0 \leq j < i\, [s_j \models \Phi_1]]]$

$\forall\, \Phi_1\, \mathcal{U}\, \Phi_2$ For all execution paths there exists a state where Φ_2 holds and all previous states up to that state satisfy Φ_1, i.e., if the current state is s_0 then $\forall\, s_0 \Rightarrow s_1 \Rightarrow \ldots\, [\exists\, i\, [s_i \models \Phi_2 \wedge \forall\, 0 \leq j < i\, [s_j \models \Phi_1]]]$

As some classes of properties occur often in CTL the $\Box$ (box) and the $\Diamond$ (diamond) operator commonly are used to define convenient abbreviations for specific formulas. The resulting abbreviations are $\exists\Diamond\Phi$ for $\exists\, \mathbf{true}\, \mathcal{U}\, \Phi$, $\forall\Diamond\Phi$ for $\forall\, \mathbf{true}\, \mathcal{U}\, \Phi$, $\exists\Box\Phi$ for $\neg\forall\Diamond\neg\Phi$, and $\forall\Box\Phi$ for $\neg\exists\Diamond\neg\Phi$.

TCTL extends CTL by timing annotations to formulas as in CTL it is not possible to specify the length of a computation path: the formula $\exists\Diamond\Phi$ is satisfied if there is a path where Φ eventually becomes true. However, no information is encoded on when this event happens. TCTL annotates CTL operators with time bounds to restrict their scope [65]. TCTL formulas Φ take the form given by

$$\langle\Phi\rangle \quad \models \quad \phi \mid \mathbf{false} \mid \langle\Phi\rangle \rightarrow \langle\Phi\rangle \mid \exists\, \langle\Phi\rangle\, \mathcal{U}_{\sim c}\, \langle\Phi\rangle \mid \forall\, \langle\Phi\rangle\, \mathcal{U}_{\sim c}\, \langle\Phi\rangle$$

where ϕ again is an atomic proposition, $\sim$ is a relational operator from the set $\{<, \leq, =, \geq, >\}$, and c is a natural number ($c \in \mathbb{N}_0$). The modified meaning of $\exists\Phi_1\, \mathcal{U}_{\sim c}\, \Phi_2$ and $\forall\Phi_1 \mathcal{U}_{\sim c} \Phi_2$ is that the subscripts characterize the length of the path satisfying Φ_1 until Φ_2 becomes true, e.g., $\exists\Phi_1\mathcal{U}_{\leq 3}\Phi_2$ means that within 3 time units Φ_2 becomes true and beforehand Φ_1 was satisfied the whole time. The abbreviations from CTL carry over by annotating the diamond and box operators with equivalent timing constraints: $\Diamond_{\sim c}$ and $\Box_{\sim c}$.

The initially mentioned reachability analysis can then be formalized by a formula of the form $\exists\Diamond_{\geq 0}\Phi$ with the intention of searching for a path where Φ becomes satisfied eventually.

3.2.2 Statistical Model Checking

In contrast to symbolic model checking statistical model checking (SMC) does not explore the complete state space of the underlying model, or at least all required parts of it, to reason about the model's behavior. Instead, the SMC technique performs simulations of the model and uses statistical methods like *hypothesis testing* to estimate the probability of particular model behaviors. Thus, the model-checking problem that

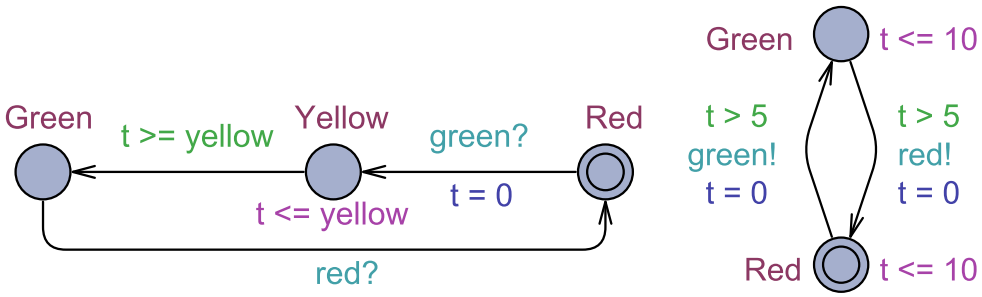
is solved by SMC no longer provides an absolute answer as in the symbolic model-checking approach with exhaustive enumeration. Instead of deciding whether $\mathcal{M} \models \Phi$ the statistical model-checking problem is $\mathcal{M} \models P_{\geq\theta}(\Phi)$: does the model $\mathcal{M}$ satisfy Φ with a probability greater or equal to θ?

In general, only properties over computation paths of finite length can be handled as one needs to obtain a final verdict on whether the simulated path satisfies the property in question upon finishing the simulation. SMC considers the simulation runs as individual Bernoulli experiments B_i. For every simulation run i we obtain a binary result b_i, where $b_i = 1$ if the run satisfies Φ and $b_i = 0$ otherwise. The Bernoulli experiment formulas $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$ then allow reasoning about p when examining the results b_i [68]. The hypothesis testing approach from statistics is the main approach for the reasoning about p. In hypothesis testing the two hypotheses $H : p \geq \theta$ and $K : p < \theta$ are evaluated and the probabilities of the two potential errors, accepting K though H holds (false negative) and accepting H though K is true (false positive), are estimated. These error estimates allow the estimation of an interval for p: $p \in [p_0 - \delta, p_0 + \delta]$. The size of this interval, i.e., the value of δ, obviously depends on the number of simulation runs N. Thus, in statistical model checking an input parameter to the technique is a confidence value. This value defines a lower bound for the probability to erroneously accept a hypothesis and implicitly defines how many simulation runs are executed for the probability estimation. As in statistical model checking not all states are explored but only a subset of available paths is computed SMC does not suffer from the state explosion problem that exhaustive model checking has.

3.3 UPPAAL

UPPAAL is a tool for modeling and verifying timed systems created jointly at Uppsala University and Aalborg University. The first version was already released in 1995. As its underlying modeling formalism UPPAAL uses an extended version of the timed automata formalism presented in Subsection 3.1.1. It thus provides more powerful modeling constructs although its logic is restricted. For one, in UPPAAL a model is not necessarily a single timed automaton. Instead, a model may include several automata that run concurrently and communicate via channels. Furthermore, data variables like bounded integers are available to, e.g., exchange information between automata. Up to now numerous improvements have been made to increase UPPAAL's performance and the number of its features. For example, version 4.0 introduced symmetry reduction [44], zone-based abstraction techniques [14], and user-defined functions (on data variables). Over the years, several case studies in academia and industry have used UPPAAL successfully, e.g., to prove properties of communication protocols [53]. For academic use UPPAAL is available free of charge at `http://www.uppaal.org`.

The UPPAAL tool is divided into two applications: the verification engine that may be used in a stand-alone manner to simulate and verify models, and a graphical



(a) Traffic light model (b) Operator model

Figure 3.5: Example model of a simple traffic light

user interface (GUI) that enables the user to easily create and edit models as well as a direct interface to the simulation and verification engine. The GUI has three core components: the editor, the simulator, and the verification component. The editor displays graph representations of the active model that may be edited by adding or deleting locations, inserting new automata, declaring data variables, modifying invariants, etc. The simulator permits stepwise simulation of the model displaying possible transitions and the current state. It was extended in version 4.1.18 such that in addition to the simulation of symbolic states with clock zones also concrete states with explicit clock values can be simulated. Lastly, the verification component allows the specification and verification of properties as well as configuring the model-checking engine. Also, if a verification attempt fails a diagnostic trace may be loaded into the simulation component to investigate the issue.

In the remaining part of this section, first, modeling in UPPAAL is presented in Subsection 3.3.1 and then a formalization of UPPAAL's extended timed automata formalism is given in Subsection 3.3.2. At last, Subsection 3.3.3 presents UPPAAL-SMC, UPPAAL's statistical model-checking extension.

3.3.1 Modeling in UPPAAL

UPPAAL allows easy modeling of concurrent systems by using parallel composition of automata [19]. Parallel composition allows internal transitions in individual components as well as synchronization transitions between two (binary synchronization) or more (broadcast synchronization) automata. Figure 3.5 displays a simple example model of a passive traffic light (Fig. 3.5a) and its operator model (Fig. 3.5b). The traffic light model has three locations, **Green**, **Yellow**, and **Red**. It accepts two synchronization events: a signal on the `green` channel when in the **Red** location, and a `red` signal when in the **Green** location. When a `green` signal is received the model stops at the intermediate location **Yellow** for a duration defined by the variable `yellow` before it transitions to the **Green** location. The switch from the **Green** location to the **Red** location is instantaneous. The operator model simply cycles between switching the traffic light to red and to green: every green or red phase is at least 5

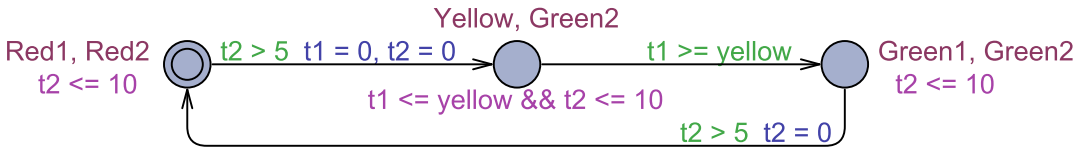


Figure 3.6: Product automaton of traffic light example

time units long (`t > 5`) and has a maximum length of 10 time units (`t <= 10`).

The parallel composition of the two automata can be unified to a single automaton: the *product automaton*. It captures all potential synchronizations and interactions between all participating automata. The product automaton of the traffic light example is depicted in Figure 3.6. In general, a product automaton of a complex model is a highly combinatorial data structure and therefore UPPAAL avoids computing it explicitly. Instead, its transitions are computed on-the-fly during simulation or verification by storing the states of the individual automata and performing explicit synchronization transitions instead of resolving them beforehand. Still, the explicit computation of a product automaton may come in handy for debugging purposes. For instance, closer examination of the example product automaton shows a potential flaw in the example model: if the variable `yellow` is set to a value greater than 10 a deadlock occurs; more accurately, a time-actionlock occurs as neither time nor action transitions may be performed after the model transitions to the **Yellow, Green2** location and the clock `t2` advances to 10.

In the following the modeling constructs UPPAAL provides are introduced:

Templates In UPPAAL all automata are instances of templates. This feature allows fast creation of models with many similar automata as a template does not only include the definition of the automaton layout, but it may also specify template parameters. Template parameters must be specified only when the system is instantiated and thus provide the user with some flexibility in automaton design. For example, the traffic light model (Fig. 3.5a) is defined in the `Trafficlight` template that specifies three parameters: `chan &green`, `chan &red`, and `int yellow`.[2] The traffic light `T1`, which switches to red when it receives a signal on the channel `channel2` and switches to green with a yellow time of 10 time units when it receives a signal on the channel `channel1` can then be instantiated with an instantiation expression

```
T1 = Trafficlight(channel1, channel2, 10);
```

and the resulting fully defined automaton `T1` may be used when declaring the complete system with a system declaration:

```
system T1, ...;
```

[2]Note that the `&` symbol specifies that the channels are passed by reference.

Synchronization As seen in the examples, UPPAAL uses channels for synchronization purposes. Edges send on a channel if they have an annotation declaring a sending synchronization (`name!`) and they receive signals if marked with a receiving annotation (`name?`). UPPAAL features two synchronization semantics. These are tied to two kinds of channels: *binary channels* and *broadcast channels*. Furthermore, UPPAAL allows channels to be declared as *urgent*. The implications of the different types are given in the following:

Binary channels Binary channels are declared by

```
chan name;
```

and they connect exactly two edges of two automata. Sending and receiving edges, both, may only fire if a matching and enabled receiving respectively sending edge is available. In other words, binary synchronization employs blocking semantics. If matching edges are present both edges fire synchronously and they transition to their target locations. Update annotations on the sending edge are processed before updates on the receiving edge.

Broadcast channels Broadcast channels are declared by

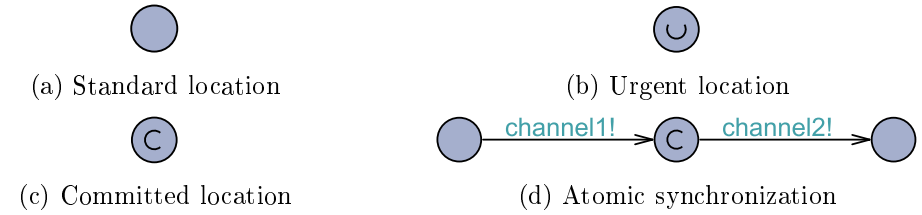
```
broadcast chan name;
```

and they allow the modeling of 1-to-N synchronization scenarios. An edge sending on a broadcast channel may fire any time it is enabled even when no receiving edge is available. Thus, it can be seen as a non-blocking synchronization action. If there are one or more matching and enabled receiving edges the broadcast synchronization results in all those edges transitioning together, in contrast to binary synchronization where only a single receiving edge can synchronize with the sending edge. Update annotations on the sending edge are again processed first. Then the updates on the receiving edges are processed in the order the automata are specified in the system declaration. For instance, if the system declaration is `system R1, R2, S;` and an edge in `S` sends on a broadcast channel and in both automata, `R1` and `R2`, a matching receiving edge synchronizes with the sending edge the update annotations are processed in the order `S - R1 - R2`.

Urgent channels Urgent channels are declared by

```
urgent chan name;
urgent broadcast chan name;
```

and they have the same synchronization semantics as standard binary and broadcast channels. However, the `urgent` declaration influences when the synchronization transition may be fired: if synchronization on an urgent channel is possible, i.e., for binary synchronization matching and enabled sending and receiving edges are available and for broadcast synchronization the sending edge is enabled, time may not advance



(a) Standard location (b) Urgent location

(c) Committed location (d) Atomic synchronization

Figure 3.7: Location types in UPPAAL

until the synchronization is performed. Effectively, urgent channels disallow delay transitions if synchronization via the channel is possible. Note that a location with an outgoing edge with a sending urgent broadcast synchronization effectively turns into a location where time may not pass, i.e., an urgent location (see below), because a sending broadcast annotation does not require a receiver. Unintentional zeno-timelocks, a state where action transitions but no delay transitions may be performed, may be the result.

Locations in UPPAAL also come in different flavors to ease the modeling task. In addition to the locations as defined in Def. 2 (Fig. 3.7a) UPPAAL provides *urgent locations* and *committed locations* as modeling constructs:

Urgent locations Urgent locations are marked with a `U` (Fig. 3.7b) and as the name implies time is not allowed to pass when an automaton is in such a location: leaving the location is urgent and may not be delayed. Marking a location urgent is equivalent to setting a new clock variable c to zero on all incoming edges and annotating the location with an invariant specifying that $c \leq 0$. Note that although time is not allowed to advance the model is not restricted from advancing differently, e.g., other edges may fire as long as they are enabled.

Committed locations Committed locations, which are marked with a `C` (Fig. 3.7c), expand on the construct of urgent locations. As in urgent locations delay transitions are prohibited. But additionally also all automata that are currently in committed locations must leave them before edges from non-committed locations may fire. Thus, committed locations allow the modeling of atomic transitions across multiple edges. This behavior is especially useful if synchronization on multiple channels is necessary as every edge may only synchronize on a single channel. For example, if an automaton needs to notify two automata via binary synchronization the intermediate location should be committed to prevent the execution of other automata while the synchronization is not complete yet (see Fig. 3.7d).

Data variables UPPAAL permits the user to employ data variables to ease the modeling process. Instead of having multiple similar locations a single location with a data variable may yield the same model behavior resulting in significantly reduced modeling effort. When employing symbolic model checking UPPAAL supports bounded

integers and boolean variables. In conjunction with the statistical model-checking extension UPPAAL-SMC (see Sec. 3.3.3) floating point variables are also supported. Variables are declared in C-like syntax:

```
int x = 5;
bool y;
int[3,5] z;
```

The declared variables are `x`, an integer of the default range [-32768,32767] (16bit) with an initial value of 5, `y` a boolean variable, and `z`, a bounded integer of the range [3,5]. Note that a `bool` declaration is an abbreviation for `int[0,1]`. Constant data values may be declared by prepending variables declarations with the `const` keyword.

3.3.2 Timed Automata in UPPAAL

When using UPPAAL's additional modeling constructs the formalization of timed automata needs to be extended accordingly:

Definition 5 (UPPAAL Timed Automaton (UTA)). *An UPPAAL timed automaton T is a tuple $T = (L, L_u, L_c, l_0, C, V, S, B, E, I)$ where*

- *L is the set of normal locations,*
- *L_u is the set of urgent locations,*
- *L_c is the set of committed locations,*
- *$l_0 \in L \cup L_u \cup L_c$ is the initial location,*
- *C is the set of clock variable symbols,*
- *V is the set of integer variable symbols,*
- *S is the set of binary synchronization channel symbols,*
- *B is the set of broadcast synchronization channel symbols,*
- *$E \subseteq L \cup L_u \cup L_c \times \mathcal{P}(C,V) \times \{\, \epsilon, s!, s? \mid s \in S \cup B \,\} \times \mathcal{P}(C,V) \times L \cup L_u \cup L_c$ is the set of edges, and*
- *$I : L \cup L_u \cup L_c \to \mathcal{P}(C,V)$ is the mapping assigning invariants to locations,*

where $\mathcal{P}(C,V)$ is a predicate over C and V.

Note that edges $(l, g, s, u, l') \in E$ may also be denoted by $l \xrightarrow{g,s,u} l'$ where g is the guard annotation, s is the synchronization label, and u is the update annotation. Unlike general timed automata, UTAs do not have a set of user-defined actions as in UPPAAL an edge can not be annotated with a name. If identification of an edge is required one can annotate the edge with a sending broadcast synchronization that uses an otherwise unused channel as a workaround. Parallel composition is handled by defining a network of timed automata:

Definition 6 (Network of UPPAAL Timed Automata (NUTA)). *A network of UPPAAL timed automata is a tuple* $N = (A_i, \ldots, A_n, U)$ *of* n *UPPAAL timed automata* $A_i = (L_i, L_{u,i}, L_{c,i}, l_{0,i}, C_i, V_i, S_i, B_i, E_i, I_i)$ *and a mapping* $U : \bigcup_i S_i \cup B_i \to \{0, 1\}$ *that assigns urgency to involved channels.*

Analogous to the UTA definition the following sets are also defined for NUTAs:

- $L = \bigcup_i L_i$, the set of all normal locations,
- $L_u = \bigcup_i L_{u,i}$, the set of all urgent locations,
- $L_c = \bigcup_i L_{c,i}$, the set of all committed locations,
- $C = \bigcup_i C_i$, the set of all clock variable symbols,
- $V = \bigcup_i V_i$, the set of all integer variable symbols,
- $S = \bigcup_i S_i$, the set of all binary synchronization channel symbols,
- $B = \bigcup_i B_i$, the set of all broadcast synchronization channel symbols, and
- $E = \bigcup_i E_i$, the set of all edges.

A location in such a network is represented by a location vector $\bar{l} = (l_1, \ldots, l_n), l_i \in L_i \cup L_{u,i} \cup L_{c,i}$ that contains the location of every component automaton. We use $\bar{l}[l'_i/l_i]$ to denote the location vector that is obtained by replacing the location l_i by l'_i in $\bar{l}$. Furthermore, we define the location function $P : \{ L \cup L_u \cup L_c \}^n \to L \cup L_u \cup L_c, (l_1, \ldots, l_n) \mapsto \{ l_i \}$ that maps a location vector $\bar{l}$ to the set of all locations contained in it and the invariant mapping $I : \{ L \cup L_u \cup L_c \}^n \to \mathcal{P}(C, V), (l_1, \ldots, l_n) \mapsto \bigwedge_i I_i(l_i)$, which assigns conjunctions of invariants to location vectors according to the component invariant mappings.

In analogy to the previously introduced timed automata the semantics of a network of UTA can be given by a labeled timed transition system (S, s_0, M, T) where S is the set of all states, s_0 is the initial state, $M = \{ \epsilon \} \cup \mathbb{R}^+$ is the set of transition labels, and $T \subseteq S \times M \times S$ is the set of transitions. A state $s \in S$ is defined by its location vector, its valuations of the clock variables C, and its valuations of the integer variables V. Thus, a state in the transition system can be represented by $s = (\bar{l}, v_c, v_v)$, where $\bar{l}$ is a vector of locations, $v_c : C \to \mathbb{R}_0^+$ is a clock valuation function, and $v_v : V \to \mathbb{Z}$ is a data variable valuation function. The initial state of the transition system is $s_0 = (\bar{l}_0, v_{c,0}, v_{l,0})$ where $\bar{l}_0 = (l_{0,1}, \ldots, l_{0,n})$, $v_{c,0} \equiv 0$, and $v_{v,0} \equiv 0$. The following rules then define the semantics of action transitions, $s \overset{\epsilon}{\Rightarrow} s'$, or $s \overset{c}{\Rightarrow} s'$, and delay transitions, $s \overset{\delta}{\Rightarrow} s', \delta \in \mathbb{R}^+$:

1. Internal action transition

$$(\bar{l}, v_c, v_v) \overset{\epsilon}{\Rightarrow} (\bar{l}[l'_i/l_i], u_i(v_c), u_i(v_v))$$

Applicable for any transition $l_i \xrightarrow{g_i, \epsilon, u_i} l'_i \in E_i$ such that $v_c \models g_i$, $v_v \models g_i$, $u_i(v_c) \models I(\bar{l}[l'_i/l_i])$, $u_i(v_v) \models I(\bar{l}[l'_i/l_i])$, and $P(\bar{l}) \cap L_c \neq \emptyset \implies l_i \in L_c$.

This rule defines an action transition without any synchronization.

2. Action transition with binary synchronization

$$(\bar{l}, v_c, v_v) \xRightarrow{c} (\bar{l}[l'_i/l_i, l'_j/l_j], u_j(u_i(v_c)), u_j(u_i(v_v)))$$

Applicable for any transitions $l_i \xrightarrow{g_i, c!, u_i} l'_i \in E_i$ and $l_j \xrightarrow{g_j, c?, u_j} l'_j \in E_j$ with $i \neq j$ such that $c \in S$, $v_c \models g_i \wedge g_j$, $v_v \models g_i \wedge g_j$, $u_j(u_i(v_c)) \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$, $u_j(u_i(v_v)) \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$, and $P(\bar{l}) \cap L_c \neq \emptyset \implies \{ l_i, l_j \} \cap L_c \neq \emptyset$.

This rule defines a binary synchronization transition. Two edges annotated with matching synchronization labels (`c!`/`c?`) in different components are fired simultaneously.

3. Action transition with broadcast synchronization without receiver

$$(\bar{l}, v_c, v_v) \xRightarrow{c} (\bar{l}[l'_i/l_i], u_i(v_c), u_i(v_v))$$

Applicable for any transition $l_i \xrightarrow{g_i, c!, u_i} l'_i \in E_i$ such that $c \in B$, $v_c \models g_i$, $v_v \models g_i$, $u_i(v_c) \models I(\bar{l}[l'_i/l_i])$, $u_i(v_v) \models I(\bar{l}[l'_i/l_i])$, and there is no transition $l_j \xrightarrow{g_j, c?, u_j} l'_j \in E_j$ with $i \neq j$ such that $v_c \models g_j$, $v_v \models g_j$, and $P(\bar{l}) \cap L_c \neq \emptyset \implies l_i \in L_c$.

This rule defines a broadcast synchronization transition without a receiving edge. An edge sending on a broadcast channel is free to fire even without receivers. If there are receivers the next rule applies.

4. Action transition with broadcast synchronization with receiver(s)

$$(\bar{l}, v_c, v_v) \xRightarrow{c} (\bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}], u_J(u_i(v_c)), u_J(u_i(v_v)))$$

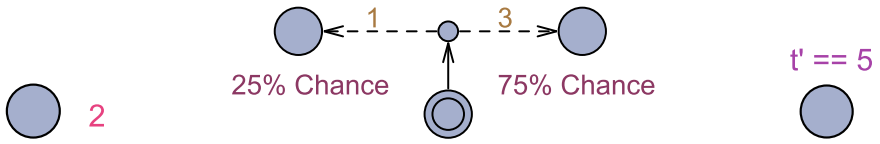
Applicable for any transition $l_i \xrightarrow{g_i, c!, u_i} l'_i \in E_i$ where $c \in B$ and $J \subseteq \{ 1..n \} \backslash \{ i \}$ is the maximal set of indices such that for any $j \in J$ there is a transition $l_j \xrightarrow{g_j, c?, u_j} l'_j \in E_j$, where $v_c \models g_i \wedge \bigwedge_{k \in J} g_k$, $v_v \models g_i \wedge \bigwedge_{k \in J} g_k$, $u_J(u_i(v_c)) \models I(\bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}])$, $u_J(u_i(v_v)) \models I(\bar{l}[l'_i/l_i, (l'_j/l_j)_{j \in J}])$, and $P(\bar{l}) \cap L_c \neq \emptyset \implies (\{ l_i \} \cup \{ l_j \mid j \in J \}) \cap L_c \neq \emptyset$.

This rule defines a broadcast synchronization transition with receiving edges. If an edge sending on a broadcast channel is fired it will synchronize with all available receiving edges.

Note that $\bar{l}[(l'_j/l_j)_{j \in J}]$ denotes the location vector that results from replacing l_j by l'_j for every index $j \in J$. Furthermore, u_J denotes the sequential execution of the updates $u_{j_0}, \ldots, u_{j_m}$ in the order given by the position of the individual automata instances in the system declaration; in other words, all receiving edges execute their updates in the same order as the automata of the edges are listed in the system declaration of the model.

5. Delay transition

$$(\bar{l}, v_c, v_v) \xRightarrow{\delta} (\bar{l}, v_c + \delta, v_v)$$



(a) Location with rate (b) Probabilistic splitting edge (c) Clock rate invariant

Figure 3.8: UPPAAL-SMC modeling constructs

where $(v_c + \delta)(x) = v_c(x) + \delta$.

Applicable for any $\delta \in \mathbb{R}^+$ such that $(v_c + \delta) \models I(\bar{l})$ and $P(\bar{l}) \cap L = P(\bar{l})$ if no action transition is possible with synchronization on a channel $c \in S \cup B\,[U(c) = 1]$.

This rule defines a global delay transition. All clocks advance by δ time units. Delaying is only possible if the current state does not include urgent or committed locations and no synchronization involving an urgent channel is possible. Note that this rule prevents individual automata from having unequal clock advances; all automata advance at the same rate.

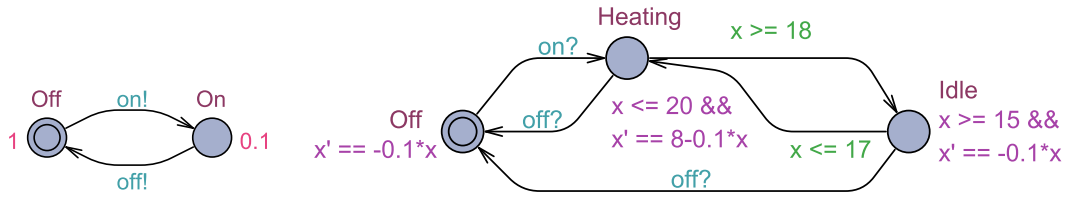
3.3.3 UPPAAL-SMC

UPPAAL-SMC is an extension for UPPAAL included in releases since version 4.1. It extends UPPAAL with statistical model-checking features as described in Section 3.2.2. UPPAAL-SMC uses the modeling formalism Networks of Priced Timed Automata (NPTA) for its statistical model-checking features. Priced timed automata are modified timed automata where clocks may advance with different rates in contrast to the synchronously advancing clocks in TAs. The expressiveness of such models thus increases and, in fact, it is possible to encode linear hybrid automata [27] with NPTAs, allowing UPPAAL-SMC to perform model checking of hybrid systems.

For the modeling of probabilistic systems UPPAAL-SMC introduces a few new constructs (Fig. 3.8):

Location Rates In case a location has no invariant assigned to it, i.e., the model may stay indefinitely at a location, UPPAAL-SMC requires the specification of an exponential rate for the location to specify the probability distribution for leaving the location spontaneously. Figure 3.8a shows a location with an exponential rate of 2, yielding a leaving probability after time t of $1 - e^{-2t}$.

Split Edges Split edges allow the modeling of probabilistic transitions, i.e., the target location depends on a non-deterministic choice. The splitting edges may be annotated with weights to define the probability for each target location. Figure 3.8b shows a split edge that transitions to the left location with a probability of 25% and to the right location with a probability of 75%



(a) User model (b) Heater model

Figure 3.9: Example hybrid model of a heater in UPPAAL-SMC

Clock Rate Invariants The specification of the clock rates is achieved by annotating locations with an invariant that sets the clock rate to an expression. Figure 3.8c shows a location where the clock `t` advances 5 time units for every global time unit. In contrast to the default clocks `t` runs 5 times as fast. Clock rate invariants can be combined with regular invariants and can also include more complex expressions (see linear hybrid automata in Subsection 3.1.2).

Floating Point Variables In addition to integer and boolean data variables, UPPAAL-SMC permits the usage of floating-point variables. In contrast to symbolic model checking, floating point variables do not cause an explosion in the number of states to check as statistical model checking assigns concrete values to variables during simulation and no symbolic representation of the current state is necessary.

As an example, Figure 3.9 shows the UPPAAL-SMC model of the heater hybrid automaton example from Subsection 3.1.2. The user model (Fig. 3.9a) operates the heater and the user is 10 times as likely to switch the heater on than off. The heater model (Fig. 3.9b) exactly implements the heater hybrid automaton from Figure 3.3. Note that the temperature is represented by the clock variable `x` and that the clock rate invariants impose the correct behavior of `x` in the different locations. An example simulation run is given in Figure 3.10. The temperature `Heater.x` is kept in the desired range of [15,20] as long as the heater is switched on. The drop in temperature at around 27 time units is a direct result from the heater being off as indicated by the `User.On` data series.

Properties in UPPAAL-SMC are specified using *Weighted Metric Temporal Logic* (WMTL$_{\leq}$). WMTL$_{\leq}$ formulas take the form given by the grammar

$$\langle\Phi\rangle \quad \models \quad \phi \mid \neg\langle\Phi\rangle \mid \langle\Phi\rangle \wedge \langle\Phi\rangle \mid \bigcirc\langle\Phi\rangle \mid \langle\Phi\rangle\, \mathcal{U}^{x}_{\leq d}\, \langle\Phi\rangle$$

where ϕ is an atomic proposition, d is a natural number, and x is a clock. In contrast to TCTL (see Subsection 3.2.1) WMTL$_{\leq}$ does not distinguish between different paths and thus a formula is evaluated using a concrete simulation trace $w = s_0 \Rightarrow s_1 \Rightarrow \ldots$. The formula $\bigcirc\Phi$, accordingly, is satisfied for a state s_i if Φ is satisfied in the concrete following state s_{i+1}. The until operator $\mathcal{U}^{x}_{\leq d}$ behaves similarly to the one in TCTL:

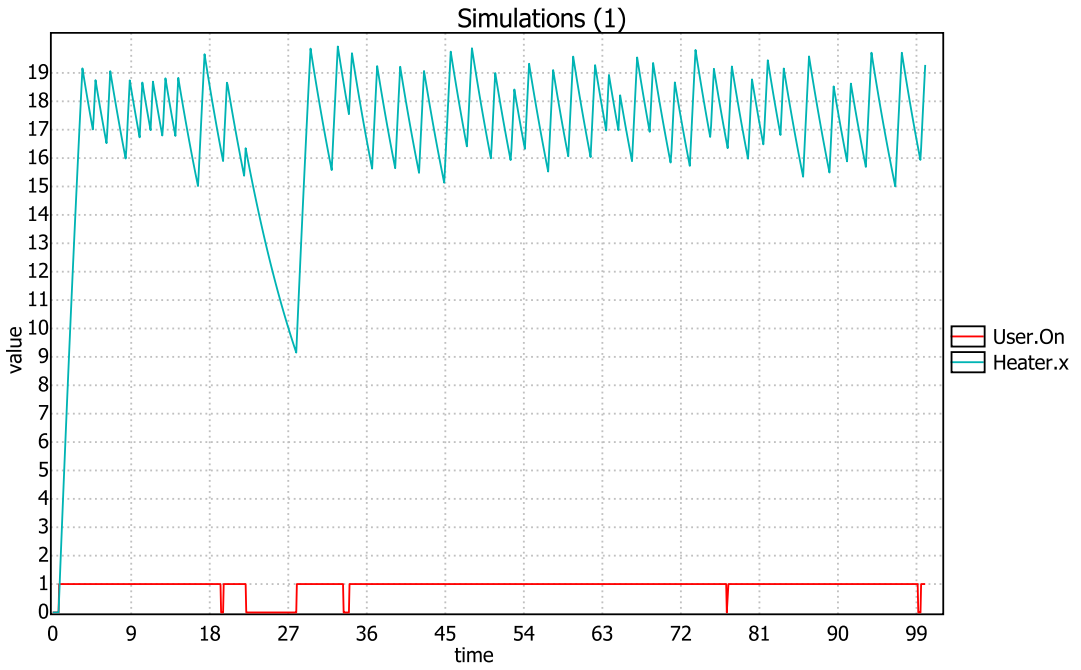


Figure 3.10: Simulation run of the heater example

the only difference is that here the specific clock to check (x) must also be specified and only $\leq$ is allowed as a comparator.

As in TCTL, the following abbreviations can be defined: $\Diamond^x_{\leq d}\,\phi = \mathbf{true}\,\mathcal{U}^x_{\leq d}\,\phi$ and $\Box^x_{\leq d}\,\phi = \neg\Diamond^x_{\leq d}\,\neg\phi$. Additionally given the probabilistic modeling constructs, we can define the probability for a property ϕ given a certain NPTA A: $Pr[A \models \phi]$ denotes the probability that a random run of A satisfies ϕ. For example, assume an automaton E with a non-deterministic uniform binary choice where only one choice satisfies ϕ. Then $Pr[E \models \phi]$ is 50%.

Hypothesis testing (see Subsection 3.2.2) in UPPAAL-SMC can then be carried out for an NTPA A by specifying formulas like

`Pr[c <= n](<> ` ϕ`) <=` p, or
`Pr[c <= n]([] ` ϕ`) <=` p

where the first formula is equivalent to $Pr[A \models \Diamond^c_{\leq n}\,\phi] \leq p$ and the second one is equivalent to $Pr[A \models \Box^c_{\leq n}\,\phi] \leq p$.

3.4 Online Model Checking

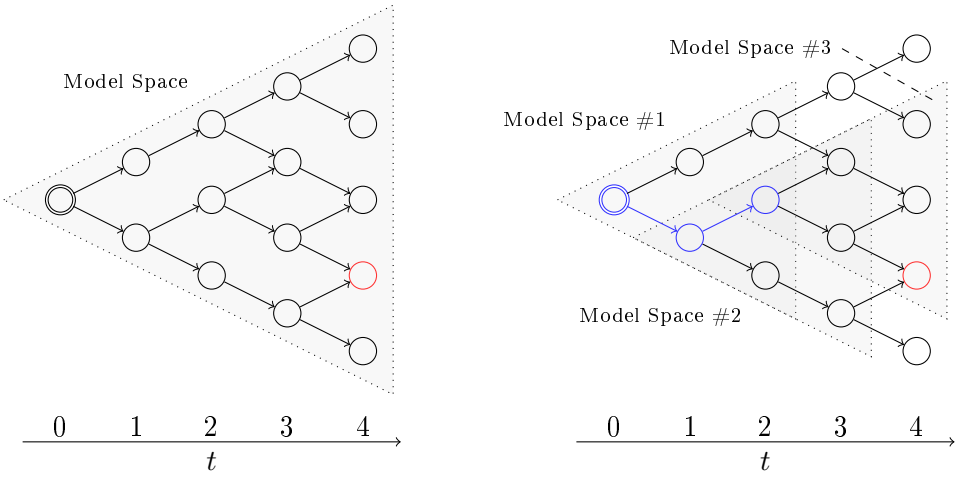
Model checking provides guarantees with respect to a model of a system. The dependency on a model has two disadvantages for model checking:

1. Complex models may suffer from *state space explosion*. Symbolic model checking systematically explores all relevant states of the model. In models where paths

split often the number of individual states rises to the extent that handling them in reasonable time becomes infeasible. Unfortunately, many real-world applications require complex models to capture the system behavior correctly. As a result often simplified models of the systems are employed and verified. Such a simplified model however may not cover all erroneous states and the guarantees given by model checking have to be taken with a grain of salt.

2. It may not be possible to obtain accurate models for complex real-world systems. The behavior of components may not be known in detail to accurately reflect all interactions of components with the rest of the system. For example, in the medical domain, systems often interact with patients and, thus, the patient should be part of the system model if one wants to guarantee their safety. Unfortunately, the behavior of the human physiology can not (yet) be modeled accurately to accommodate all potential reactions of the patient to stimuli by the system. For instance, think of the human breathing process where spontaneous coughing may arise or the breathing patterns may be significantly different for different patients because of injuries to the lungs, smoking, diseases, etc. When using such inadequate models for model checking the obtained safety guarantees for the patients may be invalid due to the gap between the model and the real world. In other words, if a positive guarantee is given for a model, but the model does not reflect the real world, all obtained guarantee must be invalidated as they can not be used for statements about the real-world system.

Online model checking is a variant of model checking that addresses those two problems of classic model checking. Instead of statically verifying the system once and for all during development the online model-checking approach employs a dynamic approach: the system is periodically verified for a limited time scope while the system is in operation. That way, the model checker produces guarantees that are only valid for a limited time frame. But those guarantees expire before they become invalid and can be trusted in the meantime. Limiting the scope of a guarantee resolves the problem of state explosion as only parts of the model, the parts that are relevant for the next time interval, are explored during verification, which can significantly reduce the number of states explored at a time. The problem of model inaccuracy can be addressed by adjusting the models that are verified to match the observed state of the real-world system. For example, if the model predicts a value this value can be compared to a real-world measurement and any differences can be resolved by adjusting the current model state. This state adjustment brings the model simulation back on track if predicted values were inaccurate. In the same way the model itself can be adjusted to more accurately reflect the current behavior of the system, e.g., prediction parameters could be derived from the real-world observations. Figure 3.11 contrasts the classic and the online model-checking approach. Figure 3.11a shows the classic model-checking approach. All states need to be incorporated in the state space of the model and the complete state space needs to be analyzed to obtain guarantees. In the online model-checking case, depicted in Figure 3.11b, there are three different model state spaces, which are analyzed in sequence. They all have a limited time scope of

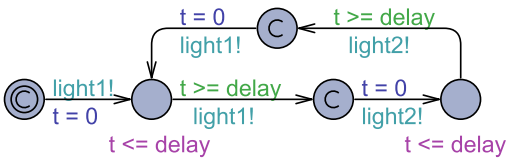


(a) Classic model checking (b) Online model checking

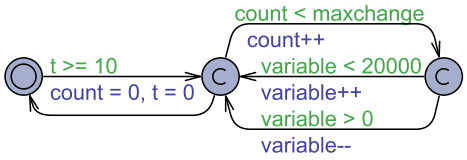
Figure 3.11: Classic model checking vs. online model checking

two time steps. If we now take the system's transitions in the real world into account (marked in blue) we can adjust the model to the current state and start a verification run from that state. That verification yields guarantees for the particular run of the system for the next two time steps. Thus, as long as the verifications are performed in such a pipeline fashion, i.e., the analyzed state spaces overlap in some manner, we always obtain a safe time frame. For example, if we assume the state marked in red is an unsafe state then the verification at time 2 fails, but the verification result at time 1 is still valid at time 3. Therefore, a safe time interval of 1 time units has been derived where measures can be taken to prevent the unsafe state. Note that the adaptation step can also change the model in such a way that it contains new states that a static model might not have covered.

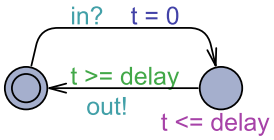
As an application example I now expand on the traffic light example introduced in Subsection 3.3.1. Assume a system with two traffic lights at a crossroad, a controller unit, and two communication channels that link the controller unit with the traffic lights. The controller unit is responsible for switching both traffic lights periodically from green to red or from red to green. Furthermore, assume that the communication channels have a transmission delay that may change non-deterministically but its maximum rate of change is bounded. This system can be modeled in UPPAAL using a traffic light model (Fig. 3.5a), a controller model (Fig. 3.12a), a channel model (Fig. 3.12c), and a delay-changing model (Fig. 3.12b) with the system declaration given in Figure 3.12d. The controller model initially instructs the first traffic light to switch to green because both traffic light models start in the red locations. It then periodically sends signals to both traffic lights according to the period given by `delay`. This value is the third parameter in the template instantiation in the system declaration and resolves to 100 time units. The channel model accepts a signal, then delays for a particular time and then issues a signal on the outgoing channel.



(a) Controller template



(b) Delay changer template



(c) Channel template

```
chan channel1, channel2, light1, light2;
int delay1 = 0, delay2 = 0;
Control = Controller(channel1, channel2, 100);
C1 = Channel(channel1, light1, delay1);
C2 = Channel(channel2, light2, delay2);
D1 = DelayChanger(delay1, 1);
D2 = DelayChanger(delay2, 1);
T1 = Trafficlight(light1, light1, 15);
T2 = Trafficlight(light2, light2, 15);
system Control, C1, C2, D1, D2, T1, T2;
```

(d) System declaration

Figure 3.12: UPPAAL model of traffic light example

Note that `delay` is passed in the template instantiations by reference and resolves to the variables `delay1` and `delay2`. The delay-changer model non-deterministically modifies the passed variable: at maximum every time unit the variable may change `maxchange` units. In the instantiation the delay variables are passed by reference and the maximum change is set to 1 unit per time unit.

An obvious requirement for such a traffic light system is that both lights should never show green simultaneously to prevent accidents. In terms of verification properties in UPPAAL this means that no state is reachable where the two traffic lights, `T1` and `T2`, are in their green locations. Formally, the property `E<>(T1.Green and T2.Green)` can be proved by UPPAAL, but unfortunately the result shows that such states do indeed exist: the delay of one traffic light may become so big that a signal instructing it to turn red is not processed before the other traffic light turns green. To obtain a safety guarantee with classic model checking, one would need to make further assumptions on the transmission delays. But such assumptions may not reflect the real-world behavior, yielding a discrepancy between the system model and the real system. The online model-checking approach can help here: instead of verifying the system with the time unbounded, we restrict the scope to the next 200 time units. The resulting property is `E<>((tau <= 200) and (T1.Green and T2.Green))`. This property is satisfied in the model because the change rate of the transmission delay prevents the delay from reaching a critical level within 200 time units. It follows that a suitable approach to operating the traffic light system safely would be to verify the system every 100 time units for a 200 time unit scope while updating the delays of the channels to the observed values in the real world before verification. This procedure yields reliable

guarantees such that not both of the traffic lights show green simultaneously: if at some point the verification fails there is still a safe time window of 100 time units where safe operation is guaranteed and safety measures can be taken. For example, the controller could immediately send instructions to both traffic lights to show red until observation of the real-world delays shows that safe operation can be continued.

Note that while online model checking expands the class of systems that can be evaluated with model checking there are still requirements that a particular application must fulfill such that online model checking can be employed. The requirements are as follows:

1. It must be possible to generate a system model (using UPPAAL's time automata formalism) that is accurate for a limited time scope.

2. The model must be flexible enough to be adjusted to a changing real-world system state. Note that there are multiple ways to adjust a model (see Chapter 6).

3. Crucial parameters of the real-world system must be observable such that meaningful model adaptation can be performed.

4. The time consumed by the verification and by the model adjustment steps must be limited as OMC is a real-time system. If the deadlines for the verification are exceeded failure of the verification must be assumed to guarantee safe operation and thus potentially unnecessary countermeasures are performed.

Summarizing this chapter, the modeling constructs of finite, timed, and hybrid automata were introduced and formalized and model checking as a way to obtain guarantees for such models was presented. Furthermore, two problems of model checking, the state space explosion and the dependency on accurate models, were discussed and the online model-checking approach was explained as a dynamic process that may solve these issues. Moreover, the model checker UPPAAL was presented and a formal specification of its extended timed automata formalism was given. With the theoretic foundation covered, we can now continue to the next chapter where a preliminary case study on laser tracheotomy is presented, which was conducted to evaluate the feasibility of online model checking with UPPAAL.

4 Laser Tracheotomy – A Preliminary Case Study

In this chapter I present a preliminary case study on online model checking with UPPAAL. This case study has been carried out to evaluate the feasibility of OMC with UPPAAL such that an educated decision could be made on whether online model checking with UPPAAL should further be explored in the context of this dissertation.

Evaluating whether UPPAAL's performance in practice meets the real-time requirements imposed by online model checking is crucial to evaluate its potential. In this case study we encode in UPPAAL the models of a previous online model checking case study [70], which models a laser tracheotomy surgery with hybrid models for the model checker PHAVer. This reimplementation enables us to compare the results to the previous work and lets us focus on UPPAAL's performance and suitability. We carry out the online model-checking process with our derived models using a prototype implementation for automatic online model checking with UPPAAL. The accuracy of the parameter prediction and the run-time performance is compared to the results of the original case study. As a general result UPPAAL is suitable for use in an online model-checking context in practice. Slightly more detailed, the results show that the relative errors of blood oxygen (SpO_2) estimations were on average about 2%, which is slightly worse than the original case study. For performance, a verification step took on average about 50ms, which is a significant improvement over the hybrid models in PHAVer. Parts of this work have been carried out by Xintao Ma and have already been published [74, 75].

The rest of the chapter is organized as follows: Section 4.1 presents the case study components and shows the encoded UPPAAL models. Section 4.2 provides the experiment results and an evaluation of those. At last, Section 4.3 summarizes the chapter and discusses the results with regards to this dissertation.

4.1 A Medical Case Study

In this section the medical context and the modeling for the laser tracheotomy case study are presented. Subsection 4.1.1 introduces laser tracheotomy and provides related safety requirements that are checked in the case study. The concrete system of UPPAAL models that was derived from the case study by Li et al. [70] is the focus of Subsection 4.1.2.

4.1.1 Laser Tracheotomy

Tracheotomy is a surgery performed on patients that have problems breathing through their nose or mouth, e.g., when the tongue muscle falls back and blocks the air flow while sleeping. During the surgery a direct access to the windpipe of the patient is created, usually from the front side of the neck. Laser tracheotomy refers to the kind

of tracheotomy where the access to the windpipe is created using a laser scalpel, a medical device capable of cutting tissue with focused light. Using laser for the cut has several benefits, including greater precision and a reduction of blood loss since blood vessels are closed immediately. However, during tracheotomy the laser also poses the threat of tissue burns in case the oxygen concentration in the windpipe of the patient is too high.

In the original case study the goal is to ensure that the laser may only be triggered when its operation is safe, i.e., it will not cause harm to the patient. Additionally, as the patient is ventilated during the surgery, it is necessary to ensure that the blood oxygen of the patient does not drop to dangerously low levels, because ventilation must be suspended during the cutting process. Lastly, for convenience of the surgeon, an additional requirement is that once the use of the laser is approved the laser should be safe to emit for a minimum amount of time such that the cut is not interrupted unnecessarily. The verification properties for UPPAAL-SMC (see Subsection 3.3.3) are given in Weighted Metric Temporal Logic ($\text{WMTL}_{\leq}$, see Subsection 3.3.3, [27]):

- Oxygen concentration (O_2) above threshold (Th_{0_2}) while laser emits
  - `Pr[<=100](<> O2 > Th_O2 && LaserScalpel.LaserEmitting)`
- Blood oxygen (SpO_2) below threshold (Th_{SpO_2}) while laser emits
  - `Pr[<=100](<> SpO2 < Th_SpO2 && LaserScalpel.LaserEmitting)`
- Laser stops emitting early
  - ```
    Pr[<=100](<> (O2 > Th_O2 || SpO2 < Th_SpO2) &&
                t_appr < Th_appr && LaserAppr == true)
    ```

These properties characterize states that should be unreachable in a well-controlled system. Thus, the expected probability is zero for each of them.[1]

4.1.2 System Modeling

The laser tracheotomy scenario described by Li et al. has four different communicating components [70]:

- **Patient** The patient model represents the patient currently under surgery. It is characterized by its current windpipe (O_2) and blood (SpO_2) oxygen levels.
- **Ventilator** The ventilator model represents the ventilation device that regulates the patient's breathing rate during the surgery. It is characterized by the current position of the pressure cylinder, i.e., whether or not the pump air reservoir is currently empty, full, or somewhere in between.

[1] The probability is never calculated as zero in a statistical model-checking approach but approximated in an interval (see Subsection 3.2.2).
    ```

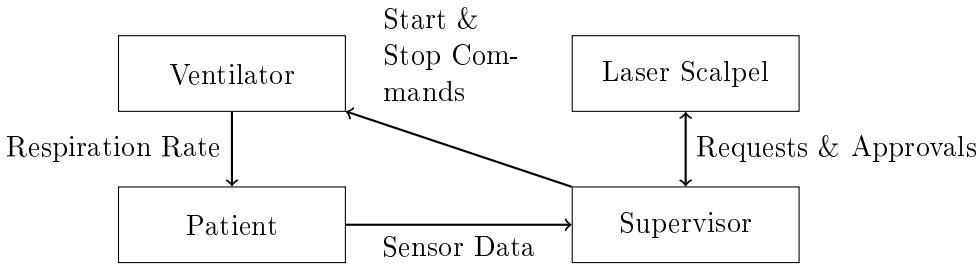


Figure 4.1: Laser tracheotomy system [70]

- **Laser Scalpel** The laser scalpel model represents the physical laser scalpel device used to cut the opening to the windpipe. It is characterized by its current operation state, whether or not the surgeon currently wants to operate the laser, and if such usage is allowed.
- **Supervisor** The supervisor model represents the controlling unit. It is responsible for ensuring the safety requirements of the system as given in Subsection 4.1.1. The supervisor approves the usage of the laser scalpel if safe, and controls the ventilator accordingly.

Figure 4.1 shows the connections of the system components with their respective communication data. The ventilator regulates the respiration rate of the patient. The physiological signals of the patient are measured by sensors and forwarded to the supervisor. The supervisor analyzes the values and either approves the usage of the laser scalpel that was requested previously and consequently stops the ventilator, or, otherwise, usage is prohibited and the ventilator continues normal operation. Additionally, when an approved cut is finished the supervisor instructs the ventilator to continue operation.

The UPPAAL models of the components for online model checking are now discussed in more detail. All of the models were derived from the original hybrid models using the encoding for hybrid automata from Subsection 3.3.3. The main difficulty in the transformation of the models is representing the continuous variables O_2 and SpO_2 in the hybrid models using clock variables in UPPAAL-SMC and ensuring correct system behavior using synchronization. We use clock variables as they allow the specification of continuous behavior in locations while data variables can only model discrete behavior. The remaining parts are straightforward because the graph components and transition constraints carry over directly as they are based on the same basic finite state machine formalism. The models allow online adaptation of the O_2 and SpO_2 parameters of the patient. Those values can be obtained by standard measurement devices like pulse oximeters. All of the models therefore have an initialization sequence for this model adjustment that enables the online model-checking approach.

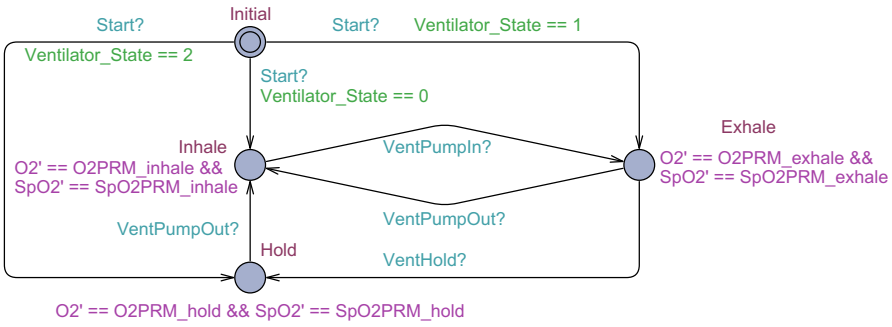


Figure 4.2: Patient UPPAAL model

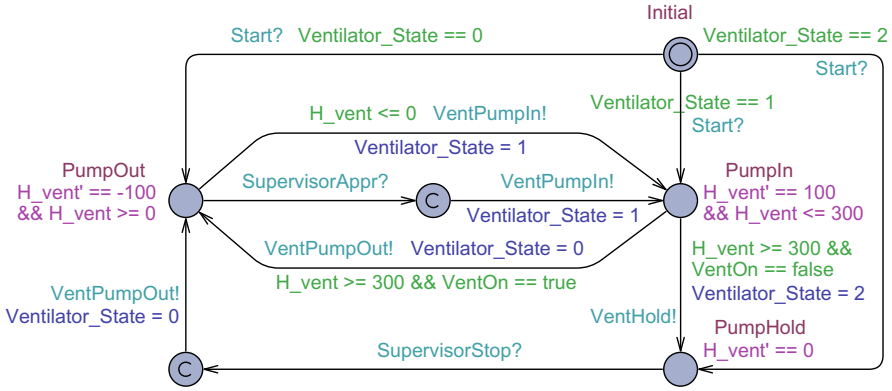


Figure 4.3: Ventilator UPPAAL model

Patient

The patient model is depicted in Figure 4.2. It has three locations plus the additional initialization location. Their modeling function is as follows:

- **Inhale** The patient is inhaling with assistance of the ventilator. The derivatives of O_2 and SpO_2, $\dot{O}_2$ and $\dot{SpO}_2$, are set such that they represent the inhalation process.
- **Exhale** The patient is exhaling with assistance of the ventilator. The derivatives $\dot{O}_2$ and $\dot{SpO}_2$ are set such that they represent the exhalation process.
- **Hold** The patient is exhaling without the assistance of the ventilator. The derivatives $\dot{O}_2$ and $\dot{SpO}_2$ are set such that they represent the exhalation process.

The patient switches between locations when it receives synchronization events from the ventilator model. The respective communication channels are `VentPumpIn`, `VentPumpOut`, and `VentHold`, which pass the state of the ventilator model on to the patient.

Ventilator

The ventilator model has three functional locations, two technical location, and the required initialization location (Fig. 4.3).

- **PumpOut** The ventilator is pumping air out of its pressure cylinder. The derivative of the position of the pressure cylinder is therefore set to a negative value (-100).
- **PumpIn** The ventilator is pumping air into its pressure cylinder. The derivative of the position of the pressure cylinder is thus set to a positive value (100).
- **PumpHold** The ventilator is not pumping any air at all. The derivative of the position of the pressure cylinder is therefore set to zero.

The two technical locations are committed locations as indicated by the `C` letters. They are used to chain communication events together. Here, the model reacts to the input signals from the supervisor model as follows:

- `SupervisorAppr` The supervisor approves usage of the laser. The ventilator changes to the **PumpIn** location and sends the `VentPumpIn` signal with the intent to switch to the **PumpHold** location, as soon as the pressure cylinder is completely filled with air.
- `SupervisorStop` The supervisor restricts usage of the laser. The ventilator changes to the **PumpOut** location and sends the `VentPumpOut` signal to continue ventilation of the patient.

Note that the data variables in the conditions are modified by the supervisor model.

Laser Scalpel

The laser scalpel model represents the interaction between the surgeon and the laser scalpel. It has four locations plus the initialization location (Fig. 4.4).

- **LaserIdle** The laser is idle. It accepts a usage request from the surgeon via the synchronization channel `SurgeonReq`.
- **LaserRequesting** The surgeon intents to use the laser. The approval of the usage by the supervisor model is pending. The request can either be approved (`SupervisorAppr`) or canceled by the surgeon (`SurgeonCancel`).
- **LaserEmitting** The laser is cutting. The invariants ensure that the laser may only fire up to a maximum duration. The emitting of the laser may also be stopped by the supervisor (`SupervisorStop`) or by the surgeon (`SurgeonStop`).
- **LaserCancelling** The laser stopped operation without the influence of the supervisor model. The supervisor model resynchronizes with the laser scalpel model and switches back to the **LaserIdle** location.

Note that all instructions from the surgeon may occur at any time.

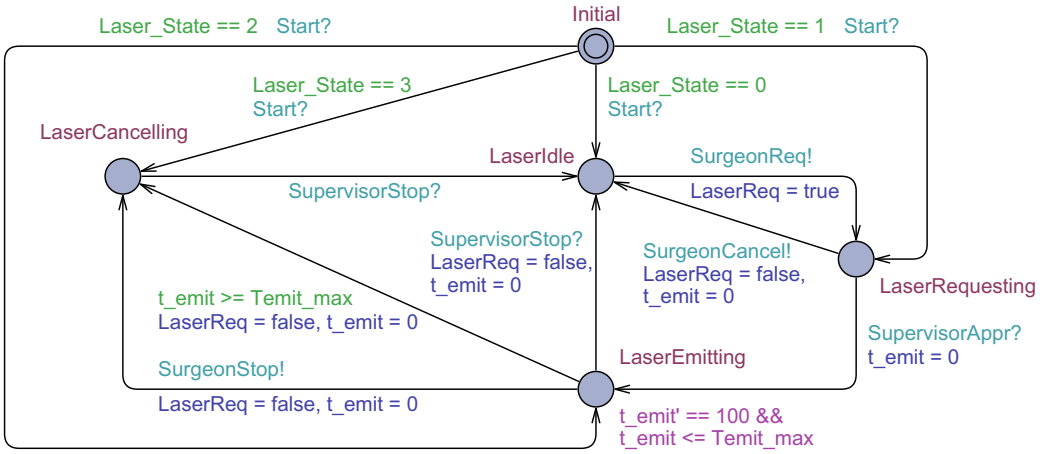


Figure 4.4: Laser scalpel UPPAAL model

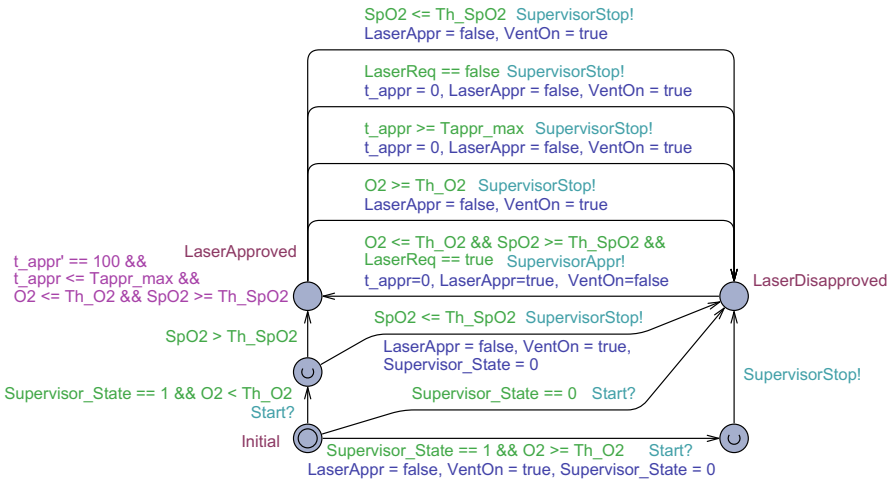


Figure 4.5: Supervisor UPPAAL model

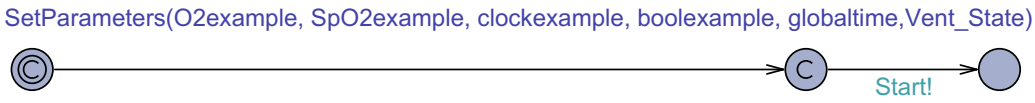


Figure 4.6: Initialization UPPAAL model

Table 4.1: PhysioNet databases and patient IDs

	Database	#1	#2	#3	#4	#5	#6
O_2 (CO_2)	MGF/MF	mgh077	mgh077	mgh089	mgh057	mgh019	mgh110
SpO_2	MIMIC v2	a45463	a45436n	439n	n10301n	a45611n	477n

Supervisor

The supervisor model checks if the physiological parameters of the patient are within safe boundaries and approves the laser usage (Fig. 4.5). It has two locations for this purpose: **LaserApproved** and **LaserDisapproved**. If any of the safety requirements gets violated the supervisor revokes a previous approval. Note that the initialization part here also needs to check if a safety requirement was violated. This check is necessary because the O_2 and SpO_2 values may change when the model is adapted, which might invalidate a previous approval.

Initialization

Lastly, the initialization model has the purpose of initializing all constants that may have been adapted to real-world values during model adaptation (Fig. 4.6). Using broadcast synchronization, a starting transition guarantees a common starting point for the whole system.

4.2 Experiments and Evaluation

To evaluate the online model-checking approach with UPPAAL we carried out several experiments with the models. The necessary real-world patient data for the adaptation steps was extracted from the PhysioNet database, an open medical database offering a large collection of recordings of medical signals of various kind.[2] Six different patient traces were assembled and used as a basis. Every patient trace was executed ten times yielding 60 experiments in total. Table 4.1 shows the PhysioNet databases and the patient IDs of the data used. More information on the data can be found in the original thesis on this topic [74]. All experiments ran the system for 600 seconds where every three seconds a model adaptation and verification step was performed. Thus, the workflow of every three-second cycle is as follows: first we adjust the O_2 and SpO_2 values in the model to the observed values. Then we try to verify the system properties for the next six seconds. Lastly, we evaluate the verification results such that if a property was not verified we derive that in three seconds at the earliest an unsafe system state is reached and thus emergency measures should be taken beforehand. Note that if the models predict the short-term behavior of O_2 and SpO_2 correctly and the supervisor strategy is effective such an emergency can not arise. For the adjustment of the models to the real patient's windpipe oxygen and blood oxygen values a linear regression approach with a sliding window was used

[2] http://www.physionet.org

Table 4.2: Relative errors of O_2 and SpO_2 estimation

[%]	#1	#2	#3	#4	#5	#6
Min SpO_2	0	1.43	0.52	2.28	0.48	0.59
Max SpO_2	6.01	1.62	0.63	2.99	0.59	4.18
Avg SpO_2	1.59	1.54	0.60	2.82	0.54	2.27
Min O_2	0.6	18.0	12.3	16.8	11.3	8.3
Max O_2	66.0	23.2	14.0	20.5	15.3	10.5
Avg O_2	21.7	20.8	13.4	18.9	12.3	9.2

to estimate the model parameters. The history window for the linear regression to obtain these parameters from the real O_2 and SpO_2 values was 30 seconds in all cases. The confidence level for the statistical model checker was set to 99%. We evaluated three aspects of the approach: first we checked whether the safety requirements given in Subsection 4.1.1 are violated for any patient trace. Then we compared the relative errors of our O_2 and SpO_2 predictions to the values in the reference paper [70]. Lastly, we evaluated the execution times with a focus on the real-time requirements for online model checking.

The first result is straightforward: during all experiments all three safety properties were satisfied at all times with the confidence level of 99%. Thus, our models seem to be accurate enough to predict the physiological parameters of the patient for a time bound of three seconds. Moreover, the supervisor strategy implemented in the models proves to be effective at preventing accidental tissue burns resulting from triggering the laser at inappropriate times.

Table 4.2 shows the relative errors of our parameter estimation. The SpO_2 estimates are very consistent and in general show a relative error of about 2%. These results are accurate enough to guarantee the safety of the patient with regards to the blood oxygen. In contrast, the estimation of windpipe oxygen is not that precise, with an average relative error of about 16%. However, due to the supervisor strategy the safety of the patient is still guaranteed as the supervisor disallows any usage of the laser scalpel if the patient's safety is at risk. As a result, the scalpel may not be used during a significant amount of time. A more sophisticated prediction strategy than linear regression is likely to yield better prediction results, which enable the supervisor to approve the use of the laser more often reducing the time where the scalpel may not be used. Compared to the results of the original case study our SpO_2 results are slightly less accurate but still useful for safety statements. As the original case study does not specify exactly which patient traces were used as an experiment basis differences in the results may simply stem from the selection of different traces. For the O_2 results Li et al. provide no relative error results.

Table 4.3 shows the execution times of an adaptation step of the models and the following verification of the safety properties. Our experiments were carried out on a Macbook Pro 2.66 GHz with 4GB memory using iOS 10.6.8. In the experiments our approach took in the worst case 320 milliseconds for a cycle while in the original

Table 4.3: Model checking execution times

[s]	Minimum	Maximum	Average
UPPAAL-SMC	0.033	0.32	0.047
PHAVer	0.571	1.445	0.727

case study nearly 1.5 seconds elapsed. Unfortunately, the original case study does not specify the hardware used. In spite of that, the approach using simulation of timed automata in UPPAAL-SMC for verification performs significantly better than the symbolic verification of hybrid automata in PHAVer. Thus, we assume the speedup can not be attributed only to differences in hardware, especially because our hardware is not on the top end. Thus, using UPPAAL-SMC provides a performance advantage in practice. Looking at the absolute values in this case study where the hard real-time constraints were three seconds for one verification cycle, we generally observed execution times of about 10% of the real-time deadlines. It therefore seems feasible to use UPPAAL-SMC for the implementation of online model checking.

4.3 Conclusion and Discussion

This chapter presented a medical case study on laser tracheotomy using UPPAAL-SMC and used it to evaluate online model checking in practice. The online model-checking approach periodically adjusts the underlying system model to real-world values and analyzes the new models, e.g., for patient safety issues. The case study showed that this approach is capable of providing reliable safety guarantees even if the patient's physiological behavior is modeled only roughly using a simple linear regression approach when parameters are continuously adapted to the real-world values. Although this study identifies online model checking as a useful technique to reduce false positives, further research is necessary to support this claim. Accordingly, this dissertation continues exploring online model checking with UPPAAL in the following chapters. In particular, I provide a unified approach for the development of OMC applications that includes an interface for automatic model adaptation to ease the development of systems that should be monitored with online model checking. This automatic adaptation interface synthesizes necessary means to adapt a model from a classic model and executes the online model-checking procedure to allow seamless simulation and verification of the system in question. The next chapter presents the developed framework, which is one of the main contributions of this dissertation.

5 An Online Model-Checking Framework

In the context of this dissertation a framework has been developed that allows the creation of online model-checking applications in conjunction with the UPPAAL model-checking tool. With the framework I provide a unified approach to the development of online model-checking applications as several common problems need to be solved across all OMC applications (see the preliminary case study in Chapter 4). Instead of solving these problems for each individual case I provide a general solution that yields a simple workflow for the application development. One problem that is solved is how the state space adaptation to the real world is handled and the framework includes an algorithmic approach that allows users to specify such adaptations without altering the original UPPAAL model. The benefit is that previously developed UPPAAL models can be reused and employed in an online model-checking context in rapid fashion even when the user has no in-depth knowledge of the OMC approach. Furthermore, all OMC applications observe the real-world system to determine differences between the real system and the system model such that the model can be adapted. To support this task a general processing pipeline is included in the framework to allow simple data acquisition, processing, and validation. Note that in contrast to the strict definition of a framework in object-oriented programming, which requires deriving from classes, in this dissertation a framework only denotes a development environment that provides useful components to the user that enable the creation of an application. No specifics are given on how the framework components must be used. The framework is implemented in Java as the UPPAAL tool suite supplies an application programming interface (API) in the form of Java classes and thus integration with UPPAAL is relatively easy. The framework is available for download at `http://www.tuhh.de/sts/research/projects/online-model-checking.html`.

In the remaining part of this chapter the framework is presented in detail and its development context is laid out. Section 5.1 discusses the goals of the framework. Section 5.2 then presents the architecture of the framework and introduces its components. Lastly, Section 5.3 demonstrates the usage of the framework by developing an example OMC application.

5.1 Goals

The online model-checking framework was designed and developed with four goals in mind. This section briefly discusses those goals and motivates them.

The first goal was to provide a development environment where experience with model checking with UPPAAL carries over to developing online model-checking applications. Thus, the complete design of the verification model is still done using UPPAAL's model editor and no special constructs need to be introduced to the model to gain the benefits of online verification. A key feature that results from this goal is

that previously developed UPPAAL models can be reused in the framework mostly unmodified. No special modification of the model is necessary to make it work with the online model-checking framework.

The second goal was to provide an automatic way to online model checking with the framework. After creating the system model in UPPAAL and specifying the system interactions using the framework classes the verification procedure runs on its own and informs the user if the system is about to become unsafe. The usability of the framework is also a key concern for this goal insofar that the specification of the system should be intuitive. This requirement induces that most of the automation specifics of the framework should be hidden from the user and simple parameters should be sufficient for the specification of an application. For example, defining how data is gathered from the real system and how the model is adapted to this data should be straightforward for the user and the implementation details should be handled by the framework to keep the procedure of developing online model checking close to classical model checking.

The third goal was to provide a visualization of the online model-checking process as most of the actions performed happen behind the scenes and debugging one's application may be difficult. Showing the adapted models, the current simulation traces, the variable valuations, the verification results, and potential intermediate components for data acquisition and processing makes the online model-checking process transparent and provides a solid foundation for future research, e.g., in the context of model adaptation techniques (see Chapter 6).

Lastly, the fourth goal was to ensure results given by the framework are *sound*, i.e., every guarantee deduced from the system model and the real-time parameters observed is a logical consequence of the model semantics. A key observation here is that the online model-checking approach requires real-time processing and that the complete system thus forms a real-time system with soft or hard deadlines depending on the particular application. Special attention needs to be given to the model adjustment and verification steps in the real-time context, especially when automation is desired for the adaptation.

5.2 Architecture

The framework can be divided into three parts: the data acquisition and processing component, the simulation and verification engine, and the visualization component. Figure 5.1 and Figure 5.2 show class diagrams of the main classes of these parts, the complete framework with the implementation classes consists of 116 classes with about 11500 lines of code. The data acquisition and processing classes provide functionality to extract information from the real-world system. This part employs a pipeline system for data processing. The classes of this component are presented in Subsection 5.2.1. The simulator and verification engine is responsible for running the UPPAAL model of an application in real time, providing means to adapt the model according to data gained by data acquisition, and executing required verifications

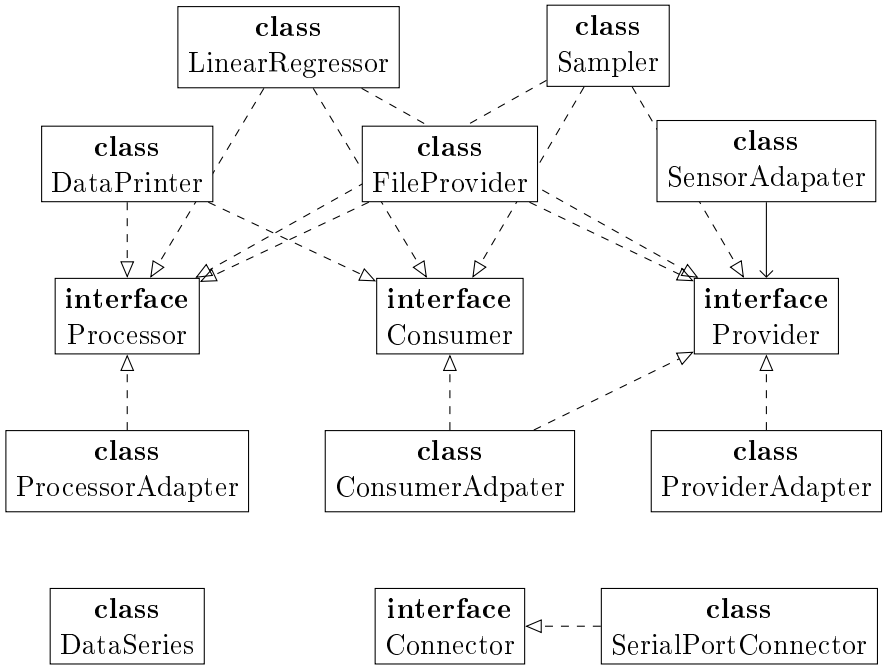


Figure 5.1: Class diagram of the main data acquisition and processing classes

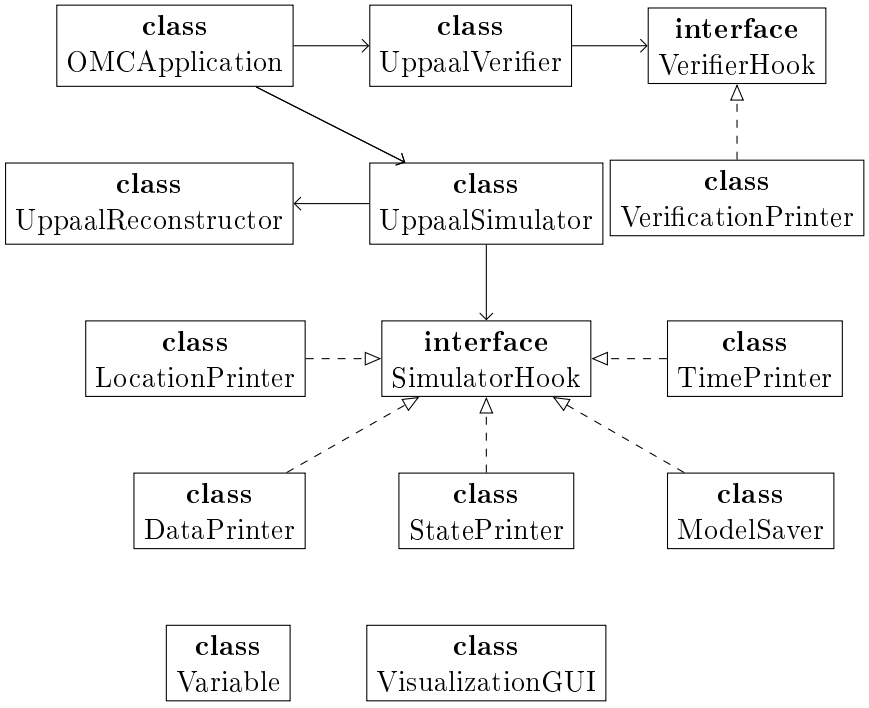


Figure 5.2: Class diagram of the main simulation and verification classes

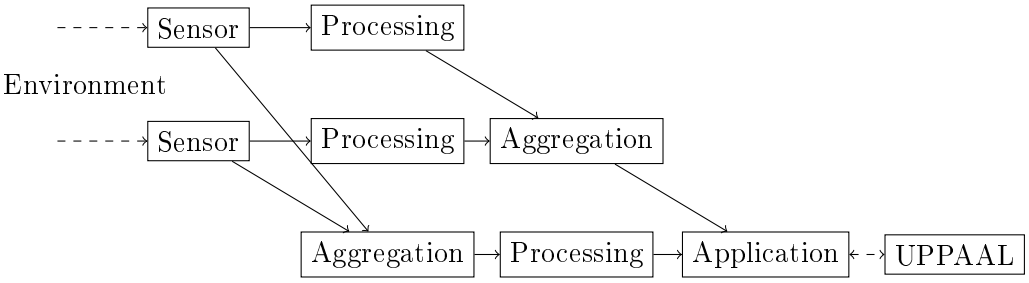


Figure 5.3: Example data flow of data acquisition and processing

to extend guarantees from the current model state. This part of the framework is presented in Subsection 5.2.2. The visualization component is a stand-alone graphical user interface (GUI) application that permits loading of online model-checking applications for the purpose of understanding the data flow, model simulation, and verification results. The GUI was developed by Axel Neuser as part of his Bachelor thesis [80], which was supervised by me. The GUI application is presented in Subsection 5.2.3.

5.2.1 Data Acquisition and Processing

The data acquisition and processing module uses a pipeline architecture. Figure 5.3 shows an example processing pipeline. On the left, data is gathered from the environment using sensor components. Then the data is forwarded to processing or aggregation components. Afterwards a second processing and aggregation stage takes place before the resulting data is passed to the online model-checking application. The OMC application then uses the processed information to evaluate system properties using the UPPAAL model checker.

In the framework all components in the pipeline and the OMC application itself are autonomous objects. They all must implement the `Processor` interface, which just requires that a component can be started and stopped, and that its current running state can be accessed. For convenience a working implementation of the `Processor` interface is provided with the `ProcessorAdapter` class such that new components can reuse its functionality to implement the `Processor` interface. The `ProcessorAdapter` implementation executes a task periodically after an initial delay has passed. The task to run is specified by implementing the Java `Runnable` interface and calling `setProcess(Runnable)` with the implementing object on the `ProcessorAdapter`. The initial delay and the period for the execution are specified by calling `setDelay(long)` and `setPeriod(long)` respectively where the parameter specifies the desired time in nanoseconds. Note that those times may be changed while the process is running. Changing the period allows the specification of tasks with aperiodic scheduling if required by an application as the changes take effect during the next task execution. Changing the delay however has only an effect when the component is stopped and started again.

The data that is acquired and processed is stored in `DataSeries` objects. A `DataSeries` object has a specific length n and it contains the last n values added to it. It thus provides a history of the values of length n. For convenience the `DataSeries` class calculates statistics about the contained values; the calculation is done on-the-fly: the variance and the mean are accessible using `getVariance()` and `getMean()`. The `DataSeries` objects are created by classes implementing the `Provider` interface. This interface defines that a class provides data series that may be consumed. The `DataSeries` objects are accessed by `getDataSeries(int)` where the parameter defines which data series to access. The number of data series provided by a `Provider` object can be obtained by calling `getDataSeriesCount()`. A description of the data series is also available (`getDataSeriesDescription(int)`). Furthermore, a `Provider` object may also be locked (`lock()`) and unlocked (`release()`), which is necessary when accessing multiple data series of it synchronously. Otherwise, a data series may update while reading another one resulting in inconsistent data. The adapter class, `ProviderAdapter`, for the `Provider` interface implements the `Provider` functionality for reuse in new components. To use its locking functionality one synchronizes on the `getLock()` object and then uses the `waitForLock()` function in the `Runnable` object of the `ProcessorAdapter` object to ensure that it does not update a data series while the `Provider` object should be locked. See Algorithm 7 later in this chapter for an example of the locking process. For components that require data series as input the `Consumer` interface is available. It only specifies the function `addDataSeries(Provider, int)`, which can be called to register `DataSeries` objects of other `Provider` instances for consumption. For this interface also a matching adapter class, `ConsumerAdapter`, exists. The adapter just stores the registered data series information and lets the `Consumer` object access the registered `DataSeries` as if only one `Provider` object would provide them.

The framework comes with four ready-to-use components that may be used to construct a data acquisition and processing pipeline.

- `DataPrinter` The `DataPrinter` object helps during development of the data pipeline by periodically printing the registered `DataSeries` objects to the console.

- `FileProvider` The `FileProvider` object constructs `DataSeries` objects from a column separated input file (csv file). It provides a single `DataSeries` object for every column. The timing can be configured such that the object either periodically reads a new data record or one column is interpreted as a sequence of timestamp values. The timestamp variant is useful to play back previously recorded sensor data in real time to emulate a real sensor while testing the system.

- `LinearRegressor` The `LinearRegressor` object periodically estimates parameters of a linear function, which approximates the input data. The first registered `DataSeries` object is assumed to contain the x-values, while the second `DataSeries` object contains the y-values. The output data series are the es-

timation timestamp, the alpha value, and the beta value where the estimated function is $y = \alpha \cdot x + \beta$.

- `Sampler` The `Sampler` object periodically samples the registered `DataSeries` objects. This may be necessary if one wants to synchronize multiple `Provider` objects or if one needs to reduce the update rate of a certain `Provider`.

To provide better understanding of the individual parts required to construct a processing component the `LinearRegressor` class is now broken down. Algorithm 3 shows the constructor of the class. The `ProcessorAdapter` variable `processor` is configured to run with the arguments passed, the `ProviderAdapter` variable `provider` is created to accommodate the three `DataSeries` objects with the provided length, and an `ConsumerAdapter` object is created to store the input data that is passed by the user later on. The implementation of the three required interfaces `Processor`, `Provider`,

Algorithm 3 Constructor of `LinearRegressor` class

```
1  public LinearRegressor(long delay, long period, int length) {
2
3      processor = new ProcessorAdapter();
4      processor.setProcess(process);
5      processor.setPeriod(period);
6      processor.setDelay(delay);
7
8      provider = new ProviderAdapter(3, length);
9      provider.setDataSeriesDescription(0, "LR: Timestamp");
10     provider.setDataSeriesDescription(1, "LR: Alpha");
11     provider.setDataSeriesDescription(2, "LR: Beta");
12
13     consumer = new ConsumerAdapter();
14 }
```

and `Consumer` is straightforward. All calls are forwarded to their respective adapter objects. Although the implementation is simple Algorithm 4, Algorithm 5, and Algorithm 6 show the source code for the completeness of the example. Note that error checking has been omitted for brevity. The remaining part is the definition of the Runnable object `process` such that it actually calculates the linear regression of the given input values. Algorithm 7 shows the task definition. In lines 6 and 8 the synchronization takes place: only if updates of the output data series are allowed the task is permitted to proceed. Then, in lines 10 and 14 the input data is locked so that both data series are synchronized. With the consistent data the linear regression is then calculated and the resulting values are added to the output `DataSeries` objects.

As the last component for the data acquisition and processing pipeline the framework supplies the `Connector` interface. This interface is meant to be used when implementing interfaces to real-world sensors where a connection needs to be established

Algorithm 4 `Processor` implementation in the `LinearRegressor` class

```
1  @Override
2  public void start() {
3     processor.start();
4  }
5
6  @Override
7  public void stop() {
8     processor.stop();
9  }
10
11 @Override
12 public boolean isProcessing() {
13    return processor.isProcessing();
14 }
```

Algorithm 5 `Provider` implementation in the `LinearRegressor` class

```
1  @Override
2  public int getDataSeriesCount() {
3     return provider.getDataSeriesCount();
4  }
5
6  @Override
7  public DataSeries getDataSeries(int index) {
8     return new DataSeries(provider.getDataSeries(index));
9  }
10
11 @Override
12 public String getDataSeriesDescription(int index) {
13    return provider.getDataSeriesDescription(index);
14 }
15
16 @Override
17 public void lock() {
18    provider.lock();
19 }
20
21 @Override
22 public void release() {
23    provider.release();
24 }
```

Algorithm 6 `Consumer` implementation in the `LinearRegressor` class

```
@Override
public void addDataSeries(Provider provider, int index) {
   consumer.addDataSeries(provider, index);
}
```

Algorithm 7 Periodic task implementation in the `LinearRegressor` class

```
private Runnable process = new Runnable() {

   @Override
   public void run() {

      synchronized (provider.getLock()) { // Acquire lock

         provider.waitForLock(); // Ensure updates are allowed

         consumer.lock(); // Lock providers
         DataSeries x = consumer.getDataSeries(0);
         DataSeries y = consumer.getDataSeries(1);
         double created = System.nanoTime() / 1000000.0;
         consumer.release(); // Release providers

         double xvals[] = x.getValues();
         double yvals[] = y.getValues();
         double xmean = x.getMean();
         double ymean = y.getMean();
         double xx = 0, xy = 0;

         for (int i = 0; i < x.getCount(); ++i) {
            xx += (xvals[i] - xmean) * (xvals[i] - xmean);
            xy += (xvals[i] - xmean) * (yvals[i] - ymean);
         }

         double alpha = xy / xx;

         provider.getDataSeries(0).addValue(created);
         provider.getDataSeries(1).addValue(alpha);
         provider.getDataSeries(2).addValue(ymean - alpha * xmean);
      }
   }
};
```

before the interface may be used. It defines two connection functions, `connect()` and `disconnect()`, and a status query function, `isConnected()`. Currently, the framework supports sensors connected via the serial port, either in hardware or a virtual one. Users may use the `SerialPortConnector` adapter class to rapidly define a sensor that communicates via the serial port. Connecting to a sensor via bluetooth is also possible. The `POSensor` class, which is part of the `Heartrate` example, a demonstration OMC application included in the framework, connects to a sensor via bluetooth. However, the bluetooth connection functionality has not yet been generalized into, e.g., a `BlueToothConnector` class.

The data acquisition and processing pipeline is constructed in the actual online model-checking application displayed on the right in Figure 5.3. This component is responsible for receiving the pipeline data, adjusting the UPPAAL model to these observation, and querying the verification engine for guarantees about the system. For the convenience of the application developer the `OMCApplication` class is supplied, which serves as a base class. An OMC application component is a `Processor` like the other processing units. In the periodic task the application receives the data from the data pipeline, schedules changes for the UPPAAL model, reconstructs the model to incorporate the scheduled changes, and then verifies properties with the updated model. During the construction process of the `OMCApplication` object the object assembles the data pipeline, initializes the simulation and verification engine, and registers data series providers for their visualization in the GUI. For registering components for visualization the class provides the `SensorAdapter` variable `sensors`. A call to `addSensor(Provider)` makes the passed `Provider` object available in the visualization GUI.

5.2.2 Simulation and Verification

For the interactions with the UPPAAL tool the framework provides three classes that may be used by an OMC application: the `UppaalSimulator` class, the `Variable` class, and the `UppaalVerifier` class.

The `UppaalSimulator` class is responsible for the real-time simulation of the UPPAAL model and the modifications to it. The simulator class allows loading and saving of UPPAAL model files using `load(String)` and `save(String)`. When a model has been modified it will be saved including the initialization sequence that was synthesized by the framework to perform the modifications. The real-time simulation can be started, stopped, and paused by calling the respective methods (`start()`, `stop()`, and `pause()`). Note that before starting the simulation a model must be loaded, the first initialization sequence for the model must be synthesized by triggering an initial reconstruction with `reconstruct()`, and the real-world duration of a time unit in the UPPAAL model must be specified. Setting this duration is accomplished by calling `setRealtime(long)` with the duration in milliseconds. The constants `UppaalSimulator.SECONDS` and `UppaalSimulator.MILLISECONDS` can be used to increase readability of the source code. A reconstruction of the model is simply triggered by calling `reconstruct()` on the simulator. During a model reconstruction

the following steps are performed:

1. If the simulation is running the simulation is paused to ensure the consistency of processed models.
2. The current simulation state is extracted from the simulation.
3. A transition sequence to the extracted state is calculated. Reduction methods may be applied to obtain an appropriate sequence that meets the real-time deadlines of OMC.
4. The transition sequence is modified such that scheduled state modifications are carried out when the sequence is processed.
5. A new model is synthesized with an initialization path derived from the adjusted transition sequence.
6. The new model is loaded into the simulator and its state is advanced until the simulator leaves the initialization sequence.
7. If the simulation has paused initially the simulation resumes.

During the initial reconstruction, thus, no functional initialization path, i.e., a path that recreates a transition sequence, is synthesized because no transitions have been executed during simulation yet. However, all means necessary for adjustments are added to the model such that future reconstructions can modify the model. To schedule a modification for the model during the next reconstruction the function `scheduleDataChange(Variable)` is provided by the framework. The function lets the user set a data variable of the UPPAAL model to any value in its data range. For example, if the model has a data variable declared by `int[0,5] count;` this variable could be set to any value between 0 and 5.

To refer to a variable the framework uses the `Variable` class. `Variable` objects take the name of the variable in the UPPAAL model as a constructor parameter and their values can be changed by calling `setValue(int)` on them. Note that variables local to an automaton need to be prefixed with the instance name of the template. For example, if the previously mentioned `count` variable was declared in a template called `T1` one can refer to this variable by instantiating a matching `Variable` object with `new Variable("T1.count")`. For more information on model reconstruction and its adaptation see Chapter 6.

To check properties of the system from the current simulation state for a bounded future the `UppaalVerifier` class is used. An instance can be obtained by querying the `UppaalSimulator` with `getVerifier()`. Before properties can be checked the scope of the verification needs to be specified, i.e., how far into the future the model should be checked. This is done by calling `setScope(long)` with the desired number of model time units. Then, properties can be checked with symbolic model checking by calling one of the `verifiy...` methods. Alternatively, statistical model checking can be employed with `estimateProbability(String, double[])`. The available functionality is as follows

Method	Behavior
`verifyPossibly(String)`	Verifies $\exists\Diamond_{\leq s}\Phi$.
`verifyEventually(String)`	Verifies $\forall\Diamond_{\leq s}\Phi$.
`verifyPotentiallyAlways(String)`	Verifies $\exists\Box_{\leq s}\Phi$.
`verifyInvariantly(String)`	Verifies $\forall\Box_{\leq s}\Phi$.
`estimateProbability(String, double[])`	Verifies $Pr[A \models \Phi]$ with $\Phi = \Diamond_{\leq s}\Theta$ or $\Box_{\leq s}\Theta$.

where s is the scope and the properties Φ are passed to the functions as the string.[1] Note that for the `verify...` methods the returned boolean value is the final verification result. For probability estimation the estimated interval is returned in the passed `double` array.

Both the simulator and the verifier classes allow the registration of hook classes. These classes have specific functions that are called when certain events occur within the framework. When implementing a hook class the user therefore can gather information or even react to events. The simulator allows the registration of objects that implement the `SimulatorHook` interface. When registered the object is informed after the initialization of the simulator, before and after a model reconstruction, and before and after a transition in the model. The object can then react to this input. Some ready-to-use simulator hook objects are supplied by the framework. Table 5.1 summarizes those classes. The verifier accepts objects implementing the `VerifierHook` interface. The registered object then is informed when a verification or estimation is started, when the process finishes, and when an error occurs. A built-in `VerifierHook` object is the `VerificationPrinter` object that just prints verification results to the console.

5.2.3 Online Model-Checking Visualization

A graphical user interface (GUI) has been developed under my supervision in a Bachelor thesis by Axel Neuser [80] for the visualization of the online model-checking process.

[1] See Subsection 3.2.1 and Subsection 3.3.3 for more information on properties.

Table 5.1: Simulator hook classes of the framework

Class	Functionality
`LocationPrinter`	After every transition prints the current locations to the console.
`DataPrinter`	After every transition prints the current data variable valuations to the console.
`TimePrinter`	After every transition prints the current time constraints to the console.
`StatePrinter`	After every transition prints all of the above.
`ModelSaver`	After every model reconstruction saves the reconstructed model to a specified file.

The main window of the visualization application can be seen in Figure 5.4. It is divided into six regions. The control interface is located in the top left corner. Here, OMC applications can be loaded, the simulation can be started, or raw commands can be sent directly to the simulator engine. The main view in the center shows the current model state. In Figure reffigure:gui:main one instance of a template is visualized. Each instance is divided into two parts: on the left side the actual model is depicted as constructed in UPPAAL. The current locations are marked with a red circle. On the right side the initialization sequence is shown that was synthesized by the simulator during reconstruction. Valuations of data variables of the model can be seen in the region in the middle on the left side. Here, one can observe and validate that updates to data variables during reconstruction are performed correctly. Below the variable section is the sensor section. `DataSeries` objects maintained by components of the data acquisition and processing pipeline that have been added in the OMC application are listed here and a click on them opens a plot of them. These plots are updated in real time such that the processing pipeline can be evaluated. Figure 5.5 shows an example plot of a `DataSeries` object. The console of the application is located at the bottom of the window. Here the user is informed about verification attempts and their results. Lastly, for the user's convenience the most recent verification result is displayed in the top right corner. Here is also an indicator located that shows how long the obtained guarantee remains valid.

5.3 Usage

To demonstrate the usage of the framework we now develop an example online model-checking application for the traffic light example from Section 3.4.

In a very first step when developing an OMC application one must think about what the model adjustment step should modify to remedy the differences between a system model and the real system. In the case of the traffic light example differences between the model and the real system may occur when the modeled transmission delay does not match the real delay. Thus, in the online model-checking system we need to adapt this delay to the observed one. In the UPPAAL models from Section 3.4 the delays are encoded as simple data variables, which allows the OMC process to simply change the delay to the correct values with the framework.

Next, one needs to decide how the correct values should be obtained such that they can be set. For demonstration purposes our traffic light example uses a file that contains a sequence of delays for both traffic lights. This file provides our OMC application directly with `DataSeries` objects that contain the necessary real-world delays.

In a next step one must think about the verification intervals and deadlines within the OMC application. The UPPAAL models specify that the traffic lights change every 100 time units. If one assumes time units of $\frac{1}{10}$ seconds the verification should at least include two changes. Therefore, the verification horizon is 20 seconds. This results in a periodic verification interval of 10 seconds as the verification guarantees

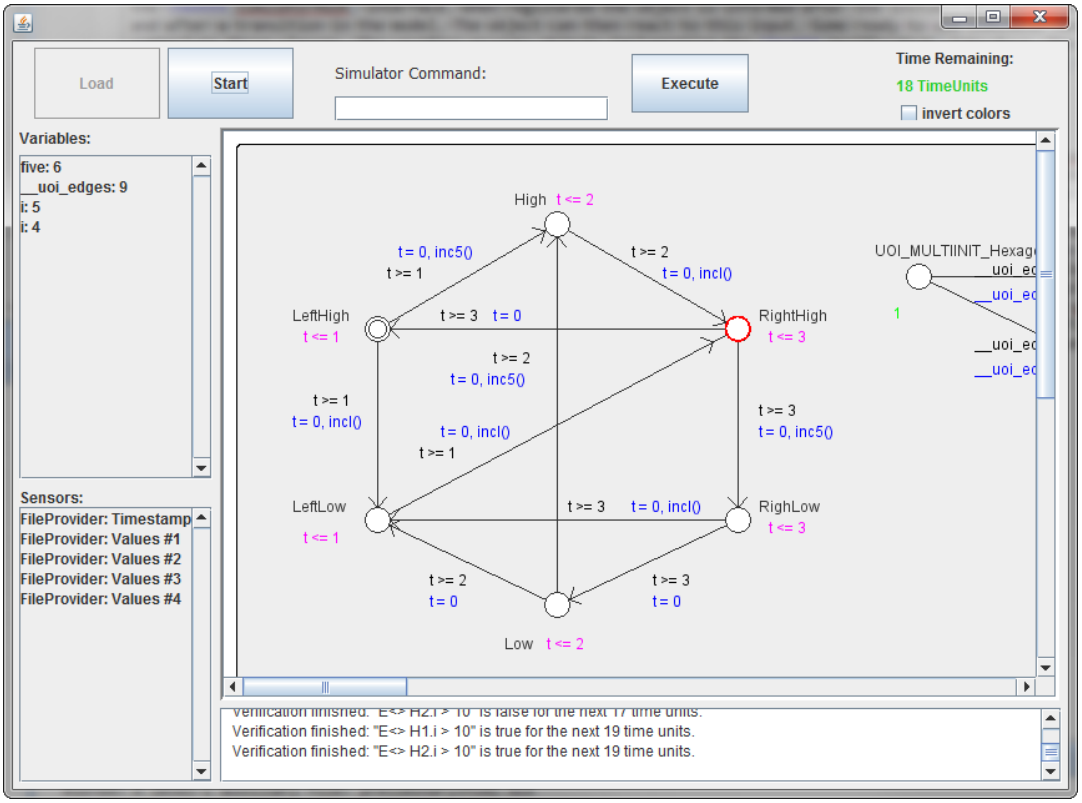


Figure 5.4: Main window of the visualization GUI

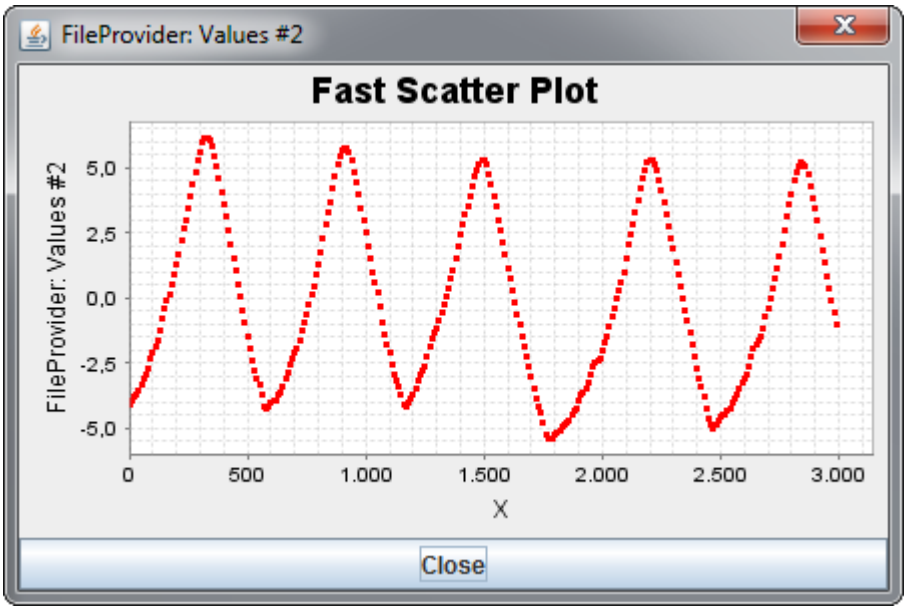
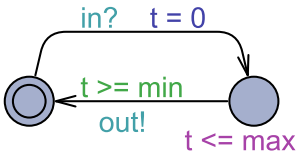


Figure 5.5: Plot of `DataSeries` object



(a) Updated channel model

```
chan channel1, channel2, light1, light2;
Control = Controller(channel1, channel2, 100);
C1 = Channel(channel1, light1, 0, 6);
C2 = Channel(channel2, light2, 0, 6);
T1 = Trafficlight(light1, light1, 15);
T2 = Trafficlight(light2, light2, 15);
system Control, C1, C2, T1, T2;
```

(b) Updated system declaration

Figure 5.6: Updated traffic light example model

must overlap. The resulting OMC application has hard real-time deadlines of 10 seconds.

After having obtained the deadlines one must ensure that those deadlines can actually be met. Experiments show that in the traffic light example the verification of the original UPPAAL models exceeds the deadline of 10 seconds most of the time and therefore a change to the models is required. Closer examination of the model shows that the original channel template leads to a huge state space because of the particular usage of data variables for the channel delays. Replacing the channel template with one that approximates the delays with a minimum and a maximum delay reduces the state space to manageable levels. Figure 5.6 shows the updated template and the modified system declaration. This modification to the channel template significantly reduces the state space of the model as no longer data variables for the delays need to be adjusted and stored. Now the necessary delay information is implicitly contained in the symbolic state of the channel clock t. This change to the model also means that the OMC application is required to calculate the minimum and maximum delay by itself. However, this fact does not cause any problems as the current delay information is known and a maximum rate of change is assumed.

With the desired functionality of the OMC application made clear one can then focus on the implementation and decide which framework classes can be used to create the application. The traffic light example only has a single class that specifies the application. This class is derived from `OMCApplication` and it makes use of the `UppaalSimulator` class, the `UppaalVerifier` class, and the `Variable` class to perform the model adjustments and the verification tasks. Furthermore, it must set up the data acquisition and processing pipeline. As mentioned above the real-world values for the delay are read from a file and thus the `FileProvider` class can provide the `DataSeries` objects with the data.

We now define the periodic task, `process`, of the online model-checking application that is used to carry out the model adaptation and the verification every 10 seconds. The source code is given in Algorithm 8. In the task we initially obtain the delays for the two traffic lights by locking the `consumer` provider, reading the delays from the two data series, and unlocking the provider again. Note that the `consumer` provider refers to the `FileProvider` that provides the real-world delays. Next, the minimum and maximum delays are calculated by taking into account a maximum rate of change

and the `Variable` objects for the delays are updated accordingly. Here, it is assumed that the delay may change by 6 time units over 2 verification periods.[2] Furthermore, note that negative minimum delays must be handled in a safe way. After having changed the delay variables the changes are committed to the simulator. This results in the updated system model where the model delays are equal to the real-world delays. This new model is then queried for a new safety guarantee in the last row of the task.

Algorithm 8 Periodic OMC task for traffic light example

```
1  private Runnable process  = new Runnable() {
2
3      @Override
4      public void run() {
5
6          consumer.lock();
7          double delay1 = consumer.getDataSeries(0).getMostRecentValue();
8          double delay2 = consumer.getDataSeries(1).getMostRecentValue();
9          consumer.release();
10
11         mins[0].setValue((int) (delay1 - 6 > 0 ? delay1 - 6 : 0));
12         maxs[0].setValue((int) (delay1 + 6));
13         mins[1].setValue((int) (delay2 - 6 > 0 ? delay2 - 6 : 0));
14         maxs[1].setValue((int) (delay2 + 6));
15
16         simulator.scheduleDataChange(mins[0]);
17         simulator.scheduleDataChange(maxs[0]);
18         simulator.scheduleDataChange(mins[1]);
19         simulator.scheduleDataChange(maxs[1]);
20         simulator.reconstruct();
21
22         verifier.verifyPossibly("T1.Green and T2.Green");
23     }
24 };
```

The remaining task for the creation of the OMC application is the specification of the data pipeline and setting up the simulation and verification environment. This setup is performed in the application constructor whose source code is depicted in Algorithm 9. At first, the periodic verification task `process` that was defined above is registered and its period is set. Next, a `FileProvider` is instantiated as a source for the observed channel delays. It is assumed that the delays are measured every second (period) and only the most recent value is of interest (history length of 1). The two delay data series provided by the `FileProvider` object are then registered at the `consumer` variable for this application as those two series are used as input to

[2] Remember that verification intervals are interleaved and the verification guarantee must hold for twice the task period.

the periodic verification task. Next comes the specification of the `Variable` objects that interface the model variables. For every minimum and maximum delay a variable is created. The `FileProvider` is then registered as a sensor such that its data can be inspected in the GUI if necessary. After that the simulator is set up by loading the model into the simulator, performing the initial reconstruction, and specifying that a time unit in the model is equivalent to $\frac{1}{10}$ seconds in the real world. Lastly, the verifier is setup in a similar manner by specifying that verifications should look 200 time units, i.e., 20 seconds, into the future and registering a verification hook for printing all verification attempts and results to the application console.

Algorithm 9 Constructor of OMC application for traffic light example

```
1  public TrafficLightExample(long period) {
2
3      processor.setProcess(process);
4      processor.setPeriod(period);
5
6      lights = new FileProvider(DELAYFILE, ",", "\n", 10000000001, 1);
7      consumer.addDataSeries(lights, 1);
8      consumer.addDataSeries(lights, 2);
9
10     mins = new Variable[] { new Variable("C1.min"), new Variable("C2.min") };
11     maxs = new Variable[] { new Variable("C1.max"), new Variable("C2.max") };
12
13     sensor.addSensor(lights);
14
15     simulator.load(MODELFILE);
16     simulator.reconstruct();
17     simulator.setRealtime(UppaalSimulator.SECONDS / 10);
18
19     verifier.setScope(200);
20     verifier.addVerifierHook(new VerificationPrinter());
21 }
```

The application can then be started by instantiating a `TrafficLightExample` object with a period of 10 seconds and calling its `start()` method assuming that this method has been overwritten to start the simulator, the file provider, and the periodic task.

Summarizing this chapter, in this dissertation a framework for the development of online model-checking applications was developed. The framework supports users by providing solutions for tasks that all OMC applications need to perform. The benefit is that users can work with an accessible and consistent environment that abstracts away most of the technical difficulties in OMC. In particular the problem of state space adaptation, i.e., the adaptation of a false model state to an observed state, is solved in a fashion that is suitable for a real-time application like OMC. Additionally the reuse of old UPPAAL models is possible because all means for adaptation are synthesized by the framework. The theoretical background of my state space adaptation approach is

the focus of the next chapter as it is the main theoretic contribution of this dissertation.

6 State Space Adaptation

For online model checking to be practical it is necessary that the simulation state space of the model can be adjusted to match observations of the real-world system. One can think of many different adjustments to the model that could be made and many ways to actually perform those adjustments. Such potential adjustments are, e.g., changing a data variable valuation, altering timing constraints, changing the current automaton location, adjusting clock valuations, or even modifying the automaton topology. However, all adjustments have in common that they need to preserve the parts of the current simulation state space that are not modified. One way to preserve those parts of the state space would be by directly modifying the data structures within the model checker. However, few model checkers allow tinkering with their internal variables during simulation and verification as such modifications can introduce invalid states and risk the model checker's integrity and therefore its soundness. In the context of this dissertation I developed an alternative way that leaves unmodified parts of the state space untouched. It reuses a concrete simulation trace to reconstruct the current state space. From the simulation trace an *initialization sequence* can be derived that provides a basis for further adjustments. These adjustments can then be performed by altering some of the initialization transformations to obtain the desired state space. Table 6.1 lists some state space adaptations that may be useful and the way they could be performed. Note that all those modifications may create invalid state spaces. For example, if one changes the current location to a new location and the new location has a different invariant than the previous one, then the new invariant may not be satisfied. How to perform particular changes in detail, what kind of invalid states may

Table 6.1: Model adjustments and means to perform them

Model modification	**Adjustment process**
Change data variable valuation	Inject an update transformation that sets the variable to the new value.
Change timing constraint	Modify constraint annotations, i.e., invariants and guards, in the model.
Change automaton location	Redirect last transition of the initialization sequence to the new location.
Change clock valuation	Inject (multiple) updates and constraint transformations to adjust the clock range and the differences to other clocks.
Change automaton topology	Insert or remove new locations and transitions to the model.
Change probabilities	Inject update transformation if probability is encoded as a data variable, otherwise modify probability annotation of the model.

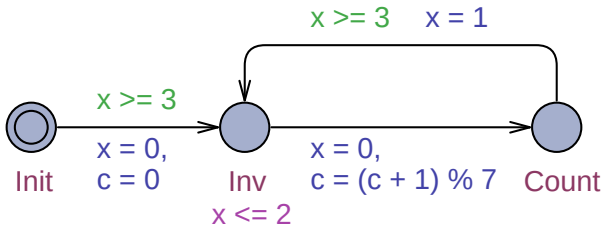


Figure 6.1: Example model

arise, and how to cope with them has not yet been explored thoroughly and could be the subject of future research in the domain of online model checking. In this dissertation I deal with the simple case of altering data variable valuations in models that do not contain locations that constrain those data variables. This restriction eliminates the possibility of invalid states occurring because of the modification; no out-of-range assignments to data variables are performed.

A problem directly connected to state space adaptation is the problem of state space reconstruction. As online model checking is a real-time application the model adjustment and verification step is time-critical. Thus, it is necessary for the initialization sequence to reconstruct the state space as efficiently as possible to leave as much time as possible for the model checker to perform its verification. It is obvious that the initialization sequence must be considerably shorter than the complete simulation trace because the simulation trace's length increases monotonously over time and eventually the time for the reconstruction process will exceed the OMC deadlines. In this dissertation I solve this previously unsolved problem that has yet to have significant attention in the literature as the online model-checking approach has been proposed only recently. Fortunately, the initialization sequence indeed can often be a lot shorter than the complete simulation trace. In many models a large number of past transitions do not have an impact on the current state space. In the model depicted in Figure 6.1 this behavior can be observed: in the location *Count* the clock x is in the range $[0, \infty)$. This valuation is completely defined by the clock reset of the ingoing edge and the absence of an invariant on the location. Differences between x and other clock variables would be defined by their respective valuation ranges. In this case, without other clocks, previous transitions do not influence the valuation of x. Therefore, instead of executing the transition sequence $Init \longrightarrow Inv \longrightarrow Count$ a single transition can create the same state: after introducing a new initial location a direct transition to *Count* with an update `x = 0` is sufficient to recreate the clock state space. During reconstruction it is thus beneficial to exploit that effects of certain state space transformations are overwritten by subsequent transformations.

In the remaining part of this chapter the adaptation of data variables in UPPAAL models with an efficient initialization sequence is presented in detail. At first, in Section 6.1, two transformation reductions are presented that are capable of deriving efficient initialization sequences. Then, Section 6.2 explains how to apply the reductions to perform online model checking with UPPAAL. Next, the reductions are

evaluated with regard to their efficiency in Section 6.3. At last, Section 6.4 summarizes model adaptation for online model checking and the transformation reduction methods.

6.1 Transformation Reductions

In this section I formalize the transformation reduction algorithms developed in the context of this dissertation. In Subsection 6.1.1 I first formalize a general transformation system and derive conditions when transformations may be removed from a transformation sequence without altering the final state. Afterwards, two transformation reductions are presented: Subsection 6.1.2 presents a graph-based method that finds shortest paths between states. This method has been presented at the International Symposium on Formal Methods 2014 (FM 2014) [85]. Subsection 6.1.3 presents a method based on use-definition chains that uses data-flow analysis techniques to derive transformations without influence on the last state. This reduction has first been published at the International Conference on Advances in System Testing and Validation Lifecycle (VALID 2013) [84] and was then extended for an article in the International Journal on Advances in Systems and Measurements [86].

6.1.1 A General Transformation System

At first, I define a general transformation system:

Definition 7. *A* valuation function *is a mapping*

$$e : \mathcal{V} \rightarrow \mathcal{D}$$

where $\mathcal{V}$ is a set of variable symbols and $\mathcal{D}$ is the valuation domain of these variables.

I denote the set of all such valuation functions by $\mathcal{E}(\mathcal{V}, \mathcal{D})$. Modifications to valuation functions are defined by *transformations*:

Definition 8. *A* transformation *is a mapping*

$$t : \mathcal{E}(\mathcal{V}, \mathcal{D}) \rightarrow \mathcal{E}(\mathcal{V}, \mathcal{D})$$

from one valuation function to another one.

I denote the set of all transformations by $\mathcal{T}(\mathcal{V}, \mathcal{D})$.

Let $e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{N-1}} e_N$ be a sequence of valuation functions created by the transformations t_i. I denote the transformation sequence $t_1, t_2, \ldots, t_{N-1}$ by $\vec{t}$ and the valuation function sequence $e_1, e_2, \ldots, e_N$ by $\vec{e}$. A transformation reduction can then be characterized by finding a subsequence $\vec{t}' \preceq \vec{t}$ such that $e_1 \overset{\vec{t}'}{\Rightarrow} e_N$ where $\overset{\vec{t}}{\Rightarrow}$ denotes the ordered application of the sequence of transformations $\vec{t}$ and $\vec{t}' \preceq \vec{t}$ if $\vec{t}'$ can be obtained by omitting transformations in $\vec{t}$. Such a transformation sequence $\vec{t}'$ reaches

the same last valuation function e_N but uses potentially fewer transformations than $\vec{t}$.

Reducing a transformation sequence requires that the influence of the involved transformations on the valuation functions is specified. I specify the effects in terms of *calculation functions*:

Definition 9. *A* calculation function *is a mapping*

$$m: \ \mathcal{V}^N \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{D}$$

where $N \in \mathbb{N}_0$ is a natural number including zero.

In this definition the purpose of $\mathcal{V}^N$ is to explicitly list all variable symbols that are used for the calculation such that I can refer back to them later on.

For the specification of transformations only *well-defined* calculation functions are relevant. The definition of well-defined calculation functions depends on the implementing term, i.e., the actual calculation instructions, of the calculation function. Therefore, the standard definitions of a *structure*, its *signature*, and its *terms* are given first [48].

Definition 10. *A* structure *A is an object defined by the following parts:*

- *A set of* elements *of the structure. This set is called the* domain *of the structure.*
- *A set of* constants *of the structure. This is a subset of the domain of A.*
- *For every positive integer n, a set of n-ary relations on the domain of A, each identified by n-ary* relation symbols.
- *For every positive integer n, a set of n-ary functions on the domain of A, each identified by n-ary* function symbols.

Definition 11. *A* signature *L of a structure A is specified by giving the set of constants of A and, for every positive integer n, the sets of function and relation symbols of A.*

Definition 12. *A* term *t of a signature L is defined recursively:*

- *Every variable symbol is a term of L.*
- *Every constant of L is a term of L.*
- *If $n > 0$, f is an n-ary function symbol of L, and $t_1, \ldots, t_n$ are terms of L, then $f(t_1, \ldots, t_n)$ is a term of L.*

Given these basic definition I can define when a calculation function m and its implementing term t_m are well-defined:

Definition 13. *A calculation function m is* well-defined *iff its implementing term t_m is* well-defined *and m is not equivalent to the identity calculation function I : $((x_1), e) \mapsto e(x_1)$.*

The calculation function I here is an identity in the sense that $\forall x\,[e(x) = I((x), e)]$, i.e., I does not modify valuations of a variable symbol x.

Definition 14. *An implementing term t_m of a calculation function $m((x_1, \ldots, x_N), e)$ is* well-defined *iff, in the structure A given by the signature specified by a set of constant symbols that includes $\{\, e(x_i) \,|\, 1 \leq i \leq N \,\}$, an empty relation symbol set, and a set of arbitrary functions symbols F^A,*

1. *t_m is a* ground *(or* closed*) term, i.e., no variables occur in it,*
2. *t_m contains for every input variable symbol x_i at least one constant symbol $e(x_i)$, and*
3. *t_m is* irreducible*, i.e., there exists no equivalent term in the term algebra that has fewer distinct constant symbols $e(x_i)$, e.g., because of an implicit zero term.*

I denote the set of all well-defined calculation functions by $\mathcal{C}(\mathcal{V}, \mathcal{D})$.

Example The calculation function

$$\text{add}: \ ((x_1, x_2), e) \mapsto e(x_1) + e(x_2)$$

is well-defined because no variables occur in the implementation term, both input variable symbols have been used with their respective valuation constants, and the term can not be further reduced if we assume + represents standard addition. In contrast, mappings from $((x_1, x_2), e)$ to the terms (1) $e(x_1) + v$, (2) $e(x_1)$, or (3) $e(x_1) + e(x_2) - e(x_2)$ would be ill-defined because (1) a variable is present, (2) not all constant symbols for the input variables were used, and (3) the term can be reduced to $e(x_1)$.

A transformation t can then be specified using a *specification set* S. A specification set is a set of *specification entries*:

Definition 15. *A* specification entry *is a tuple*

$$(x, V_x, m_x) \in \mathcal{V} \times \mathcal{V}^N \times \mathcal{C}(\mathcal{V}, \mathcal{D})$$

that associates a variable symbol x with a tuple of variable symbols V_x and a calculation function m_x where $N \in \mathbb{N}_0$ is a natural number including zero.

A specification entry encodes that the valuation of the variable symbol x, $e(x)$, needs to be updated to the value $m_x(V_x, e)$ where e is the valuation function of m_x. Again, for the specification of transformations only *well-defined* specification entries are relevant:

Definition 16. *A specification entry (x, V_x, m_x) with $V_x = (x_1, x_2, \ldots, x_N)$ and $m_x : \mathcal{V}^M \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{D}$ is* well-defined *iff $N = M$ and m_x is well-defined.*

To enable a mapping from a specification set S to the encoded transformation t the specification set itself also needs to be *well-defined*:

Definition 17. *A specification set S is* well-defined *iff* $\forall(x, V_x, m_x), (y, V_y, m_y) \in S\,[(x, V_x, m_x) \neq (y, V_y, m_y) \implies x \neq y]$*, i.e., the specification set contains at most a single entry for every variable, and all of its specification entries are well-defined.*

The set of all such well-defined specification sets is denoted by $\mathcal{S}$.

The transformation t encoded by a well-defined specification set S can then be obtained with the *specification function*.

Definition 18. *The* specification function *is a mapping*

$$\begin{aligned} s:\ \mathcal{S} &\to (\mathcal{E}(\mathcal{V},\mathcal{D}) \to \mathcal{E}(\mathcal{V},\mathcal{D})) \\ S &\mapsto (e \mapsto e') \qquad \textit{where} \\ e'(x') &= \begin{cases} m_x(V_x, e) & \textit{if } (x, V_x, m_x) \in S \wedge x' = x \\ e(x') & \textit{otherwise} \end{cases} \end{aligned}$$

We can now look at some properties of specification sets. Let f be the function that maps a tuple of values to the set of its values:

$$\begin{aligned} f:\ \mathcal{V}^N &\to 2^{\mathcal{V}} \\ (x_1, x_2, \ldots, x_N) &\mapsto \{\, x_i \,|\, 1 \leq i \leq N \,\} \end{aligned}$$

where $N \in \mathbb{N}_0$.

Definition 19. *Two specification sets S and S' are* equivalent *iff $s(S) = t = s(S')$ and $\exists(x, V_x, m_x) \in S \Leftrightarrow \exists(x, V'_x, m'_x) \in S'$ where $f(V_x) = f(V'_x)$.*

Thus, two specification sets S and S' are equivalent if they specify the same transformation t and they contain specification entries for the same variable symbols with variable symbol tuples made up from the same variable symbols.

Proposition 1. *If there is no equality relationship between the variable symbols all well-defined specification sets of a particular transformation t are equivalent.*

Proof. Let S, S', $S \neq S'$ be two well-defined specification sets such that $s(S) = t = s(S')$. Then S and S' must contain specification entries for the same variable symbols because the transformation t must perform the same valuation updates and the identity calculation function is excluded. It follows that $\forall(x, V_x, m_x) \in S, (x', V'_{x'}, m'_{x'}) \in S'\,[x = x' \implies m_x(V_x, e) = m'_{x'}(V'_{x'}, e)]$, i.e., the calculation functions in S and S' produce the same results for all variables and thus all pairs of implementing terms t_m and $t_{m'}$ must be equivalent in the term algebra. Assume w.l.o.g. that V_x contains a variable symbol y not contained in $V'_{x'}$. Then either the term t_m can be rewritten in a way that eliminates the constant $e(y)$ and t_m was not irreducible or the constant symbol $e(y)$ can not be eliminated. Then $t_m = t_{m'}$ implies a non-trivial equality relation between the variable symbols. Both cases contradict the assumption and thus $f(V_x) = f(V'_{x'})$. □

I now assume that there are no equality relations between the variable symbols and thus all specification sets S of a transformation t are equivalent. I denote any specification set of t by $S(t)$.

Proposition 2. *The* write set *of a transformation t, i.e., the set of variable symbols that have their valuations updated by t, is given by*

$$\mathcal{W}(t) = \bigcup_{(x,V_x,m_x)\in S(t)} \{ x \}$$

Proof. By definition of the specification function the valuations of exactly those variable symbols x_i are modified for which an specification entry is present in S. Thus, the set of written variables must contain exactly these variable symbols. As all specification sets for a transformation t are equivalent we can choose any representative $S(t)$ to extract the written variable symbols. □

Proposition 3. *The* read set *of a transformation t, i.e., the set of variable symbols whose valuations are necessary to calculate the updates of t, is given by*

$$\mathcal{R}(t) = \bigcup_{(x,V_x,m_x)\in S(t)} f(V_x)$$

Proof. According to the definition of the specification function a variable x is read by a transformation t iff its specification set S contains an entry (y, V_y, m_y) such that the implementing term t_m of m_y contains $e(x)$. As the calculation functions must be well-defined for the specification set $S(t)$ to be well-defined, the variable x is read iff there is a specification entry with $x \in V_y$. Thus, the set of variable symbols whose valuations are read is exactly the set of all distinct variable symbols in these tuples. As all specification sets for a transformation t are equivalent we can choose any representative $S(t)$ to extract these read variable symbols. □

In addition to simple transformations I furthermore define *compound transformations* to conveniently chain transformations together.

Definition 20. *A* compound transformation *is a transformation*

$$t_c = t_1 \circ t_2 \circ \cdots \circ t_n$$

such that $e_1 \xrightarrow{t_c} e_{n+1} \equiv e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} e_{n+1}$.

Note that $\circ$ denotes concatenation ($(a \circ b)(x) = b(a(x))$) in contrast to composition ($(a \circ b)(x) = a(b(x))$).

For compound transformations the write and read sets can be derived from the individual transformations t_i.

Proposition 4. *The* write set *of a compound transformation t_c is*

$$\mathcal{W}(t_c) = \bigcup_i \mathcal{W}(t_i)$$

Proof. As a compound transformation $t_c = t_1 \circ t_2 \circ \cdots \circ t_n$ writes all variables that the individual transformations t_i write, the write set of t_c is the union of the write sets of all transformations t_i. □

Proposition 5. *The* read set *of a compound transformation t_c is*

$$\mathcal{R}(t_c) = \bigcup_i (\mathcal{R}(t_i) \setminus \bigcup_{j<i} \mathcal{W}(t_j))$$

Proof. Let $t_c = t_1 \circ t_2 \circ \cdots \circ t_n$ be a compound transformation. The variables read by t_c are the variables every transformation t_i of t_c reads as long as that variable has not been overwritten by a transformation t_j of t_c at an earlier time, i.e., $j < i$. In that case the read value is computed within the compound transformation and its value is not an external input to t_c. Thus the read set of t_c, $\mathcal{R}(t_c)$, is the union of the individual read sets where the variables have been removed that have been overwritten by intermediate transformations. □

Using these definitions two cases exist where a transformation t_i may be removed from a transformation sequence $\vec{t}$ without changing e_N. The cases can be described in the following way:

1. *no write*
   $\mathcal{W}(t_i) = \emptyset$
   A transformation t_i may be removed if the write set $\mathcal{W}(t_i)$ is empty.

   Proof. As $\mathcal{W}(t_i) = \emptyset \implies S(t_i) = \emptyset$ by definition we find $e_i \xrightarrow{t_i} e_{i+1} \implies e_i = e_{i+1}$ and thus $e_i \xrightarrow{t_i} e_{i+1} \xrightarrow{t_{i+1}} e_{i+2} = e_i \xrightarrow{t_{i+1}} e_{i+2}$. □

2. *all writes never read*
   $\forall x \in \mathcal{W}(t_i)\,[j > i \wedge x \in \mathcal{R}(t_j) \implies \exists k\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]$
   The transformation t_i may be removed if for all following transformations t_j that read a variable written by t_i there is a transformation t_k between t_i and t_j that writes that variable.

   Proof. Let $e_i \xrightarrow{I} e'_{i+1} \to \cdots \to e'_M$ be the transition sequence $\vec{e}'$ that replaces t_i with the identity transformation I in $e_i \xrightarrow{t_i} e_{i+1} \to \cdots \to e_M$. Note that the new transformation sequence is equivalent to the old one with t_i removed. Now if we assume $e_M \neq e'_M$ then there must be a variable x such that $e_M(x) \neq e'_M(x)$. It follows that there must be a transformation t_j, $j < M$, such that $e_j(x) = e'_j(x)$ and $e_{j+1}(x) \neq e'_{j+1}(x)$ because $e_i = e'_i$. Hence, the variable x is written by t_j, i.e., $x \in \mathcal{W}(t_j)$. As $x \in \mathcal{W}(t_j)$, $e_{j+1}(x) = m_x(V_x, e_j)$ for $(x, V_x, m_x) \in S(t_j)$ and there must be a variable $y \in \mathcal{R}(t_j)$ such that $e_j(y) \neq e'_j(y)$ to satisfy $m_x(V_x, e_j) \neq m_x(V_x, e'_j)$. If $j = i$, we have a contradiction as $e_i = e'_i$ and are finished. Otherwise, we apply the same argument to the variable y and would get another transformation t_k, $k < j$, and a variable $w \in \mathcal{R}(t_k)$ such that

$e_k(w) \neq e'_k(w)$. This process leads to a contradiction to our assumption in at most $M - i$ iterations. □

Example Consider a transformation system where $\mathcal{V} = \{ x, y, z \}$ and $\mathcal{D} = \mathbb{Z}$ with an initial valuation function $e_1 \equiv 0$. In this system we want to allow assignments and additions. The calculation function add() and the family of calculation functions assign(d) provide this functionality:

$$\begin{aligned} \text{add} : \ & \mathcal{V}^2 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{D} \\ & ((x_1, x_2), e) \mapsto (e(x_1) + e(x_2)) \end{aligned} \qquad \begin{aligned} \text{assign}(d) : \ & \mathcal{V}^0 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) \to \mathcal{D} \\ & (\emptyset, e) \mapsto d \end{aligned}$$

A transformation $t_=$ that assigns the value v to the variable x can then be specified using the specification set $S(t_=) = \{ (x, \emptyset, \text{assign}(v)) \}$. It follows that $\mathcal{R}(t_=) = \emptyset$ and $\mathcal{W}(t_=) = \{ x \}$. The specification set of a transformation t_+ that performs the assignment $x = y + z$, then, is $S(t_+) = \{ (x, (y, z), \text{add}) \}$ and its read and write sets are $\mathcal{R}(t_+) = \{ y, z \}$ and $\mathcal{W}(t_+) = \{ x \}$, respectively. A more complex transformation performing the two sequential assignments $z = 2$ and $x = y + z$ can be specified using a compound transformation $t_\circ = t_1 \circ t_2$ where $S(t_1) = \{ (z, \emptyset, \text{assign}(2)) \}$ and $t_2 = t_+$. The read and write sets of $t_\circ$ are then $\mathcal{R}(t_\circ) = \{ y \}$ and $\mathcal{W}(t_\circ) = \{ x, z \}$. Note that a transformation with $S(t) = \{ (z, \emptyset, \text{assign}(2)), (x, (y, z), \text{add}) \}$ is not equivalent to $t_\circ$ as in t the addition will use the previous value for z. Thus, $\mathcal{R}(t)$ is $\{ y, z \}$ and not $\{ y \}$.

6.1.2 A Graph-based Transformation Reduction

The graph-based transformation reduction finds a transformation sequence $\vec{t}'$, $e_1 \overset{\vec{t}'}{\Rightarrow} e_{N'}$, derived from a given transformation sequence $\vec{t}$, $e_1 \overset{\vec{t}}{\Rightarrow} e_N$, such that $e_{N'} = e_N$ and $\forall j \, [1 < j < N' \implies \exists i \, [1 \leq i \leq N \wedge t'_j = t_i]]$, and the length of $\vec{t}'$ is minimal. To find such a sequence I developed a data structure denoted as a *reduction graph*. The reduction graph is a directed graph that captures all transitions between valuation functions that are possible with the transformations performed up to now. To give a better idea of the structure of the reduction graph I define a congruence relation between valuation functions under a particular transformation t:

$$e \equiv_t e' :\Longleftrightarrow \forall x \in (\mathcal{V} \setminus \mathcal{W}(t)) \cup \mathcal{R}(t) \, [e(x) = e'(x)]$$

Proposition 6. *This congruence relation identifies valuation functions as congruent if the application of the transformation t has the same resulting valuation function, i.e., $e \equiv_t e' \implies t(e) = t(e')$.*

Proof. Let x be a variable symbol from $\mathcal{V}$, t a transformation, and e and e' two congruent valuation functions according to $\equiv_t$. As either $x \in \mathcal{W}(t)$ or $x \notin \mathcal{W}(t)$ two cases for the application of t must be shown:

1. $x \in \mathcal{W}(t)$: In this case $\forall y \in \mathcal{R}(t)\,[e(y) = e'(y)]$ according to the definition of $\equiv_t$ and thus $t(e)(x) = t(e')(x)$ as t calculates the same new value for x for both valuation functions.

2. $x \notin \mathcal{W}(t)$: In this case $e(x) = e'(x)$ according to the definition of $\equiv_t$ and thus $t(e)(x) = t(e')(x)$ as the valuation of x remains unchanged.

The case analysis shows that in both cases $t(e)(x) = t(e')(x)$ and thus $e \equiv_t e' \implies t(e) = t(e')$. □

To construct the reduction graph I developed an algorithm that identifies congruent valuation functions using a projection. It constructs the graph on-the-fly when transformations are applied to the current valuation function.

Let $e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{N-1}} e_N$ be the sequence of valuation functions created by the transformations t_i. Then $\vec{G} = G_1 \xrightarrow{t_1} G_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{N-1}} G_N$ is the reduction graph sequence where $G_i = (N_i, V_i)$ are directed graphs with the node sets $N_i = \bigcup_{j \leq i} \{\, e_j \,\}$ and the labeled arc relations $V_i \subseteq N_i \times N_i \times \bigcup_{j<i} \{\, t_j \,\}$.

I now describe how the labeled arc relation sequence V_i is constructed based on the transformation sequence t_i and the projection operator $\Pi(t, e)$ that is derived from the congruence relation:

$$\begin{aligned} \Pi : \mathcal{T}(\mathcal{V}, \mathcal{D}) \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{E}(\mathcal{V}, \mathcal{D}) \\ (t, e) &\mapsto e' \text{ where} \\ e'(x) &= \begin{cases} e(x) & \text{if } x \in (\mathcal{V} \setminus \mathcal{W}(t)) \cup \mathcal{R}(t) \\ d & \text{otherwise} \end{cases} \end{aligned}$$

with d, a previously chosen, fixed element from $\mathcal{D}$, e.g., a zero element. Note that this projection sets all values of variable symbols that are not relevant for the application of t to d and thus equalizes the input valuation functions in these dimensions.

The construction of the V_i sequence is then as follows: Initially, the labeled arc set is empty, i.e., $V_1 = \emptyset$. The following arc sets are then obtained by adding new shortcut arcs using the forward and backward arc generators F and B and by adding the original arc from the sequence:

$$V_{i+1} = V_i \cup \{\, (e_i, e_{i+1}, t_i) \,\} \cup \mathrm{F}(i+1, t_i) \cup \mathrm{B}(i+1)$$

where

$$\begin{aligned} \mathrm{F}(i, t) &= \{\, (e_j, e_i, t) \mid j < i \wedge \Pi(t, e_j) = \Pi(t, e_i) \,\} \\ \mathrm{B}(i) &= \{\, (e_i, e_j, t_k) \mid j < i \wedge k < i - 1 \wedge \Pi(t_k, e_i) = \Pi(t_k, e_j) \,\}. \end{aligned}$$

In addition to the trivial arc, (e_i, e_{i+1}, t_i), this construction process adds new arcs to all congruent valuation functions of the transformation in question, that is, all valuation functions that are transformed to the same result by the transformation

Table 6.2: Graph transformations in the example

Transformations	Transformation Operations	Read Set $\mathcal{R}(t)$	Write Set $\mathcal{W}(t)$
t_1	$x = 1, z = -4$	$\{\,\}$	$\{\,x, z\,\}$
t_2	$x = x + z, y = 3, z = x + z$	$\{\,x, z\,\}$	$\{\,x, y, z\,\}$
t_3	$x = x + y, y = x + y, z = -3$	$\{\,x, y\,\}$	$\{\,x, y, z\,\}$
t_4	$x = x + y, y = -3$	$\{\,x, y\,\}$	$\{\,x, y\,\}$

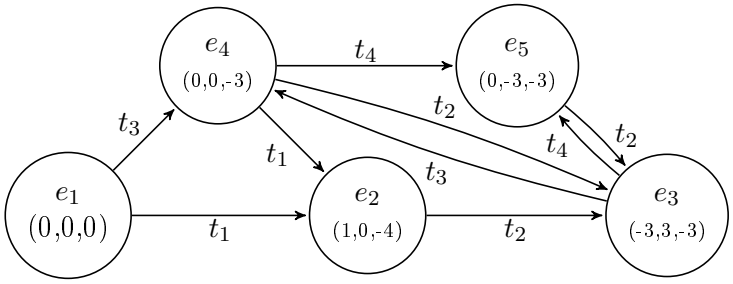


Figure 6.2: Graph G_5 of the example

obtain a new arc. Also, the new valuation function obtains additional outgoing arcs to previous valuation functions if it is congruent under a previous transformation.

The resulting graph structure reduces the initial problem of finding the shortest transformation sequence from a given start valuation function to a matching final valuation function using only known transformations to finding a shortest path in the reduction graph. Thus, an application of any shortest path search yields the reduced transformation sequence.

Example Consider the transformation sequence $e_1 \xrightarrow{t_1} e_2 \xrightarrow{t_2} e_3 \xrightarrow{t_3} e_4 \xrightarrow{t_4} e_5$ using the transformations given in Table 6.2 and the example transformation system from Subsection 6.1.1. Then, the resulting graph G_5 is depicted in Figure 6.2 and the construction process can be seen in Table 6.3. The shortest transformation sequence from e_1 to e_5 is $e_1 \xrightarrow{t_3} e_4 \xrightarrow{t_4} e_5$.

Correctness The transformation abstraction with its read and write sets accurately captures the transformation's effects as stated in Proposition 2 and Proposition 3.

Table 6.3: Graph construction process of the example

Iteration i	Transition Arc	Generator $\mathrm{F}(i+1, t_i)$	Generator $\mathrm{B}(i+1)$
1	(e_1, e_2, t_1)	$\{\,\}$	$\{\,\}$
2	(e_2, e_3, t_2)	$\{\,\}$	$\{\,\}$
3	(e_3, e_4, t_3)	$\{\,(e_1, e_4, t_3)\,\}$	$\{\,(e_4, e_2, t_1), (e_4, e_3, t_2)\,\}$
4	(e_4, e_5, t_4)	$\{\,(e_3, e_5, t_4)\,\}$	$\{\,(e_5, e_3, t_2)\,\}$

Thus, for the developed reduction approach to be correct, for all arcs added to the reduction graph it must be true that the application of the corresponding transformation on the source valuation function results in the target valuation function. All added arcs are part of either a forward arc generator set or a backward arc generator set, which are both based on the projection operator $\Pi(t, e)$. Thus, showing that $\Pi(t, e) = \Pi(t, e') \implies e \equiv_t e'$ yields the correctness of the approach:

Proof. Let x be a variable symbol from $\mathcal{V}$, t a transformation, and e and e' two valuation functions such that $\Pi(t, e) = \Pi(t, e')$. Then $\forall x \in (\mathcal{V} \setminus \mathcal{W}(t)) \cup \mathcal{R}(t)\,[e(x) = e'(x)]$ according to the definition of Π. This is the definition of $\equiv_t$ and thus $\Pi(t, e) = \Pi(t, e') \implies e \equiv_t e'$. □

Optimality The reduced transformation sequence obtained by the graph-based transformation reduction is optimal in the sense that no shorter transformation sequence can be derived from the original sequence by only removing transformations. The optimality directly results from the application of a shortest path algorithm as long as the arc generator functions do not fail to add a forward arc. No arcs are missed though as the forward generator F checks all potential forward transformation applications: a new transformation is applied to all previous valuation functions. The approach occasionally finds even shorter sequences when arcs are added by the backward arc generator B. These arcs, however, do not result in a transition sequence that can be obtained by only removing transformations from the original sequence.

6.1.3 A Use-definition Chain Reduction

As an alternative to the graph-based reduction approach I also developed a reduction based on use-definition chains [79]. In contrast to the graph-based method this method consumes less memory as only local reductions are performed and the state space does not need to be stored. However, this results in overall worse reduction results. The use-definition chain reduction identifies transformations satisfying the requirements presented in Subsection 6.1.1 by constructing use-definition chains. A use-definition chain is a data structure that provides information about the origins of variable values: for every use of a variable the chain contains the statements that have written the variable and ultimately lead to the current value allowing the identification of the origin of a particular variable value. The idea is to adapt the use-definition chain technique from static data-flow analysis on a program's source code to the state space reconstruction problem [79]: every entry in the model's difference bound matrix is treated as a variable and its uses and modifications are observed. DBM entries are only modified by applying a state space transformation on the DBM. Therefore an analysis of the read and write access to matrix entries for every transformation can be used to derive the use-definition chains where the transformations are the basic operations.

I now present an algorithm that removes unnecessary transformations using use-definition chains. The algorithm consists of two smaller algorithms: the Apply al-

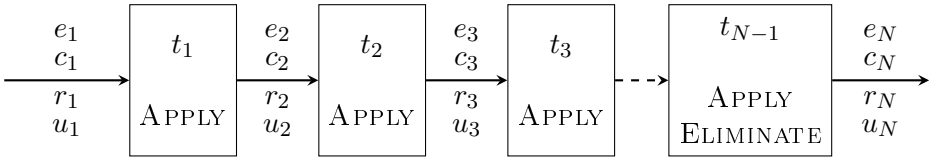


Figure 6.3: Algorithm execution sequence

gorithm and the Eliminate algorithm. The Apply algorithm is used to perform a transformation and the Eliminate algorithm removes unnecessary transformations in a sequence of applied transformations. The algorithm generates the following sequences:

- Transformation sequence $\vec{t} = t_1, t_2, \ldots, t_{N-1}$
- Valuation function sequence $\vec{e} = e_1, e_2, \ldots, e_N$
- Reference counter sequence $\vec{c} = c_1, c_2, \ldots, c_N$
- Responsibility sequence $\vec{r} = r_1, r_2, \ldots, r_N$
- Use-definition sequence $\vec{u} = u_1, u_2, \ldots, u_N$

where a *reference counter* is a mapping $c : \mathcal{T}(\mathcal{V}, \mathcal{D}) \cup \{\perp\} \to \mathbb{N}_0$, a *responsibility mapping* is a mapping $r : \mathcal{V} \to \mathcal{T}(\mathcal{V}, \mathcal{D}) \cup \{\perp\}$, and a *use-definition mapping* is a mapping $u : \mathcal{T}(\mathcal{V}, \mathcal{D}) \to 2^{\mathcal{T}(\mathcal{V},\mathcal{D})}$. Usage of the algorithms is assumed as follows: for a sequence of transformations $\vec{t}$ first the Apply algorithm is executed for all transformations in order, then the Eliminate algorithm is executed to obtain the reduced transformation sequence. Note that a transformation sequence $\vec{t}$ could be split into two subsequent sequences $\vec{t_1}$ and $\vec{t_2}$ and the Eliminate algorithm could be run on the sequence $\vec{t_1}$ as soon as all transformations in $\vec{t_1}$ have been applied. Thus, the Eliminate algorithm can be run with an appropriate transformation sequence after every execution of Apply. This procedure would achieve on-the-fly removal. However, for formalization purposes, I assume the algorithm execution sequence as given in Figure 6.3 where the initial mappings c_1, r_1, and u_1 satisfy $\forall t \in \vec{t}\,[c_1(t) = 0 \wedge u_1(t) = \emptyset] \wedge \forall x \in \mathcal{V}\,[r_1(x) = \perp] \wedge c_1(\perp) = |\mathcal{V}|$.

Algorithm 10 shows the Apply algorithm. The algorithm takes as parameters the to-be-applied transformation t, a valuation function e, a reference counter c, a responsibility mapping r, and a use-definition mapping u. Except for t all parameters are stored locally in lines 2 to 5 to allow internal manipulations. The results are returned in lines 17 to 20. The algorithm can be divided into two parts, one handling the reads of the transformation t and one handling its writes. Lines 6 to 11 process the variables read by t. For every variable x_i it is determined if the transformation responsible for its current valuation $r(x_i)$ is already used by t in line 7. If $r(x_i)$ was not marked used yet it is marked used in line 9 and the reference counter is increased accordingly in line 8. Here the algorithm essentially stores the transformation that

has produced the current value of x_i, which can be looked up in the responsibility mapping. Lines 12 to 16 then handle the writes of t. First the valuation function is updated to map x_i to the value $m_i(V_i, e)$ as specified by the transformation. Then line 14 reduces the reference counter of the transformation previously responsible for the value of x_i and line 15 updates the responsibility mapping such that now the transformation t is responsible for the value of x_i. This update makes sure that at all times one can use the responsibility mapping to obtain the transformation that has created the most recent valuation of a variable. Lastly in the return section in line 18 the reference counter is set to $|S(t)|$ to show that t is now responsible for $|S(t)|$ variable valuations.

Algorithm 11 shows the ELIMINATE algorithm. The algorithm takes a sequence of transformations $\vec{t}$, a reference counter c, and a use-definition mapping u as parameters where all transformations in $\vec{t}$ must already have been applied with the APPLY algorithm. In lines 2 and 3 the transformation sequence and the reference counter are again stored locally for modifications. The results of the algorithm are returned in lines 16 and 17. In between, in lines 4 to 15, a fix point is calculated by removing as many transformations from $\vec{t}$ as possible. The algorithm checks for every transformation t in $\vec{t}$ if the reference counter $c(t)$ evaluates to zero in line 7. If a transformation t satisfies $c(t) = 0$ the algorithm at first removes t from $\vec{t}$ in line 8, then adjusts the reference counter by decreasing all counters of transformations used by t in lines 9 to 11, and lastly schedules another iteration of the fix point calculation in line 12 as the modifications to the reference counters may enable additional removals.

Correctness I now show the correctness of the algorithm. To unify the handling of transformations I assume that there is implicitly a transformation t_e appended to the transformation sequence $\vec{t}$ that satisfies $\mathcal{R}(t_e) = \mathcal{V}$ to indicate that the final calculated values are actually read and need to be recreated correctly. Otherwise, the whole transformation sequence $\vec{t}$ could be removed as no data is consumed. As a transformation t can only be removed in line 8 of the ELIMINATE algorithm, which is only executed if $c(t) = 0$, the following implications need to be shown to prove the correctness of the algorithm:

1. $\mathcal{W}(t) = \emptyset \implies \forall c \in \vec{c}[c(t) = 0] \quad (\Rightarrow)$
   If a transformation t does not write any variable then the transformation may be removed at any time.

2. $\forall x \in \mathcal{W}(t_i)\,[\forall j\,[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists k\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \implies \forall c_l \in \vec{c}[l > \max k \implies c_l(t_i) = 0] \quad (\Rightarrow)$
   If every variable written by a transformation t_i is overwritten by transformations t_k before it is read by a transformation t_j then the transformation may be removed as soon as the last overwriting transformation t_k is performed.

3. $i < l \wedge c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset \vee \forall x \in \mathcal{W}(t_i)\,[\nexists j\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall k\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]] \quad (\Leftarrow^1)$

[1] *Derivation Note* Application of identities yields $\forall x \in \mathcal{W}(t_i)\,[\forall t_j \in \vec{t}[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists t_k \in$

Algorithm 10 APPLY algorithm

```
1: procedure APPLY(t, e, c, r, u)
2:     e_l ← e
3:     c_l ← c
4:     r_l ← r
5:     u_l ← u
6:     for all x_i ∈ R(t) do
7:         if r(x_i) ∉ u_l(t) then
8:             c_l ← c_l[r(x_i) ↦ c_l(r(x_i)) + 1]
9:             u_l ← u_l[t ↦ u_l(t) ∪ { r(x_i) }]
10:        end if
11:    end for
12:    for all (x_i, V_i, m_i) ∈ S(t) do
13:        e_l ← e_l[x_i ↦ m_i(V_i, e_l)]
14:        c_l ← c_l[r(x_i) ↦ c_l(r(x_i)) − 1]
15:        r_l ← r_l[x_i ↦ t]
16:    end for
17:    e' ← e_l
18:    c' ← c_l[t ↦ |S(t)|]
19:    r' ← r_l
20:    u' ← u_l
21: end procedure
```

Algorithm 11 ELIMINATE algorithm

Require: $\forall t \in \vec{t}[\text{APPLY}(t, ...)]$

```
1: procedure ELIMINATE(t⃗, c, u)
2:     t⃗_l ← t⃗
3:     c_l ← c
4:     repeat
5:         b ← true
6:         for all t ∈ t⃗_l do
7:             if c_l(t) = 0 then
8:                 t⃗_l ← t⃗_l \ { t }
9:                 for all s ∈ u(t) do
10:                    c_l ← c_l[s ↦ c_l(s) − 1]
11:                end for
12:                b ← false
13:            end if
14:        end for
15:    until b = true
16:    t⃗' ← t⃗_l
17:    c' ← c_l
18: end procedure
```

If a transformation t_i may be removed the transformation either does not write any variable or every variable written by it is overwritten by transformations t_k without being read beforehand.

I first derive two lemmas that characterize execution dependencies of certain lines in the algorithms to break the proof down into smaller parts.

Lemma 1. *Every execution of line 10 in the* ELIMINATE *algorithm is preceded by an execution of line 8 of the* APPLY *algorithm where $r(x) = s$ for some variable x, i.e., for every decrement of $c_j(s)$ in line 10 there is an increment of $c_i(s)$ in line 8 where $i < j$.*

Proof. An execution of line 10 in the ELIMINATE algorithm for a transformation s implies $s \in u(t)$ for some transformation t because of line 9. As only line 9 of the APPLY algorithm modifies u an execution of it is implied where $r(x) = s$ for some variable x. Additionally, line 10 may not be executed multiple times with the same transformation s for a single execution of line 8 because $u(t)$ is a set and therefore does not contain duplicates of s and the transformation t is removed from $\vec{t}_l$ in line 8 rendering a subsequent access to $u(t)$ impossible. It follows that every execution of line 10 in the ELIMINATE algorithm is paired with a preceding execution of line 8 in the APPLY algorithm. □

Lemma 2. *Before the execution of the* APPLY *algorithm for a transformation t there are zero variables x_i that satisfy $r(x_i) = t$. After its execution there are at all times at most $|S(t)|$ variables x_i that satisfy $r(x_i) = t$ for a transformation t.*

Proof. Only line 15 of the APPLY algorithm can modify $r(x)$. A valuation satisfying $r(x) = t$ can only be established when the algorithm is executed for the transformation t. In that case line 15 is executed $|S(t)|$ times due to line 12 and because $\forall (x, V_x, m_x), (x', V'_x, m_x) \in S(t)\,[x \neq x']$ there are exactly $|S(t)|$ variables x_i that satisfy $r(x_i) = t$ after the execution of the APPLY algorithm for t. Because subsequent executions of line 15 always result in valuations $r(x) \neq t$ for a variable x it follows that the number of variables x_i that satisfy $r(x_i) = t$ can only be reduced. The proposition follows by taking into consideration that initially $\forall x \in \mathcal{V}\,[r(x) = \bot]$. □

Corollary 1. *Line 14 of the* APPLY *algorithm may be executed for a transformation t at most $|S(t)|$ times, i.e., line 14 reduces $c(t)$ by one at most $|S(t)|$ times.*

Proof. Execution of line 14 for a transformation t implies that $r(x) = t$ for some variable x. Additionally, the execution is always followed by an execution of line 15, which sets $r(x) \neq t$. It follows that every execution of line 14 for a transformation t implies a reduction of the number of variables that satisfy $r(x) = t$ by one. It follows that line 14 may at most be executed $|S(t)|$ times for a transformation t due to Lemma 2. □

$\vec{t}[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \Leftrightarrow \forall x \in \mathcal{W}(t_i)\,[\nexists t_j \in \vec{t}[i < j \wedge x \in \mathcal{R}(t_j) \wedge \forall t_k \in \vec{t}[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$. Adding $c_l(t_j) \neq 0$ only makes sure the transformation can not be removed, i.e., it really exists.

I now prove the algorithm correct by showing that the presented requirements hold.

1. $\mathcal{W}(t) = \emptyset \implies \forall c \in \vec{c}\,[c(t) = 0]$

   Proof. There are four occurrences where reference counters may be modified: in lines 8, 14, and 18 of the Apply algorithm and in line 10 of the Eliminate algorithm.

   a) Apply algorithm, line 8
   Due to Lemma 2 $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset \implies \forall r \in \vec{r}\,[\nexists x \in \mathcal{V}\,[r(x) = t]]$ and thus line 8 cannot modify $c(t)$.

   b) Apply algorithm, line 14
   Due to Lemma 2 $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset \implies \forall r \in \vec{r}\,[\nexists x \in \mathcal{V}\,[r(x) = t]]$ and thus line 14 cannot modify $c(t)$.

   c) Apply algorithm, line 18
   As $\mathcal{W}(t) = \emptyset \implies S(t) = \emptyset$ line 18 sets $c(t)$ to $|S(t)| = 0$ if Apply is executed for t and $c(t)$ is not modified otherwise.

   d) Eliminate algorithm, line 10
   As line 8 cannot modify $c(t)$ (see above) line 10 cannot modify $c(t)$ due to Lemma 1.

   According to the case analysis it follows that $c_{\text{new}}(t) = c_{\text{old}}(t)$ or $c_{\text{new}}(t) = 0$ for every application of the Apply or Eliminate algorithm. Using that $c_1(t) = 0$ it follows that $\mathcal{W}(t) = \emptyset \implies \forall c \in \vec{c}\,[c(t) = 0]$ by induction. □

2. $\forall x \in \mathcal{W}(t_i)\,[\forall j\,[i < j \wedge x \in \mathcal{R}(t_j) \implies \exists k\,[i < k < j \wedge x \in \mathcal{W}(t_k)]]] \implies \forall c_l \in \vec{c}\,[l > \max k \implies c_l(t_i) = 0]$

   Proof. There are four occurrences where reference counters may be modified: in lines 8, 14, and 18 of the Apply algorithm and in line 10 of the Eliminate algorithm. Without loss of generality I consider the transformation t_i:

   a) Apply algorithm, line 8
   Assume a transformation t_j modifies $c(t_i)$ in line 8. Then Lemma 2 implies that $i < j$ and $\exists x \in \mathcal{V}\,[r(x) = t_i]$. It follows that $\exists x \in \mathcal{V}\,[x \in \mathcal{R}(t_j) \wedge x \in \mathcal{W}(t_i)]$ must be satisfied. Thus, the inner premise in the proposition holds and there must be a transformation t_k satisfying $i < k < j$ and $x \in \mathcal{W}(t_k)$. The execution of the Apply algorithm for the transformation t_k, however, then results in $r(x) \neq t_i$ (see proof of Lemma 2) and t_j can no longer modify $c(t_i)$. The premise thus prevents line 8 from modifying $c(t_i)$.

   b) Apply algorithm, line 14
   As only modifications to $c(t_i)$ are of interest only executions for transformations $t_j, j > i$ need to be considered due to Lemma 2 as line 14 requires $\exists x \in \mathcal{V}\,[r(x) = t_i]$ to modify $c(t_i)$. According to Corollary 1 line 14 may reduce $c(t_i)$ at most by $|S(t_i)|$. The maximum reduction occurs

if $\forall x \in \mathcal{W}(t_i)\,[\exists t_j \in \vec{t}[i < j \wedge x \in \mathcal{W}(t_j)]]$ (see proof of Corollary 1). According to the premise of the proposition this requirement is satisfied if $\forall x \in \mathcal{W}(t_i)\,[\exists t_j \in \vec{t}[i < j \wedge x \in \mathcal{R}(t_j)]]$. This requirement, however, is always satisfied because of the implicit transformation t_e at the end of the transformation sequence, which satisfies $\mathcal{R}(t_e) = \mathcal{V}$. It follows that the premise of the proposition implies the existence of a set of transformations T_r with transformations after t_i, which satisfies $\mathcal{W}(t_i) \subseteq \bigcup_{t \in \vec{t_r}} \mathcal{W}(t)$ and, thus, line 14 reduces $c(t_i)$ by $|S(t_i)|$ in total.

c) Apply algorithm, line 18
As only modifications to $c(t_i)$ are of interest only the execution for t_i needs to be considered. In that case line 18 sets $c(t_i)$ to $|\mathcal{W}(t_i)|$ as $|\mathcal{W}(t_i)| = |S(t_i)|$.

d) Eliminate algorithm, line 10
Due to Lemma 1 there is no execution of line 10 that modifies $c(t_i)$ as there is no modification of $c(t_i)$ in line 8 in the Apply algorithm (see above).

According to the case analysis $c(t_i)$ is modified in the following way: at first line 18 sets $c(t_i)$ to $|S(t_i)|$. Then the transformations from the set T_r are executed and lead to a monotonic descending $c(t_i)$ value (no reads). The execution of the last transformation from T_r results in $c(t_i) = 0$. This transformation is the transformation with the highest k of the transformations t_k in the proposition as all transformations t_k satisfy $\mathcal{W}(t_i) \cap \mathcal{W}(t_k) \neq \emptyset$. Thus, $\forall c_l \in \vec{c}\,[l > \max k \implies c_l(t_i) = 0]$ is satisfied and the proposition holds. □

3. $i < l \wedge c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset \vee \forall x \in \mathcal{W}(t_i)\,[\nexists j\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall k\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$

Proof. There are three occurrences where reference counters may be set or reduced to zero after t_i is processed. Lines 14 and 18 of the Apply algorithm and line 10 of the Eliminate algorithm potentially modify $c_l(t_i)$ in such a way:

a) Apply algorithm, line 14
In this case I prove $i < l \wedge c_l(t_i) = 0 \implies \forall x \in \mathcal{W}(t_i)\,[\nexists t_j \in \vec{t}[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in \vec{t}[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$ by contraposition. Assume x to be a variable satisfying $x \in \mathcal{W}(t_i)$ and assume t_j to be a transformation satisfying $i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall t_k \in \vec{t}[i < k < j \implies x \notin \mathcal{W}(t_k)]$. When executed t_j increases $c_j(t_i)$ in line 8 as $x \in \mathcal{R}(t_j)$ and no intermediate transformation t_k invalidates $r(x) = t_i$. Then for the transformation t_{l-1} to reduce $c_l(t_i)$ to zero in line 14 it is necessary to revert the increase by an execution of line 10 in the Eliminate algorithm before t_{l-1} is executed due to Corollary 1. Due to Lemma 1 the reduction must result from $c(t_j)$ being reduced to zero (Eliminate algorithm, line 7) as otherwise additional increases would have happened in line 8 beforehand. Assume t_m, $j < m < l - 1$ to be the transformation

that reduced $c_{m+1}(t_j)$ to zero to revert the increase of $c_j(t_i)$. One finds that $c_{m+1}(t_j) = 0 \implies c_l(t_j) = 0$ unless $c(t_j)$ is increased again between the execution of t_m and t_{l-1}. However, a reduction of $c(t)$ to zero implies $\nexists x \in \mathcal{V}\,[r(x) = t]$ and thus such an increase is impossible. It follows that if t_j exists $c_l(t_i)$ can not be reduced to zero.

b) Apply algorithm, line 18
Line 18 may only modify $c_l(t_i)$ if Apply is executed for t_i. In that case $c_l(t_i)$ is set to zero if $|S(t_i)| = 0$. As $|S(t_i)| = 0 \implies \mathcal{W}(t_i) = \emptyset$ it follows that in this case $c_l(t_i) = 0 \implies \mathcal{W}(t_i) = \emptyset$, which is part of the proposition.

c) Eliminate algorithm, line 10
If line 10 reduces $c_l(t_i)$ to zero then there must have been a reduction of a different reference counter to zero beforehand as line 7 requires $c(t) = 0$. As this argument holds recursively a reduction in line 10 ultimately implies a reduction due to either line 14 or line 18 in the Apply algorithm. Thus, a reduction in line 10 implies the implications for those lines found above.

Combining the case analysis results with the premise $i < l \wedge c_l(t_i)$ results in all cases in either $\mathcal{W}(t_i) = \emptyset$ or $\forall x \in \mathcal{W}(t_i)\,[\nexists j\,[i < j \wedge x \in \mathcal{R}(t_j) \wedge c_l(t_j) \neq 0 \wedge \forall k\,[i < k < j \implies x \notin \mathcal{W}(t_k)]]]$ yielding the proposition. □

6.2 State Space Adaptation in UPPAAL

To carry out state space adaptation for the UPPAAL model checker two questions need to be answered. The first question is on how the general transformation system used in the transformation reductions can be specialized for difference bound matrices such that the reductions can be used in UPPAAL's time space. It is answered in Subsection 6.2.1. The second question, on how to synthesize a new UPPAAL model given the reduced transformation sequence and the original model, is covered in Subsection 6.2.2.

6.2.1 Reducing DBM Transformations

To apply the reduction methods to UPPAAL's state space the general formalization components from the reductions are now specialized such that they represent UPPAAL's state space and its transformations correctly. The reductions are applied to the time state of UPPAAL only. The data state need not be considered because it can be set directly and thus does not need to be reconstructed in an online model-checking context.

Consider an UPPAAL model $\mathcal{M}$ with the clock variable set $\mathcal{C}$. Due to the layout of a difference bound matrix $|\mathcal{C}_0|^2$ variables are necessary to represent the time state of $\mathcal{M}$. It follows that for the application of the reductions I can specialize the general transformation system by defining the variable set by $|\mathcal{V}| = |\mathcal{C}_0|^2$ and the domain of those variables by $\mathcal{D} = \mathcal{K}$, the set of DBM entries (see Sect. 3.2.1). I denote the variables by $\mathrm{DBM}_{r,c}$ where r denotes the row number and c denotes the

column number. Furthermore, I need to define the transformations on DBMs that UPPAAL uses to modify the time state in the transformation system. The relevant DBM transformations are the UP transformation, the RESET(x, d) transformation, and the CONSTRAINT(x, y, k) transformation [13]:

UP	Removes all upper bounds on all single clocks
RESET(x, d)	Sets the clock variable x to the value d and adjusts constraints on that clock accordingly
CONSTRAINT(x, y, k)	Introduces a new upper bound on a clock or on a difference of clocks and propagates dependencies

To specify these transformations in the general transformation system I first introduce three families of calculation functions and a single calculation function. The first two families deal with constant values: one is the assign(k) calculation, which is used to assign a constant value to a variable. The second one is the add(d) calculation, which assigns to a variable the sum of a constant value and the valuation of a variable:

$$\begin{aligned} \text{assign}(k) : \mathcal{V}^0 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{K} & \qquad \text{add}(d) : \mathcal{V}^1 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{K} \\ (\emptyset, e) &\mapsto k & ((x_1), e) &\mapsto e(x_1) + (d, \leq) \end{aligned}$$

Next, I define minassign(v) and minadd, which assign minima to variables. The calculation function family minassign(k) assigns to a variable the minimum of a valuation of a variable and a constant value. The calculation function minadd calculates the minimum between the valuation of a single variable to the sum of two variable valuations:

$$\begin{aligned} \text{minassign}(k) : \mathcal{V}^1 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{K} \\ ((x_1), e) &\mapsto \min(e(x_1), k) \\ \text{minadd} : \mathcal{V}^3 \times \mathcal{E}(\mathcal{V}, \mathcal{D}) &\to \mathcal{K} \\ ((x_1, x_2, x_3), e) &\mapsto \min(e(x_1), e(x_2) + e(x_3)) \end{aligned}$$

Using theses calculation functions the UP, RESET(x, d), and CONSTRAINT(x, y, k) transformations can be defined by giving their specification sets $S(t)$. Note that I denote the indices for the clocks x and y in the corresponding DBM by i_x and i_y. I start with the UP transformation, which sets all values in the first DBM column except the first one to ∞:

$$S(\text{UP}) = \{\, (\text{DBM}_{i,1}, \emptyset, \text{assign}(\infty)) \mid 1 < i \leq |\mathcal{C}_0| \,\}$$

The RESET(x, d) transformation sets the upper and the lower bound of x to d and then adjusts all constraints in its row and column accordingly. It thus can be modeled as a compound transformation:

$$\begin{aligned} \text{RESET}(x, d) &= t_s \circ t_p \\ S(t_s) &= \{\, (\text{DBM}_{i_x,1}, \emptyset, \text{assign}((d, \leq))), (\text{DBM}_{1,i_x}, \emptyset, \text{assign}((-d, \leq))) \,\} \\ S(t_p) &= \{\, (\text{DBM}_{i_x,i}, (\text{DBM}_{1,i}), \text{add}(d)), \\ &\qquad (\text{DBM}_{i,i_x}, (\text{DBM}_{i,1}), \text{add}(-d)) \mid 1 < i \leq |\mathcal{C}_0| \,\} \end{aligned}$$

The $\textsc{Constraint}(x, y, k)$ transformation is divided into the introduction of the constraint and the propagation of the constraint and is, thus, also modeled as a compound transformation:

$$\begin{aligned}\textsc{Constraint}(x, y, k) = t_c \circ\ & \\ & t_{1,1} \circ \cdots \circ t_{1,|\mathcal{C}_0|} \circ \\ & t_{2,1} \circ \cdots \circ t_{2,|\mathcal{C}_0|} \circ \\ & \vdots \\ & t_{|\mathcal{C}_0|,1} \circ \cdots \circ t_{|\mathcal{C}_0|,|\mathcal{C}_0|} \\ \text{with} \quad t_{i,j} = t_{i,j,1} \circ t_{i,j,2}\end{aligned}$$

$$\begin{aligned}S(t_c) &= \{\, (\mathrm{DBM}_{i_x,i_y}, (\mathrm{DBM}_{i_x,i_y}), \mathrm{minassign}(k)) \,\} \\ S(t_{i,j,1}) &= \{\, (\mathrm{DBM}_{i,j}, (\mathrm{DBM}_{i,j}, \mathrm{DBM}_{i,i_x}, \mathrm{DBM}_{i_x,j}), \mathrm{minadd}) \,\} \\ S(t_{i,j,2}) &= \{\, (\mathrm{DBM}_{i,j}, (\mathrm{DBM}_{i,j}, \mathrm{DBM}_{i,i_y}, \mathrm{DBM}_{i_y,j}), \mathrm{minadd}) \,\}\end{aligned}$$

Example Consider the transformation sequence $\textsc{Up} \to \textsc{Reset}(x, 2)$. Its application to the time state represented by the DBM

$$\begin{array}{c|cccc} & \mathbf{0} & x & y & z \\ \hline \mathbf{0} & (0,\leq) & (-3,\leq) & (-2,\leq) & (0,\leq) \\ x & (5,<) & (0,\leq) & (1,\leq) & \infty \\ y & \infty & (-1,\leq) & (0,\leq) & \infty \\ z & (7,<) & (3,\leq) & (0,<) & (0,\leq) \end{array}$$

results in

$$\xrightarrow{\textsc{Up}} \begin{bmatrix} (0,\leq) & (-3,\leq) & (-2,\leq) & (0,\leq) \\ \infty & (0,\leq) & (1,\leq) & \infty \\ \infty & (-1,\leq) & (0,\leq) & \infty \\ \infty & (3,\leq) & (0,<) & (0,\leq) \end{bmatrix} \xrightarrow{\textsc{Reset}} \begin{bmatrix} (0,\leq) & (-2,\leq) & (-2,\leq) & (0,\leq) \\ (2,\leq) & (0,\leq) & (0,\leq) & (2,\leq) \\ \infty & \infty & (0,\leq) & \infty \\ \infty & \infty & (0,<) & (0,\leq) \end{bmatrix}$$

6.2.2 Synthesizing Adjusted Models

As a last step to enable online model checking with UPPAAL the reduced transformation sequence needs to be converted into an updated UPPAAL model. The complete state space reconstruction process for UPPAAL implemented in the OMC framework (see Chapter 5) consists of three phases:

1. *Initialization* This phase is performed once when the system model is loaded and reconstructed for the first time. It canonizes the model by introducing general starting points for future model syntheses and extracts necessary information from the model, e.g., clock and variable definitions.

2. *Simulation* This phase is the main phase where the system is actually running. Here transitions in the model are selected according to intended system behavior, the transitions are executed and stored together with the resulting model states. Simultaneously, the reduction method of choice is applied to either construct the reduction graph or the use-definition chains for future reductions during the synthesis step.

3. *Synthesis* This phase is performed every time a model reconstruction is initiated. Afterwards the OMC framework returns to the simulation phase. The synthesis phase first calculates the reduced transformation sequence based on the data structure generated during simulation. Then all transformations of reduced DBM transformation sequences are grouped such that they form valid UPPAAL transitions. The resulting initialization sequence is then inserted into a copy of the original model such that the last transitions connect to those locations that were active when the reconstruction was initiated. Data updates are also performed by these final initialization transitions.

Note that the synthesis of the actual UPPAAL model from the reduced transformation sequence has to take into consideration that in UPPAAL automata are potentially instantiated multiple times with possibly different parameters. Therefore, during the initialization step the analysis of the model finds the definitions for the automaton instantiation and saves the relevant parameters. Also, as the location state needs to be correctly reconstructed an automaton that is instantiated multiple times has multiple initializations sequences for every instantiation. The implementation uses a single bounded integer variable in conjunction with appropriate guards to correctly allocate the initialization sequence to instantiations within a template.

Another important aspect of the synthesis step is that the model initialization needs to be self-contained, i.e., the initialization of multiple automata needs to finish synchronously to prevent parts of the model from advancing prematurely. As the number of initialization transitions per automaton can differ a broadcast channel synchronizes the last transitions to the original model. These final transitions also initialize the data variables to their potentially intentionally modified values. The initialization of global variables is done in an additional initialization automaton that is added if necessary.

The synthesis algorithm can be described on a high level as follows if one assumes $\vec{t}$ to be the reduced transformation sequence of DBM transformations:

1. **Initial Locations** For every template in the UPPAAL model create a new initial location. If a template is instantiated multiple times add one location per instance and add edges from the newly created initial location to these locations. Add guards to the edges such that one obtains one initial location per template instance, e.g., enumerate the edges, assign a guard based on its number, and add an update that increases the current number for the next transition. For every instance store the new initial locations as the current locations.

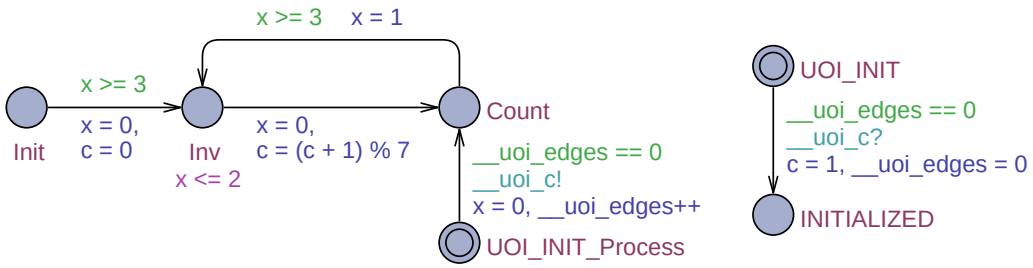


Figure 6.4: Reconstructed example model

2. **Global Initialization** Add a new dummy template with two locations to the model and add it to the system declaration once. Mark the initial location of the template as the current location and the other one as the final location.

3. **Slicing** While it is possible to obtain a subsequence of the form

   $$(\textsc{Constraint}_{<}\ \textsc{Reset}^{*})?\ (\textsc{Up}|\textsc{Constraint}_{>})\ \textsc{Constraint}^{*}$$

   from $\vec{t}$ where $\textsc{Constraint}_{<}$ denotes a constraint with a negative value k and $\textsc{Constraint}_{>}$ denotes one with a positive value perform the following steps with the longest so matching sequence $\vec{s}$:

   a) **New locations** For every template that has a clock mentioned in a Constraint or Reset transformation in $\vec{s}$ add a new location to that template, add an edge from the current location to the new one, and mark the new one as the current location.

   b) **Prevent deadlocks** For every template where the current location is restricted by an invariant add a new location, add an edge from the current location to the new one, and mark the new one as the current location.

   c) **Transformations** For every transformation in $\vec{s}$ add the matching annotation to the created edges, e.g., for a $\textsc{Reset}(x, d)$ transformation add an `x = d` update annotation. For global clocks use the dummy template.

   d) **Synchronization** To all newly created edges in this step add synchronization labels such that all edges fire at the same time.

4. **Finalization** For every template instance add an edge from its current location to its final location in the original model. Add appropriate synchronization such that all edges are fired at the same time. Furthermore, add the data variable updates to these edges to obtain the desired values.

To give an idea of a model synthesis, Figure 6.4 depicts on the left side the reconstructed model of the example model from Figure 6.1 after the execution of two transitions. On the right side the additional initialization automaton is shown that sets the global, bounded integer c to 1. The model initialization transition sets the clock x to 0 and the location is correctly initialized to *Count*. The reconstructed model

only needs to execute a single transition in contrast to the original model, which requires two transitions to reach the correct state. For DBM transformations the reconstructed model uses three transformations, UP, RESET$(x, 0)$, and UP, while the original model needs seven, UP, CONSTRAINT$(0, x, (3, \leq))$, RESET$(x, 0)$, UP, CONSTRAINT$(x, 0, (2, \leq))$, RESET$(x, 0)$, UP.

6.3 Reductions Evaluated

The presented reduction methods and the transformation system specialization to UPPAAL's state space have been implemented in the OMC framework (see Chapter 5). Both reductions were evaluated with respect to the real-time constraints that online model checking imposes. All experiments were carried out on an Intel Core i7-3720QM CPU running at 2.6GHz with 16GB of available memory on a system running Windows 7 64-bit. For the evaluation of the reduction methods seven different UPPAAL models were used. Four of the models come with the UPPAAL tool suite for demonstration purposes and three were taken from scientific case studies. The models are listed in the following:

- *2doors* A model of a synchronization scenario involving two doors and two users that may block each other. This model is part of UPPAAL's demonstration models.
- *bridge* A model of a system where soldiers with different walking speeds are required to cross a bridge. Crossing the bridge is only possible with a torch. Only one torch is available and may be shared by two soldiers. This model is part of UPPAAL's demonstration models.
- *train_gate* A model of a system where multiple trains pass a gate that may only accommodate a single train at a time. Trains need to stop in time and a first-come first-serve scheduling is employed. This model is part of UPPAAL's demonstration models.
- *fischer* A simple model of Fischer's mutual exclusion protocol [64] with six participants. This model is part of UPPAAL's demonstration models.
- *cdmacd* A model of the carrier sense multiple access method with collision detection. This model has been developed in a case study [53]. Note that the number of participating entities is variable and is appended to the model name for clarity.
- *tdma* A model of a start up sequence for the time division multiple access method. This model has been developed in a case study [73].
- *bmp* A model of the biphase mark protocol, a protocol used for transmission of bit strings and clock edges, e.g., in microcontrollers. This model has been developed in a case study [94].

Table 6.4: Reduction results of the graph-based method

Model	Length Average		Length Deviation		3σ-Bound
	Beginning	End	Beginning	End	
bridge	277.2	5150.8	158.4	162.6	-
csmacd2	5.2	5.1	1.4	1.2	8.7
2doors	55.3	35.1	20.3	13.0	74.1
bmp	55.1	44.2	33.9	27.5	126.7
train_gate	466.8	1289.1	278.5	194.2	1871.7
fischer	286.6	138.2	141.0	45.7	275.3
tdma	414.6	201.8	265.0	147.6	644.6
train_gate2	1900.1	1357.3	749.8	422.8	2625.7

The experiment results for the graph-based reduction method are presented in Subsection 6.3.1. The results of the use-definition chain method are given in Subsection 6.3.2.

6.3.1 Graph-based Reduction Results

To determine the efficiency of the graph-based reduction method every model was simulated five times by executing 5000 transitions per model. I argue that five simulations per model are sufficient to deduce meaningful statements about the reduction results because the length of the executed transition sequences is big in relation to the state space of the models and thus at the end of the simulations a significant portion of the state space has been explored in every run yielding comparable reduction graphs. Note that a single transition in an UPPAAL model generally results in multiple transformations on the current difference bound matrix. Every ten transitions I determined and recorded the length of the reduced transformation sequence. This approach allowed me to evaluate the development of the length of the reduced transformation sequence over time. For online model checking a time-independent upper bound on the length is required as otherwise real-time deadlines will be missed at some point. Table 6.4 gives an overview of the experiment results by comparing the average transformation length and its deviation in the beginning of the experiments (first 10% of data points) to the end of the experiments (last 10% of data points). Additionally, an upper bound for the reduced sequence per model is obtained using the 99.9% 3σ-confidence interval. The *train_gate2* experiment is an additional, longer experiment running the *train_gate* model where 10000 transitions were executed and the first and last 20% of data points were evaluated as the initial experiments were inconclusive because the initial simulation time was too short.

In general, the results show that the average length and deviation decreases over time. This behavior can be attributed to the gain of knowledge the graph-based reduction method has: in the beginning no data of the model's data space is known but over time more and more shortcuts are added to the reduction graph. Also,

reasonable upper bounds can be obtained for every model such that timely model adjustments required by online model checking seem feasible.

For the individual evaluation of the models Figure 6.5 shows the diagrams I obtained for each model by plotting the reduced sequence length over the original lengths. The black lines indicate the general trend of the data series over time. The diagram for the *bridge* model (Fig. 6.5a) shows the worst result of the experiments. All data series show a clear upward trend and thus in this model a linear time dependency exists. The reduction has limited applicability to this model as no drops in the reduced length can be observed. This behavior can be attributed to the fact that the *bridge* model has a global clock variable that never gets reset and thus the model never returns to a previous time state.

In contrast, the diagram for the *csmacd2* model (Fig. 6.5b) clearly shows a limited state space for this model. No reduced transformation sequence length exceeds a length of seven if the initial few transformations are ignored where no information on the state space is known yet. The average reduced length is nearly constant and, thus, the model does not show a time dependency.

Next, the diagram for the *2doors* model (Fig. 6.5c) shows high variance over all data series. This high variance is the result of short cycles in the model such that a sampling rate of one data point per ten transitions is too low and thus no correlation between neighboring data points is seen. However, the absolute reduced length never exceeds 150 transformations and the average length does not increase over time.

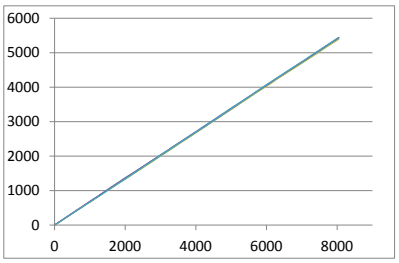
The diagram for the *bmp* model (Fig. 6.5d) shows comparable behavior: high variances in length, a low average length, and some rare spikes, although, in absolute values, the reduced length never exceeds 230 here.

In the diagram for the *train_gate* model (Fig. 6.5e) all data series show several huge drops in the reduced sequence length with near linear gains in between. This behavior can be explained by the relatively long cycles until a previous time state is reached in the model. The long cycles result from the large number of possible ways to interweave the approaching trains. In absolute values the *train_gate* model generally exhibits the longest reduced transformation sequences. As the possibility for a time dependency could not completely be eliminated in this experiment Figure 6.5f shows the extended experiment. Here, the time independence is depicted more clearly.

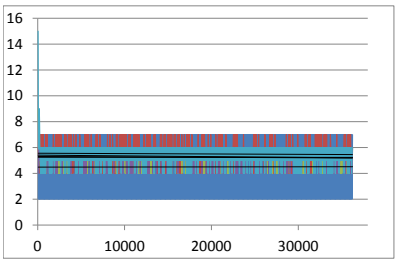
The diagram for the *fischer* model (Fig. 6.5g) clearly shows that the graph-based reduction method gains knowledge of the model state space over time. In the beginning, the reduced transformation sequences are relatively long while at the end lower average lengths are obtained and the variance decreases significantly.

The diagram for the *tdma* model (Fig. 6.5h) shows a combination of the *train_gate* behavior and the *fischer* behavior. The knowledge gain of the reduction method is clearly visible because the variance of the data series decreases over time. Still, occasionally an unknown part of the state space is explored leading to relatively huge reductions when the simulation returns to an already explored state. Again, no time dependency can be identified.

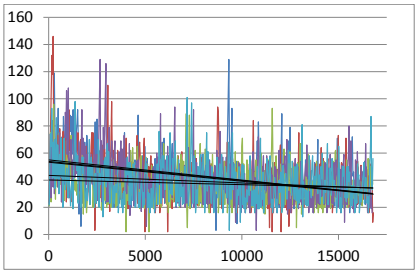
In general, this evaluation shows that as long as the model has a limited state space during simulation, i.e., eventually all clocks are reset, the proposed method



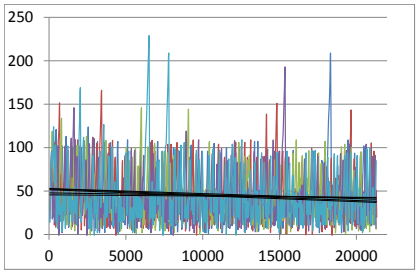
(a) *bridge* experiment



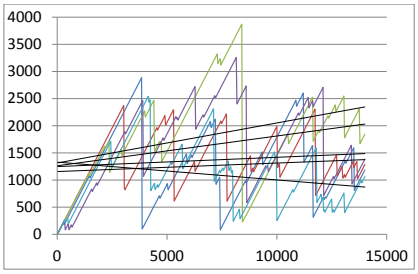
(b) *csmacd2* experiment



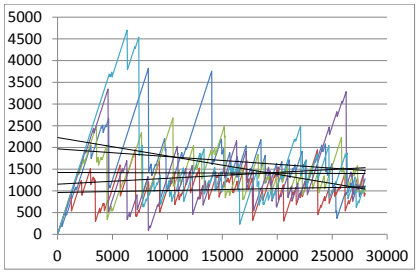
(c) *2doors* experiment



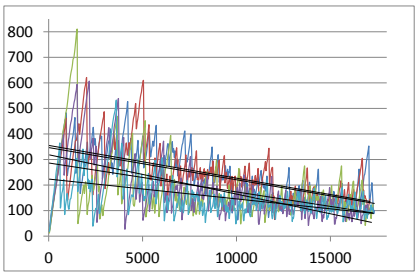
(d) *bmp* experiment



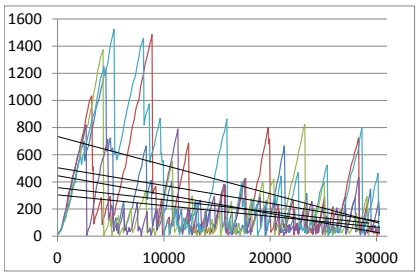
(e) *train_gate* experiment #1



(f) *train_gate* experiment #2



(g) *fischer* experiment



(h) *tdma* experiment

Figure 6.5: Graph-based transformation reductions over time

Table 6.5: Graph-based reduction run times

Model	*bridge*	*csmacd2*	*2doors*	*bmp*	*train_gate*	*fischer*	*tdma*
Graph	0.8	0.02	0.2	0.3	2.2	2.5	2.7
Search	69.8	0.2	8.0	17.6	198.1	279.5	236.6
Size	206	13	43	5908	12955	3458	>3000000

Table 6.6: Reduction results of the use-definition chain method

Model	**Transitions**			**Transformations**		
	Before	**After**	**Reduction**	**Before**	**After**	**Reduction**
2doors	100	65.89	34.1%	364.7	254.46	30.2%
bridge	100	68.21	31.8%	188.39	144.09	23.5%
train-gate	100	66.18	33.8%	320.09	214.17	33.1%
fischer	100	91.27	8.7%	345.33	249.46	27.7%
csmacd2	100	100	0%	709.71	434.19	38.8%
csmacd32	75.58	75.58	0%	1818.6	327.49	79.7%
tdma	100	68.16	31.8%	719.88	240.11	66.6%
2doors	1000	627.9	37.2%	3722.3	2612.9	29.8%
bridge	1000	641.3	35.9%	1882.8	1436.4	23.7%
train-gate	1000	606.1	39.4%	3200.1	2194.1	31.4%
fischer	1000	853	14.7%	3455.3	2486.8	28%
csmacd2	1000	1000	0%	7238.1	4375.5	39.5%
csmacd32	619.6	619.6	0%	22491.1	2540.3	84%
tdma	1000	663.1	33.7%	6446.3	2651.5	58.9%

reduces significantly the number of DBM transformations required to reach the same state. The reduction results show a positive time dependence: over time the reduction efficiency rises as information on the state space is gained. The benefit is that upper bounds for the reconstruction length can be established for every model. Furthermore, the average reduced length for a model is constant over time barring small fluctuation and, thus, the graph-based method may facilitate online model checking with its real-time requirements.

Performance The performance of the graph-based reduction method is directly related to the size of the state space of the UPPAAL model. Every DBM transformation introduces a new node in the graph if the valuation function resulting from the transformation is new. Further, a node has at maximum one incoming edge from every

node in the graph. It follows that $|N_i|$ and $|V_i|$ in the reduction graph sequence $G_i = (N_i, V_i)$ are bounded by $|N_i| \leq i$ and $|V_i| \leq |N_i|^2$. However, an additional static bound $|N_i| \leq k$ exists if the state space of the model is limited as, at some point, the complete state space is incorporated in the graph and therefore its growth comes to a halt.

The implementation in the OMC framework currently uses a simple implementation of Dijkstra's algorithm for the shortest path search and a sequential check of the projections to join nodes. In this case the worst case performance of the method is $O(k^2)$. The implementation can be further improved by using a more efficient search algorithm and implementing a cache for the projections that reuses previous projection results.

As a general overview of run times, Table 6.5 shows the average run times of the reduction method in milliseconds during the experiments. The first row displays the time necessary to extend the reduction graph for one DBM transformation. In the second row the times for the shortest path search are shown; for reference, the last row gives the number of states of the state space of the model. The state space sizes were obtained by verification of an invariantly true property in UPPAAL. The data generally validates the expected dependency on the state space size for the reduction performance. Also, all run times are within reasonable boundaries and show that the approach is feasible in practice.

6.3.2 Use-definition Chain Results

The use-definition chain reduction method was evaluated by applying it to seven different UPPAAL models and comparing it to the naive reconstruction approach. I ran two test sets for every model. The first test executed 100 times 100 random transitions of the model before reconstructing the state. The second test set executed 1000 random transitions 10 times. For the *csmacd32* model it was not always possible to execute the maximum number of transitions during simulation as the model exhibits deadlock states. Table 6.6 shows the reduction results. In the top half the results of the first test set and in the bottom half the results of the second test set are shown. All values are averages over the respective test runs. The variances between runs are small. In the experiments the reduction of transformations is between 23% and 84% while the reduction of transitions is between 0% and 39.4%. This difference mainly stems from the fact that to delete a single transition all induced transformations need to be removed. The model synthesis algorithm, however, is still unoptimized and sometimes produces unnecessary transitions. In cases where the transition reduction is higher than the transformation reduction the removal of transformations made it possible to merge multiple transitions. Interestingly, the *csmacd* models contain use-definition chains spanning the whole simulation, which prevents the removal of transitions though many transformations are irrelevant to the state. The graph-based reduction method does not have this problem as concrete variable values are taken into consideration. The adjustments to the model have a small impact on the performance as the model-checking procedure consumes most of the time. Also, compared to the

model-checking part the approach scales well with the size of the models used as the use-definition chain method only considers the simulated trace.

6.4 State Space Adaptation Summarized

In this chapter, I addressed the problem of state space adaptation and, as a prerequisite, the problem of state space reconstruction. State space reconstruction recreates a known state space of a model and modifies parts of it such that it represents an observed system correctly. The reconstruction is based on an initialization sequence that is obtained from the simulation trace of the system under surveillance. To enable online model checking the length of these traces must be kept short such that the real-time deadlines of OMC can be met. Two methods for the removal of unnecessary transformations were presented: a graph-based method and a method based on use-definition chains. These methods are of general nature and can be applied to any transformation system where a trace reduction is desired. In particular, the methods can be specialized for traces in UPPAAL's time state space in order to facilitate fast state space reconstruction and, therefore, model adaptation and online model checking.

The graph-based reduction method is based on finding shortcuts in the transformation graph by exploiting projections. Experiments show that, generally, an upper bound on the length of the transformation sequence exists and, thus, it is possible to reach a certain state of a particular UPPAAL model within limited time. Along with its good run-time performance the graph-based reduction method is practical for online model checking of real-time system.

The use-definition chain method uses data-flow techniques to track during model simulation the influence of individual transformations on the state space. An algorithm for the chain construction with reference counters was presented and its correctness was proved. A comparison to the naive reconstruction approach, which does not remove any transformations, showed that the number of DBM transformations necessary for the reconstruction of an UPPAAL time state space can be reduced by 23% to 84% resulting in initialization sequences with up to 39.4% fewer transitions.

Comparing both methods, the graph-based method is superior in its reduction results and should be preferred for online model-checking applications. However, in certain cases a model may have a very large state space and the reduction graph may consume too much memory. Then the use-definition chain method could be considered as the necessary memory is significantly less because the method operates only locally. The OMC framework allows one to change the used reduction method by calling `setReductor(Reductor)` on the underlying transformation system object that is accessible via the `UppaalSimulator` object.

In the future, now that efficient reconstruction of UPPAAL states is possible due to the developed methods, research could focus on exploring the targeted modification of such reconstruction sequences to adjust the model state to the real world. Adjusting location states or timing constraints may be useful to broaden the class of systems

that online model checking can handle. But for such modifications to be feasible understanding what kind of invalid states they introduce and how to avoid them is required.

The next chapter presents a case study that makes use of the developed OMC framework with the state space adaptation methods presented in this chapter. It is meant to demonstrate the suitability of the framework and that the included adaptation algorithms are sufficient for practical purposes.

7 Robotic Radiosurgery - A Framework Case Study

In this chapter I present a case study with the online model-checking framework developed that was carried out in conjunction with the Institute for Medical Technology of Hamburg University of Technology. The case study implements a simple OMC application that detects breathing anomalies of patients, which may arise during robotic radiosurgery, by estimating the quality of the breathing prediction model. This case study is conducted with the intention of reusing the models created in a follow-up study to formally analyze the safety of robotic radiosurgery systems in general. For the goals of this case study specifically, I was concerned with demonstrating the modeling capabilities of the developed framework, its relevance in practice, and in particular the accuracy of predicted values and artifact detection.

The chapter is organized as follows: In Section 7.1 the analyzed robotic radiosurgery system is introduced and the evaluation goal of the case study is laid out. Section 7.2 then presents how the system was modeled as an online model-checking application. The models involved are provided. Afterwards, Section 7.3 shows the performed experiments and the experiment results obtained. At last, Section 7.4 summarizes the case study and draws conclusions for the framework.

7.1 System Description

This case study implements an OMC application that detects breathing anomalies in the context of radiation therapy. Explained very briefly, in the medical domain cancer treatment with a radioactive beam that irradiates from a moving robotic arm has gained interest to increase the effectivity of the treatment [89, 90]. In this context a model of the patient's respiration has to be employed to predict the tumor position inside the patient such that the beam can correctly be targeted at the tumor. Continuous tracking of the tumor with x-rays is impossible as long exposure to x-rays harms the patient. The prediction model for the tumor position can therefore only be sparsely validated by an x-ray system. Still, the model must predict sufficiently precisely the tumor position in between validations to prevent healthy cells from being irradiated and to focus the radiation dose on the cancer cells.

In this case study such a prediction system for the chest position of a patient is developed in the form of an online model-checking application. The prediction model is validated continuously by the OMC application using the observed chest movement of the patient. This validation process effectively detects breathing artifacts like coughing or yawning by the patient. This detection enables medical staff to trigger a recalibration of the model with the internal tumor position with the x-ray system as soon as the model is invalidated. This recalibration should be performed as, when the prediction models are invalidated, the correlation between the model and the tumor position is likely to be no longer valid.

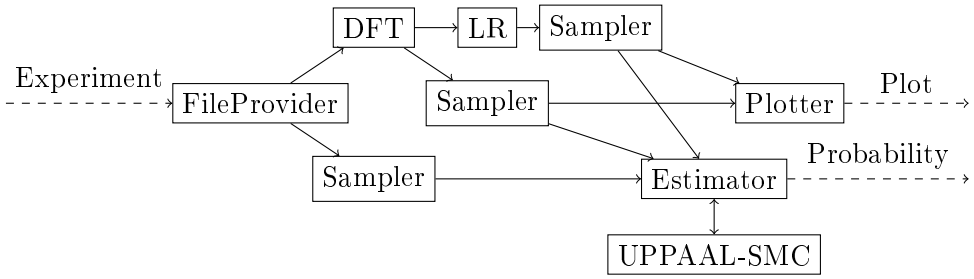


Figure 7.1: Data processing pipeline

To carry out online model checking for this system two tasks must be performed. At first, one needs to develop a model for the system in question. In this case a prediction model of a patient's respiration is created. Second, one must define the OMC application. Here the OMC framework comes in: the application is implemented in terms of the framework. The case study evaluation can then provide information on the prediction accuracy and the usefulness of the framework for the development of the system.

7.2 System Modeling

For this case study the Institute for Medical Technology at Hamburg University of Technology provided recorded data of patients' chest movements during experimental radiation therapy. The movement was tracked with a camera and three data series of the movement in the main axes were derived. The provided data includes these three data series annotated with timestamp data that allows real-time playback. The online model-checking application is fed with the data in real time. The data then is processed by the application to derive a prediction model and the validity of the models is evaluated by comparing prediction values to the real-world values.

Figure 7.1 shows the complete processing pipeline for the input data of the OMC application. The experiment data is read by the `FileProvider` component that then plays the data back in real time such that the processing units can access the data. One could replace the `FileProvider` component with a module that directly reads the sensor data during the robotic radiosurgery procedure for live evaluation results. At the end of the pipeline a probability is generated that states how likely the prediction model is to be valid. This probability is then sent to the OMC application for evaluation. Additionally, a plot of the predicted chest movement is generated for easy visual confirmation of the model parameters. As the input data includes three correlated data series the OMC application employs three such processing pipelines, one for every data series, resulting in three probabilities that are evaluated for the final verdict on the model validity. A closer look at the processing pipeline reveals that only two new components were required to create the OMC application: the `DFT` component and the `Estimator` component. The additional `Plotter` component is not required. It is only used for visualization purposes. The `DFT` component takes part

in creating the prediction model and the `Estimator` supports the model validation step. The remaining components of the processing pipeline are all provided by the framework, significantly reducing the work required to create this application.

In the rest of this section these two processing steps are presented. Subsection 7.2.1 introduces the model used to predict the chest movement and presents the processing steps to obtain the models. Afterwards, Subsection 7.2.2 explains how the OMC application validates the prediction models at runtime.

7.2.1 Breathing Prediction

As the prediction model for the chest movement of the patient I employ a simple periodic extension approach that extends a limited history of the most recent chest positions assuming that this history describes the movement trajectory in the future. The model is given by the formula

$$x(t) = d \cdot t + \sum_{i=0}^{4} c_i \cos(i \cdot f \cdot t) + s_i \sin(i \cdot f \cdot t),$$

which is a combination of a discrete Fourier series with four frequency terms and a linear component. The model parameters are the sine parameters $s_1, \ldots, s_4$, the cosine parameters $c_1, \ldots, c_4$, the baseline parameter c_0, the linear drift parameter d, and the period of the periodic extension T that yields the frequency $f = \frac{2\pi}{T}$.

For the calculation of the model parameters two components of the processing pipeline are responsible: the application-specific `DFT` component and the `LR` component, the linear regression component of the framework.

The `DFT` component has the task of computing the sine and cosine parameters as well as the baseline and the frequency of a discrete Fourier transformation of its input data series. In a first step the component estimates the breathing period and thus the breathing frequency observed in the history window. For this step it is assumed that at least two breathing cycles are present in the history window. The period is then estimated using a convolution-like process resembling autocorrelation: I assume a particular period, shift the signal by that period, and then calculate an error value for that period by building the difference between the shifted and the original signal and adding the individual errors together. To distinguish between multiples of the same period the error values of longer periods are punished with a malus value. In the end the period with the smallest error is assumed to be the breathing period and thus the breathing frequency parameter f can be estimated.

In a second step the history window is shortened such that it contains exactly the most recent period of values, i.e., all values older than one period are cut off. This step is necessary so that in the next step a smooth periodic extension can be calculated that has no jumps at the boundaries.

The shortened history window is then passed to the JTransforms library [97] that computes a fast one-dimensional Fourier transformation of the signal. From the results of the FFT call the sine and cosine parameters and the baseline parameter can be extracted.

Lastly, the linear regression component calculates the remaining drift parameter d of the model by linearly approximating the history of baseline parameters obtained from the `DFT` component. The output parameter α can be used as the drift parameter d for the prediction model, which then is completely defined.

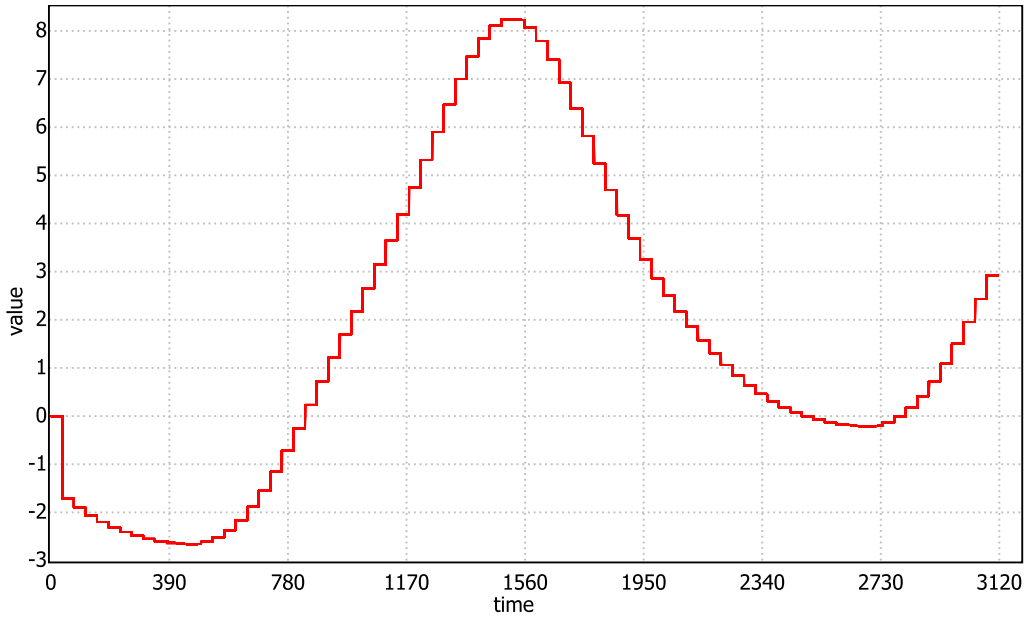
7.2.2 Model Validation

The model validation is based on validity probabilities that are estimated for the three prediction models by the `Estimator` component. The probabilities are then evaluated in the OMC application. The `Estimator` component interfaces UPPAAL-SMC, the statistical model-checking component of UPPAAL, to estimate these probabilities. It instantiates an UPPAAL model with the model parameters and then asks the model checking engine how likely the observed position of the patient's chest is in the given prediction model. The lower bound of the resulting probability interval is the confidence probability for the prediction model.

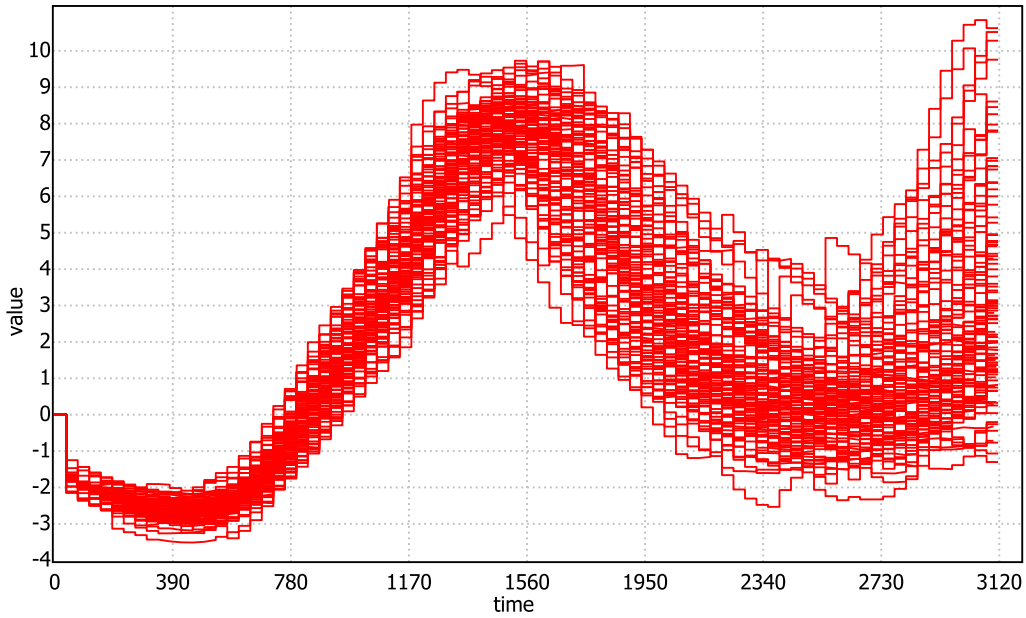
The UPPAAL model that is instantiated with the model parameters obtained from the `DFT` and `LR` components consists of five templates: the **FSTerm** template, the **Summer** template, the **CoeffModifer** template, the **TermModifier** template, and the **Timer** template. Furthermore, the UPPAAL model requires the specification of an *accuracy* parameter that defines how exact the prediction of the chest movement should be. The accuracy parameter is a floating point value in the range [0,100] where a value of 100 indicates exact simulation, i.e., the predicted value is always the same for particular model parameters. Accuracy values below 100 non-linearly introduce randomness to the simulation. Lower accuracy values allow the simulation to deviate further from the exact simulation trajectory over time. A benefit of this parameter is that the user has the opportunity to adjust the prediction results to individual patients. For example, if a patient breathes very regularly and produces nearly no breathing artifacts like coughing the model can be configured with a high accuracy value indicating that the user assumes the prediction to be quite accurate. In contrast, if the patient has a tendency to cough often or change his breathing frequency often a low accuracy value may indicate lower confidence in the prediction values. Figure 7.2 shows two plots of predicted movements with different accuracy values for the same model parameters. On the top the accuracy parameter is set to 100. In every simulation run the same prediction trajectory is generated. When the accuracy parameter is set to 70, as depicted on the bottom, the generated trajectories vary around the exact trajectory. In short, the accuracy parameter defines how fast a simulation trajectory may deviate from the exact trajectory limiting the maximum distance between them.

The details of the UPPAAL prediction model are discussed in the following paragraphs.

Instantiation and Timer The instantiation of the model with the prediction parameters is straightforward. The parameters are set by defining variables and constants



(a) Prediction with accuracy 100

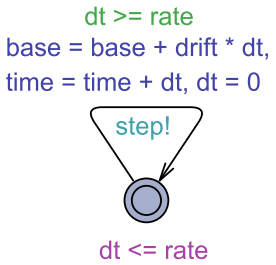


(b) Prediction with accuracy 70

Figure 7.2: Movement prediction with different accuracy values

```
const double accuracy = 70.0;
const double period = 2223.1618;
const double drift = 0.0011;
double base = 0.8553;
double a[4] = { -2.2936, -0.2597, 0.2609, -0.0639 };
double b[4] = { -3.9458, 0.7572, 0.1506, -0.1259 };
```

(a) Model instantiation



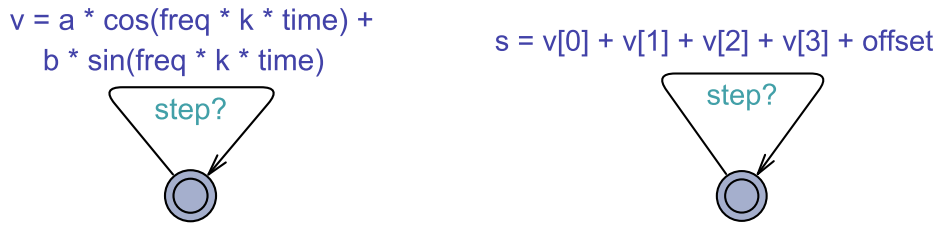
(b) Timer template

Figure 7.3: Instantiation and Timer template

of the model. As an example, an instantiation may look like depicted in Figure 7.3a, which is a snapshot of one of the later experiments. Here the accuracy is set to 70, the breathing period is about 2.2 seconds, the drift is slightly positive, the baseline about 0.85 cm. To reduce the state space of the model the complete prediction is synchronized by a single **Timer** template. This template is shown in Figure 7.3b. Periodically, every `dt` time units, the next prediction value is calculated. The template then adjusts the baseline by the drift value, updates the current time, and informs all slave templates via broadcast synchronization.

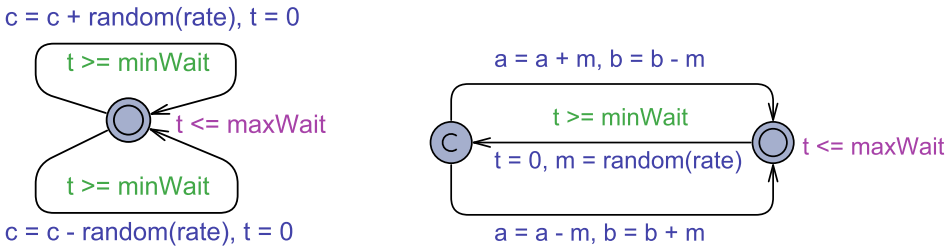
FSTerm and Summer The actual calculation of the prediction model is carried out by the **FSTerm** and **Summer** templates. Both are depicted in Figure 7.4. The **FSTerm** template represents a single term in the sum of the sine and cosine values. Thus, this template is instantiated four times, once for every multiple of the base frequency. When triggered, the template calculates a new value for its sum based on the current parameters. The **Summer** template is only instantiated once and is responsible for producing the final prediction value. It just sums the values created by the **FSTerm** instances and adds an offset: the drift-adjusted baseline value.

CoeffModifier and TermModifier The **CoeffModifier** template and the **TermModifier** template are both used for the variation of the predicted values over time. How much they vary the predicted values depends on the setting of the accuracy parameter. The templates are shown in Figure 7.5. The modifier template for coefficients is instantiated twice: for the frequency parameter and for the baseline parameter. They are modified by repeatedly adding or subtracting random values whose maximum values depend on the accuracy parameter. The rate of modification is configured during model instantiation. The **TermModifier** template works in a similar way and adjusts the values of a sine-cosine term. It is instantiated four times: once for each frequency multiplier of the prediction model. Again, the values are repeatedly changed by a random value that depends on the accuracy parameter.



(a) FSTerm template (b) Summer template

Figure 7.4: Prediction calculation templates



(a) CoeffModifier template (b) TermModifier template

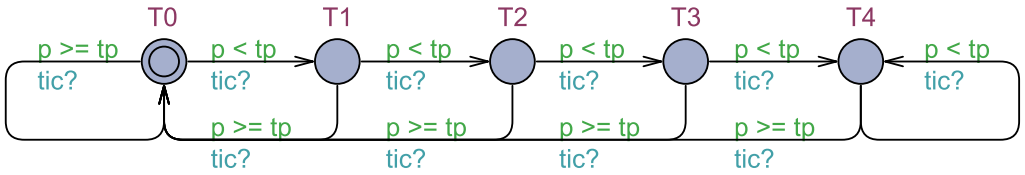
Figure 7.5: Modification templates

Using the UPPAAL model the estimation of a validity probability for a prediction model is straightforward. If we create a model at time t_{M} and observe the value x_{o} at time t_{o}, $t_{\mathrm{o}} > t_{\mathrm{M}}$, we can instruct the statistical model checker to estimate how likely we were to observe that value by evaluating the property

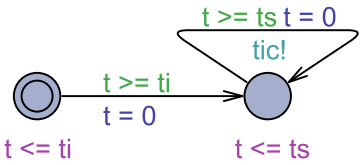
$$Pr[\Diamond_{\leq t_{\mathrm{o}}-t_{\mathrm{M}}+t_{+}}\ t_{\mathrm{o}} - t_{-} \leq t_{\mathrm{p}} \leq t_{\mathrm{o}} + t_{+} \wedge x_{\mathrm{o}} - x_{-} \leq x_{\mathrm{p}} \leq x_{\mathrm{o}} + x_{+}]$$

where t_{+}, t_{-}, x_{+}, and x_{-} are range parameters defining a rectangle around the observed time/value pair, $(t_{\mathrm{o}}, x_{\mathrm{o}})$, and $(t_{\mathrm{p}}, x_{\mathrm{p}})$ are the predicted values. This property estimates the probability with which a value in the rectangle around $(t_{\mathrm{o}}, x_{\mathrm{o}})$ is produced by the prediction model. UPPAAL-SMC runs multiple simulations and checks for every run if the predicted values are in the rectangle. By counting the positive and negative results the probability of observing the values from the real world can be estimated. For more details on statistical model checking see Subsection 3.2.2. The resulting probability can then be used to draw conclusions on the accuracy of the model for the currently observed chest movement. Note that the resulting probability depends on the accuracy parameter of the model: if the accuracy parameter is low, also the probability to predict the observed value is low.

For the final verdict on the model validity I employ a simple threshold-based agreement protocol: if the validity probabilities for all three input data series are below a threshold for a certain time I assume the model is invalid and has to be replaced. The threshold is the second parameter of the analysis system in addition to the accuracy parameter of the prediction. This evaluation process is implemented in an UPPAAL



(a) Tier template



(b) Timer template

Figure 7.6: Evaluation model

model such that the periodic verification task of the online model-checking application performs the evaluation. The underlying UPPAAL model has just two templates: a **Timer** template that is instantiated once and a **Tier** template that is instantiated once for every data series. The templates are depicted in Figure 7.6. In contrast to the **Timer** template from the prediction model that models the prediction time (see Fig. 7.3b), this **Timer** template sends periodic signals to the **Tier** templates after an initial waiting phase in order to synchronize their transitions. For the OMC framework this approach reduces the state space as only one clock is used instead of three, one for every **Tier** template. The **Tier** template consists of five locations, `T0` to `T4`, where `T0` is the initial location. Every time the synchronization signal is received the automaton evaluates whether or not the current validity probability `p` is below the threshold `tp`. If the probability is less, the automaton advances one tier, otherwise its tier level is reset to `T0`. The intuition behind the tiers is to capture how often in a row the prediction probability of a data series falls below the threshold, giving some flexibility on deciding when the prediction model should be rejected. For example, one could reject the model every time the probability is lower than the threshold, or only if it is lower multiple times in a row.

The online model-checking application simulates this model concurrently to the system. In the model adaptation step the application adjusts the probability variables `p` of the **Tier** template instances to the estimated values from the data processing pipeline. This is where the OMC framework significantly reduces work by providing a unified adaptation approach using the `Variable` class for the probabilities. After the adaptation UPPAAL is queried for the final verification verdict. The verdict is obtained by checking a simple reachability property of the model with the OMC framework:

```
E<> ((A1.T3 || A1.T4) && (A2.T3 || A2.T4) && (A3.T3 || A3.T4))
```

This formula is satisfied if for all three data series (`A1`, `A2`, `A3`) it is possible to reach

at least tier 3 within the next verification interval defined in the OMC application and the **Timer** template. This condition is equivalent to requiring that all validation probabilities are less than the threshold two consecutive times because then the third tier is already reachable. Note that the location `T4` could be removed from the **Tier** template and the property could be simplified by removing the references to `T4`. We opted to keep the location in the template to provide more flexibility when defining how many consecutive times the validation probability must fall below the threshold. Moreover, note that the OMC framework here automatically rewrites the property such that the property is only evaluated for a limited time scope. The framework effectively performs bounded model checking. Otherwise, in the unbounded case, as soon as one of the p probabilities is set to a value less than the threshold all locations become reachable immediately and the history information, i.e., the starting state of the verification that encodes how often the threshold was previously undercut, of the system has no influence on the verification.

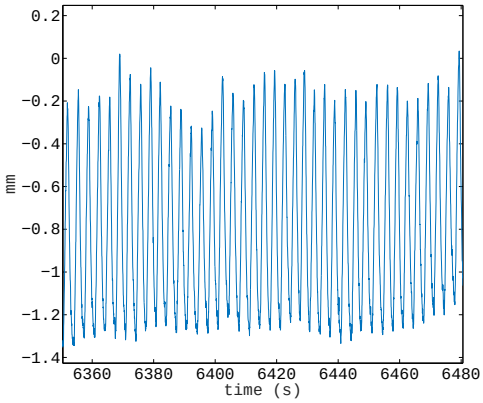
7.3 Experiments and Evaluation

To evaluate the chest movement prediction and the resulting breathing anomaly detection rate of the system model I used in the OMC application I carried out several experiments. In an initial set of simulations the chest movement traces of six patients that exhibit different breathing characteristics were used as the input data to the automatic model generation and model validation. In a second experiment, a single patient trace was chosen and simulated for different accuracy and threshold parameters. All patient traces used are about one and a half hours long and they were simulated in real time. The verification scope, i.e., the depth of the bounded model checking, was set to 6 seconds. Accordingly, the validity of the model was evaluated every 3 seconds to obtain overlapping verification intervals. The size of the detection rectangle was ± 175 ms and ± 0.35 mm.

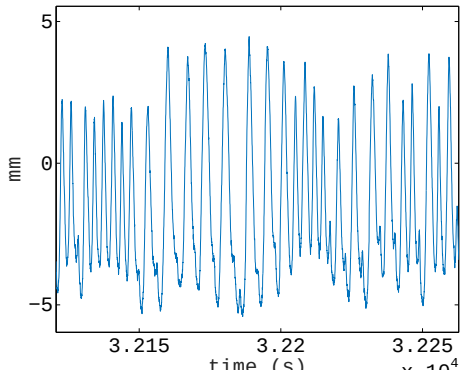
In the following, information on the initial experiments and the patient data is given in Subsection 7.3.1. Subsection 7.3.2 then presents the results of the parameter variation.

7.3.1 Initial Simulations

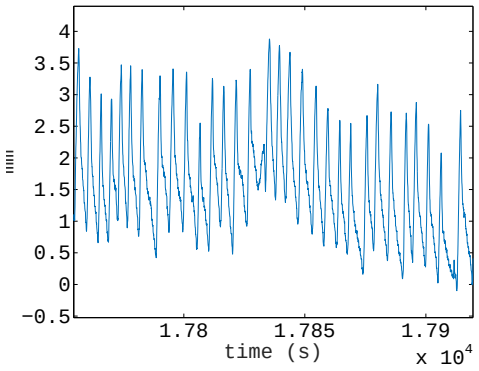
The goal of the initial simulations was to determine if the developed prediction model is general enough that it can be used to detect breathing anomalies patients that exhibit different breathing patterns. During the initial simulations the accuracy parameter was set to 80 and the detection threshold probability was set to 10%, i.e., if the probability that an observed chest position is predicted by the model is less than 10% the model is invalidated. This low threshold only detects severe differences between prediction and observation. Six different patient data sets with different breathing habits were chosen and simulated. The simulation results were recorded together with the model validation verdicts. The different breathing characteristics of the patients are shown in representative plots of the respiratory chest movement in Figure 7.7. Patient



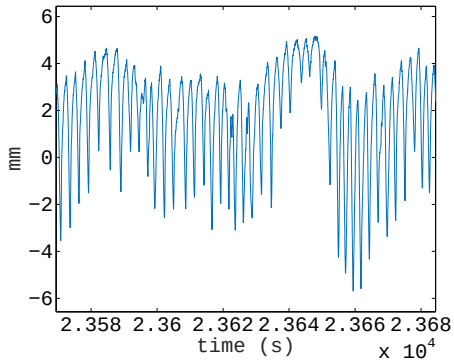
(a) Patient **DB14_Fx3**



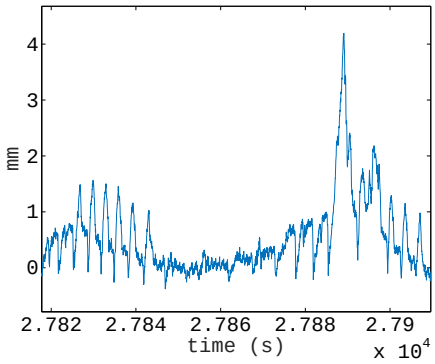
(b) Patient **DB130_Fx4**



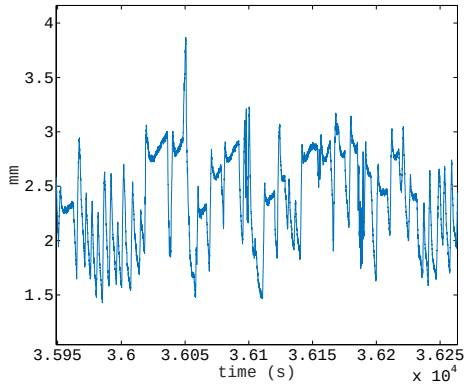
(c) Patient **DB114_Fx1**



(d) Patient **DB01_Fx1**



(e) Patient **DB106_Fx3**



(f) Patient **DB104_Fx3**

Figure 7.7: Characteristic breathing movements of patients analyzed

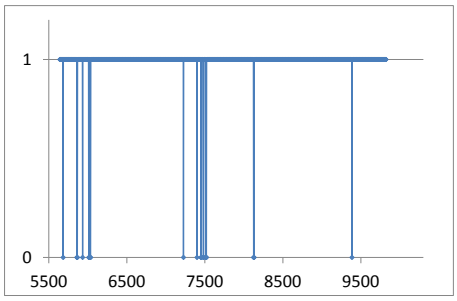
DB14_Fx3, depicted in Figure 7.7a, has the most stable respiration. Frequency and amplitude vary only by small amounts throughout the recording. The respiration of patient **DB130_Fx4**, depicted in Figure 7.7b, is slightly irregular: small frequency changes and amplitude changes occur. The same is true for patient **DB114_Fx1** (Fig. 7.7c) and patient **DB01_Fx1** (Fig. 7.7d). But here the variations are a bit more drastic and occur more often. The last two patients, patient **DB106_Fx3** (Fig. 7.7e) and patient **DB104_Fx3** (Fig. 7.7f), have very irregular breathing patterns interleaved with times of regular breathing. Their respiration sometimes exhibits significant jumps in amplitude and frequency. Occasionally, respiration even comes to a halt for a limited period of time. Predicting the breathing patterns of those patients is very difficult and poses a hard challenge for the OMC application.

Plots of the detected anomalies are depicted in Figure 7.8. The time is shown on the x-axis and the detection result is shown on the y-axis. An anomaly was detected if the value drops to a value of 0. The plot for patient **DB14_Fx3** has very few anomalies as expected by its regular breathing pattern. However, for the next three patients, patient **DB130_Fx3**, patient **DB114_Fx1**, and patient **DB01_Fx1**, many anomalies were detected, although their respiration is fairly regular. In hindsight those detections can be attributed to the fairly aggressive accuracy parameter setting of 80. The result for patient **DB106_Fx3** is also remarkable: although the breathing patterns are highly irregular relatively few anomalies have been detected. Closer inspection of the patient's data shows that often fairly regular patterns can be observed. Just because the patient does not have an ideal respiration does not imply that the patient's breathing has no pattern. The result for the patient **DB104_Fx3**, finally, is as expected: very irregular breathing with high variance produces many detections. For actual treatment one may want to exclude such patients as robotic radiosurgery methods for radiation therapy would probably fail.

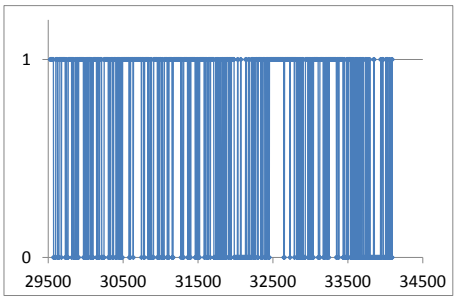
As a conclusion to the initial experiments one can say that the parameters of the model employed in the OMC approach must be tuned according to the breathing characteristics that a patient exhibits if breathing artifacts must be detected reliably. However, it appears that even then patients exist whose respiration can not be predicted with adequate accuracy with my model. For those patients OMC can not yield an improvement and they probably should be excluded from robotic radiosurgery with automatic breathing compensation. In a next set of experiments I studied the influence of the parameters on the detection results as the results for the patients **DB130_Fx4**, **DB114_Fx1**, and **DB01_Fx1** were inconclusive because of inadequate configuration of the method parameters and the result for the patient **DB106_Fx3** was surprisingly good in spite of the configuration.

7.3.2 Parameter Variation

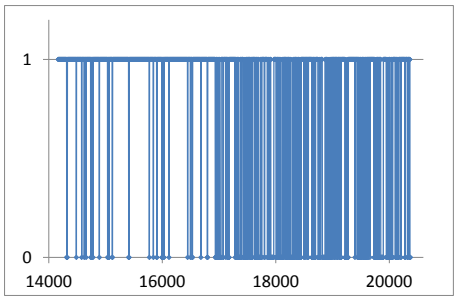
For the parameter variation experiments the patient trace **DB106_Fx3** was chosen as it promised the most diverse and insightful results. The accuracy parameter was varied from 90 to 50 in steps of 10 and the threshold probability was varied from 10% to 25% in steps of 5%. Table 7.1 shows the number of detected breathing anomalies for



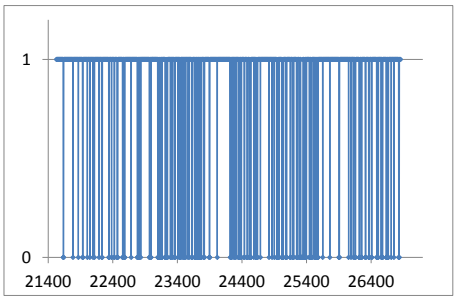
(a) Result for patient **DB14_Fx3**



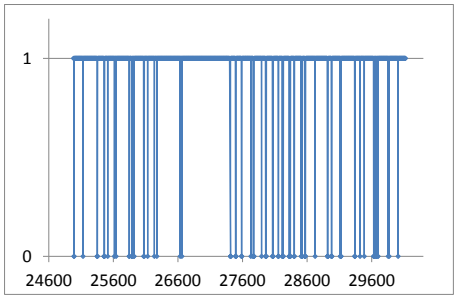
(b) Result for patient **DB130_Fx4**



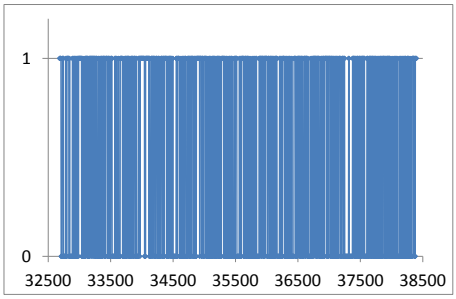
(c) Result for patient **DB114_Fx1**



(d) Result for patient **DB01_Fx1**



(e) Result for patient **DB106_Fx3**



(f) Result for patient **DB104_Fx3**

Figure 7.8: Anomaly detection results for patients analyzed

Table 7.1: Number of detected breathing anomalies

Threshold	Accuracy				
	90	80	70	60	50
10%	276	116	80	49	42
15%	218	163	114	60	67
20%	300	198	113	97	82
25%	363	232	166	153	156

Table 7.2: Validity probability statistics

	Accuracy				
	90	80	70	60	50
Average	48.1%	37.0%	33.2%	30.9%	29.3%
Deviation	27.3%	18.6%	13.6%	10.8%	9.1%

every parameter combination. Two general trends can be observed: as the threshold probability increases more artifacts are detected, and as the prediction accuracy is reduced the number of detections decreases. Both trends are expected: if the threshold probability is set to a high value more values satisfy the threshold check and thus more events are detected. The same is true if one increases the accuracy. A high prediction accuracy results in fewer values in the predicted corridor, which increases the number of observed values that have a low probability because they are outside of the prediction corridor. Thus, for the same threshold more artifacts are detected. This behavior of the validity probability can also be observed if one takes a look at the average of the observed validity probabilities and the deviation of those during the experiments. Table 7.2 shows these results. Only the accuracy parameter is relevant here as the accuracy defines the prediction; the threshold probability defines the cut-off line for these probabilities.

For a more thorough evaluation we calculated sensitivity (true positives), specificity (true negatives), false positives, false negatives, and overall correct, i.e., all true, detection rates obtained by the OMC approach for the experiments. The ground truth for this evaluation was obtained by applying a static analysis to the patient data. The static analysis distinguishes and marks two types of anomalies that form the majority of the events in the data set: *Short Term Amplification Anomalies* (STAA) and *Baseline Shifts* (BLS). The STAA detection rate is specified by a constant parameter c_{STAA}: an STAA is detected if one of the following cases occurs:

- The absolute difference between the current and the previous breathing amplitude exceeds the amplitudes of surrounding breathing cycles weighted by c_{STAA}.
- The absolute difference between the current and the previous breathing baseline exceeds the amplitude of surrounding breathing cycles weighted by c_{STAA}.
- The absolute difference between the current breathing amplitude and the mean

Table 7.3: Quality metrics of OMC approach ($\triangle t = 3$, $c_{\text{STAA}} = c_{\text{BLSHIFT}} = 0.5$)

accuracy parameter	threshold parameter	sensitivity	false negatives	specificity	false positives	accuracy (over all)
50	10	76.47	23.53	69.05	30.95	76.29
50	15	76.53	23.47	68.66	31.34	76.23
50	20	77.35	22.65	59.76	40.24	76.53
50	25	78.07	21.93	51.90	48.10	75.71
60	10	76.63	23.37	69.39	30.61	76.43
60	15	77.09	22.91	70.00	30.00	76.85
60	20	77.47	22.53	56.70	43.30	76.32
60	25	78.48	21.52	53.90	46.10	76.32
70	10	76.45	23.55	55.42	44.58	75.44
70	15	78.30	21.70	65.79	34.21	77.49
70	20	77.57	22.43	56.64	43.36	76.21
70	25	77.74	22.26	44.58	55.42	74.57
80	10	77.74	22.26	50.00	50.00	75.89
80	15	77.54	22.46	52.76	47.24	75.23
80	20	78.57	21.43	48.99	51.01	75.20
80	25	78.34	21.66	42.67	57.33	73.58
90	10	78.28	21.72	40.22	59.78	72.24
90	15	77.81	22.19	40.91	59.09	73.17
90	20	78.40	21.60	38.74	61.26	71.53
90	25	79.01	20.99	38.74	61.26	70.66

of the amplitudes of surrounding breathing cycles exceeds the mean multiplied by c_{STAA}.

For baseline shifts the constant parameter c_{BLS} defines when baseline shifts are detection by the static approach. A baseline shift is detected if the breathing baseline of the next minute and the breathing baseline of the previous minute differ more than the average amplitude during that time multiplied by c_{BLS}.

For the purpose of comparison to the OMC approach c_{STAA} and c_{BLS} were set to 0.5 as a deviation of 50% to the surrounding environment is assumed to be a significant change. A visual validation of the detected events with the chest movement plot supports this choice as the detected artifacts correspond well with the behavior of the chest movement plot. Furthermore, a comparison time window $\triangle t$ was chosen to define matches between the OMC approach and the static analysis: if the OMC approach detects an anomaly at time t then the static analysis must detect an artifact within $[t - \triangle t, t]$, i.e., in a time window $\triangle t$ before the OMC detection as the OMC delay shifts the timestamps of the detected anomalies back by the verification interval. For the comparison $\triangle t$ was thus set to 3 seconds as that was the online model-checking verification interval during the experiments.

Table 7.3 shows the quality metrics obtained by this comparison. The overall ac-

Table 7.4: Compensated quality metrics ($\triangle t = 3$, $c_{\mathrm{STAA}} = c_{\mathrm{BLS}} = 0.5$)

accuracy parameter	threshold parameter	sensitivity	false negatives	specificity	false positives	accuracy (over all)
50	10	92.65	7.35	69.05	30.95	92.08
50	15	92.73	7.27	68.66	31.34	91.81
50	20	92.81	7.19	59.76	40.24	91.26
50	25	92.92	7.08	51.90	48.10	89.22
60	10	92.60	7.40	69.39	30.61	91.95
60	15	92.64	7.36	70.00	30.00	91.86
60	20	92.73	7.27	56.70	43.30	90.73
60	25	92.97	7.03	53.90	46.10	89.53
70	10	92.53	7.47	55.42	44.58	90.76
70	15	92.91	7.09	65.79	34.21	91.14
70	20	93.15	6.85	56.64	43.36	90.79
70	25	92.62	7.38	44.58	55.42	88.03
80	10	92.91	7.09	50.00	50.00	90.05
80	15	92.74	7.26	52.76	47.24	89.02
80	20	92.86	7.14	48.99	51.01	87.86
80	25	92.76	7.24	42.67	57.33	86.07
90	10	92.62	7.38	40.22	59.78	84.31
90	15	93.08	6.92	40.91	59.09	86.53
90	20	92.85	7.15	38.74	61.26	83.47
90	25	92.81	7.19	38.74	61.26	81.60

curacy is at a solid 70% throughout all experiments. For the sensitivity and the corresponding false negative rate accuracy and threshold parameters seem to have little influence on the detection rate. This may stem from the fact that true breathing anomalies produce significant change in the observed values and the prediction model is likely to fail independently from the parameter settings. For specificity and its false positive rate however an influence of the system parameters can be observed. As expected a low threshold parameter decreases the rate of false positives as an observed value has to be more distant from the predicted value to fall below the threshold. The accuracy parameter has the same expected effect: a high accuracy parameter leads to many false positives as many observed, borderline acceptable values leave the prediction corridor.

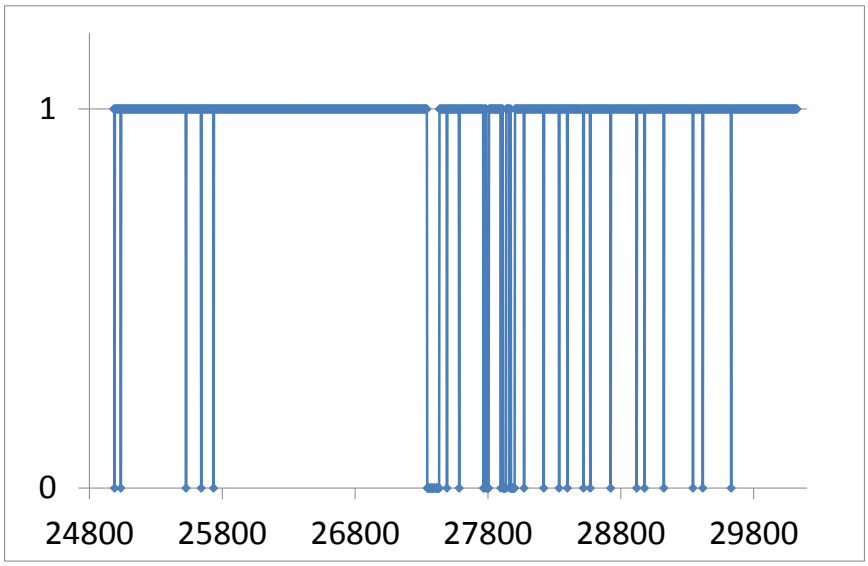
Closer examination of the comparison method shows that this comparison method is not completely fair to the online model-checking approach. The OMC approach was designed to detect anomalies as soon as they happen: once at the beginning. As such, the OMC approach generally marks long breathing artifacts only once. However, in the general case the static analysis marks the complete time interval of the anomaly. This process skews the results as later in an anomaly interval the static approach produces an event while the OMC approach does not. The comparison of these events

fails and a false negative detection is attributed to the OMC approach. In an attempt to compensate for this discrepancy we grouped in a second iteration the results of the static analysis and checked if the OMC approach detected the intervals defined by the groups. For example, assume an anomaly of 3 seconds that has been detected by the static approach three times, and only once in the beginning by the OMC approach. Instead of treating the three detections by the static approach as three individual events we aggregate them to a single continuous event and check if the OMC approach produced a detection event somewhere during the event. The resulting effects on the quality metrics are displayed in Table 7.4: the sensitivity rate increases from about 77% in the uncompensated case to about 93%. Accordingly, the false negative rate decreases to a level of about 7%. In total, the overall accuracy increases from 75% to 92% , i.e., 17% more model validations are correctly identified as correctly positive or negative. For an easy visual evaluation of the results, Figure 7.9 shows when the static analysis found artifacts and when the best configured OMC approach detected artifacts. An anomaly was detected when the graph falls from a value of 1 down to 0. As one can see both graphs are quite similar and the detection flanks occur at similar times.

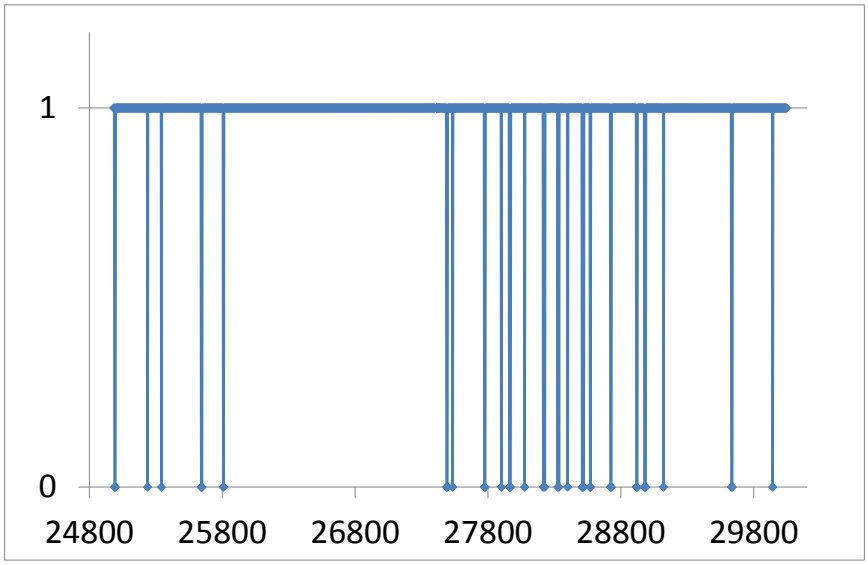
The detection results obtained are promising for future studies using online model checking in the medical domain. In addition to an overall good detection rate of about 92% the false negative rate is quite low at 7%. This fact is important as in many applications false negatives have more severe consequences than false negatives. This is especially true for medical applications: falsely declaring a condition that is harmful to a patient as a normal condition is disastrous as the patient will generally not be treated how it is required. A false positive, however, induces further examinations of the patient. Those will most likely invalidate the previously detected false verdict. However, the detection rates achieved in this study still need to be improved to make the method practical for concrete medical applications. The explicit nature of the prediction models may help to achieve this goal as more appropriate prediction models can be incorporated easily and changes are easy to implement. The development of more sophisticated breathing models, maybe even patient-tailored ones, seems feasible in this way in the OMC context.

7.4 Conclusion

This chapter presented a case study that applied the online model-checking technique with the developed UPPAAL OMC framework to the medical treatment method of radiation therapy. Medical research currently explores employing robotic radiosurgery devices to increase the efficiency of the treatment and the safety of the patients. When treating lung cancers this approach requires that the chest movement of the patient is observed and correlated to the tumor position inside the patient. The presented online model-checking application determines if the prediction model for the chest movement is plausible, which is equivalent to detecting breathing artifacts. It periodically calculates probabilities that characterize how likely an actually observed



(a) Reference anomalies detected by static analysis



(b) Anomalies detected by OMC approach (accuracy 60, threshold 10%)

Figure 7.9: Visualization of detected artifacts

value is produced by the model and then aggregates these probabilities with an agreement system to obtain a final verdict on the model validity. In case the model is deemed implausible measures can be taken that benefit the patient. For example, a resynchronization could be triggered immediately instead of irradiating the patient inaccurately. The OMC application was evaluated by simulating multiple patient traces with various parameter combinations. With appropriate configuration an overall correct detection rate of 92% was achieved. The false negative rate was approximately 7%. These results show that the OMC approach has potential in the medical domain and that it can be carried out with the developed framework in practice. The result can be generalized as the approach only depends on the developed models, which are application-specific, and the contributions of the framework are always the same for all online model-checking applications. For the framework this case study shows that the framework is capable of performing real-time online model checking, i.e., it can deal with the requirements imposed by medical modeling. Also the framework supplied useful components to ease the development process of the application. Only two new components had to be implemented in addition to the application specification. In the future, the explicit models used for breathing prediction could be replaced with more detailed models to further decrease the error rates. Furthermore, the prediction model can potentially be embedded in a formal model of the complete robotic radiosurgery process to verify safety properties for the patient or to formally derive safety margins for the irradiation.

The next chapter concludes this dissertation by summarizing its parts, discussing its contributions, and suggesting future research topics.

8 Concluding Discussion

In this final chapter I summarize the dissertation, discuss and reflect on the results, and identify interesting research questions to further advance the research field of online model checking.

8.1 Summary

This dissertation explored the online model-checking technique in the context of medical cyber-physical systems. Online model checking is a verification technique that dynamically, at runtime, provides time-bounded safety guarantees by periodically applying classic, static model checking to verify properties of a system model. Online model checking broadens the class of systems that can formally be analyzed as it has several beneficial features.

- Online model checking reduces the state space that needs to be analyzed during verification as only a smaller part of the model must be searched within a verification period. This reduction enables the verification of complex models that often arise in the cyber-physical systems context, which potentially results in an overall increase in safety in such systems.
- Furthermore, online model checking not necessarily needs to verify the same system model in every verification period: in addition to adjusting the starting state of the verification to the observed real-world state the model itself can be modified. This capability enables the application of formal methods to systems where accurate modeling is difficult. In the medical domain such modeling difficulties often arise, especially in closed-loop systems, because they often involve a model of a patient that characterizes the patients' reactions to medical treatment. Accurate long-term modeling of the physiologic features of humans is still ongoing research and thus online model checking helps to tackle such systems in the meantime as it only depends on short-term predictions.
- Moreover, in contrast to the original runtime verification approach online model checking is a proactive verification technique. The use of an explicit system model allows predictions of the system's behavior in the future and thus the verifier can check in advance whether or not the system's safety is compromised. The resulting grace periods may then, e.g., be used to shut down a system before it can harm a patient.

The presented work implements the online model-checking approach by interfacing the well-known model-checking tool UPPAAL, which uses timed automata as the underlying modeling formalism. Initially, the theoretical foundations of timed automata, model checking, and its online variant were presented. Then, a preliminary case study that implements a scenario of a patient undergoing laser tracheotomy showed that

carrying out online model checking with UPPAAL is indeed feasible in practice. Subsequently, I presented a Java framework to develop arbitrary online model-checking applications for the UPPAAL tool. The framework features a data processing pipeline that may interface sensors to obtain data from the physical world, a model simulator that runs the system model in real time synchronously to the real system, a verification interface to analyze properties of the system with UPPAAL, and an automatic state space reconstruction module that provides capabilities to adapt the system model. For the state space reconstruction two transformation reduction algorithms were presented that decrease the time for the reconstruction considerably. This performance gain enables automatic state space reconstruction within the deadlines of online model checking. Both algorithms, one based on shortest paths in graphs and projections, and one based on use-definition chains, were subject to a correctness analysis supported by proofs and correctness arguments. To test the framework in practice a concluding case study was discussed that analyzes the treatment of cancer with radiation therapy by robotic radiosurgery devices. The resulting online model-checking application showed promising results for the detection of breathing anomalies, which is a necessity for compensating the patient's movement automatically when targeting the tumor cells.

8.2 Discussion and Future Research

Both case studies carried out in this dissertation show that there is a benefit in using online model checking for safety assurances. However, even though this dissertation's contributions allow the development of basic online model-checking applications it only aims at providing a foundation for OMC applications. Further research in the field of online model checking is necessary to realize the full potential of OMC.

The developed online model-checking framework for UPPAAL, to begin with, can be subject to research in several dimensions.

1. **State space adjustments**. While the framework is functional for simple applications that only change the current state by adjusting data variables of the state space to real-world values, restricting the OMC approach to such models limits the potential of the OMC approach. Exploring other techniques to adapt the UPPAAL models and understanding their effects on the model state space may lead to modeling approaches for systems that are still difficult to tackle. Potential adaptations can be divided into two categories: modifications to the current state and modifications to the model itself. In the following I give a short overview to potential adaptation techniques, means that could be used to implement them, and problems that need to be solved for success.
    a) *State adjustment: clock valuations.* During simulation it may happen that a clock value does not match the timing behavior in the real world. For instance, some process was expected to last 5 time units, but only needed 4 to complete. In such a case the model state fails to reflect the real-world timing for the remainder of the application lifetime. Directly changing the

entries of the difference bound matrix representing the time state is currently not possible, but could be introduced to UPPAAL in the future. In the meantime it would be possible to match the real-world timing by modifying the initialization sequence obtained by the state space reconstruction. By modifying individual transitions or injecting new transitions one could set the time state to the desired values. However, when doing so one must be aware that this approach may have undesired side effects on other DBM entries as all DBM operations modify multiple values by default. In some cases this may be the correct behavior but one needs to be aware if this is not the case and handle the situation correctly. For example, it might be possible to compensate the undesired side effects by carefully crafting the initialization sequence to include compensating transitions. However, even if an algorithm for accurate modifications to the time state is discovered and its implications are understood a different problem may occur: the real-world state may not be valid in the current model. Thus, state adjustments may induce model modifications to actually represent the real-world state correctly.

b) *State adjustment: automata locations.* This problem of creating an invalid state for the model is also apparent when one needs to change the current location state. Such a need may arise when a model is in a location that represents that a lamp is on while in the real world the lamp is already turned off, e.g., due to a timing mismatch as above. Unfortunately, just changing the location may result in the violation of invariants of the new location and implicit requirements on the state may get lost. For instance, a location may require that every ingoing edge resets a certain clock but when modifying the location directly these state changes are not performed. Implementing the actual location change is quite easy in the framework, but ensuring that valid states are created is difficult. In general, I assume that a location change will induce a clock valuation change as well with all the complications such a change introduces. Future research could thus explore when direct location adjustments are safe to perform and when additional changes are required.

c) *Model modification: timing constraint.* As shown above adjustments to the state may require modifications of the model. However, changes to the model may also be desired when new information on the real-world system is gained. For instance, the timing behavior of a system may vary over time. The modification of the model in question, e.g., changing a constant in an invariant expression, only requires quite simple extensions of the framework. The main research question here is when such modifications are useful or even required for a certain class of systems. In the medical domain the modeling of a patient could benefit from such adaptations: in addition to adjusting the current simulated state to the real state of the patient one could also employ some kind of machine-learning approach to

fit the model to particular patients.

d) *Model modification: automata topology.* Changing the complete topology of an automata by introducing new locations, changing transitions, or removing parts would also be a possible model modification. In the context of the OMC framework implementing such changes would require quite some work but no fundamental changes. The resulting model synthesis approach could be useful if parts of a system are black boxes, i.e., components whose internal behavior is unknown and only external influence can be observed. In such situations again machine-learning approaches or domain-specific approaches could be explored to learn how to refine and generalize the component model. For instance, in a model a location may exist that is used when a variable is in the range 1-100, but during the observation of the black box it becomes clear that its behavior differs in the ranges 1-50 and 51-100. Then an appropriate algorithm would split the location into two locations to distinguish between the different behaviors.

2. **Reconstruction algorithms**. All these modifications would make the OMC approach applicable to a broad variety of systems bringing analysis with formal methods into practice. However, not only the adjustment of states can be explored but also the developed algorithms for state space reconstruction could be improved. Ideally a constructive approach is desirable that constructs arbitrary DBM configurations with minimal operations. Such an algorithm would also tie in to the clock valuation adjustments mentioned above as this reconstruction algorithm could calculate necessary injections and modifications. For the development of such an algorithm the first step would be to analyze the vector space created by the DBM operations. Then, when it is clear how valid DBMs look like, it may be possible to derive new operations from the DBM operations that can be used to build a particular DBM.

3. **Real-time simulation**. Another aspect of the OMC framework that could be explored in future research is how the real-time simulation is handled. At the moment timing is deterministic: every clock advances until an invariant prevents further increases and then the next transition is taken. Although, in addition to this simulation process, a prototypical interface is present that can be used to trigger transitions early from the outside, this system is far from perfect. Identifying a better way to couple the real system to the model and allowing different interactions enables one to formulate more complex models and thus more complex systems could be analyzed. The actual implementation requires some work but the challenge is identifying applicable simulation strategies and maybe even make them selectable for applications.

4. **Application builder**. Lastly, I think it is desirable to extend the framework GUI to include an application builder component: a graphical component where the user can select components, connect them, define data series, and interface sensor data to build an OMC application. From an academic standpoint this

work may not be very interesting but increasing the usability of the framework is necessary for the development of a mature OMC tool and thus promoting formal methods in practice.

Moreover, the application of the OMC framework should also be analyzed in detail in the future. Although the radiation therapy case study as an example application of the framework shows the framework's capabilities it would be nice to have more practical experience, preferably also from different domains. The example case study itself also presents opportunities to expand the research carried out in this thesis. The OMC approach is used to validate a dynamic model of the respiration of a patient and it detects anomalies in the breathing pattern that a patient exerts. One could extend the OMC application by not only predicting the breathing movement but by also drawing conclusions from it by incorporating other components of the robotic radiosurgery system. The goal should be to obtain a complete formal model of the system. Then it might become possible to derive safety margins from the model, which could lead to a more accurate irradiation of the patient. As a consequence a patient would be subject to better treatment.

In conclusion, I am confident that OMC applications can not only contribute to increase the patient safety in this robotic radiosurgery scenario but the OMC approach is capable of increasing the safety of critical cyber-physical systems in general. Further research in this direction can expand on the online model-checking foundation laid out in this dissertation, facilitating the application of formal methods in practice even further.

Bibliography

[1] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and Abstraction of the Biometric Passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer Berlin Heidelberg, 2010.

[2] Marwan M. Abdeen, Wolfram Kahl, and Tom Maibaum. FDA: Between Process and Product Evaluation. In *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007)*, pages 181–186. IEEE, 2007.

[3] Abdullah Al-Nayeem, Lui Sha, Darren D. Cofer, and Steven P. Miller. Pattern-Based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems. In *Proceedings of the 3rd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '12*, pages 65–74. IEEE, 2012.

[4] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking for Real-Time Systems. In *Fifth Annual IEEE Symposium on Logic in Computer Science LICS '90*, pages 414–425, 1990.

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

[6] Rajeev Alur and David Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] Rajeev Alur, Salvatore La Torre, and George J. Pappas. Optimal Paths in Weighted Timed Automata. In Maria Domenica Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 49–62. Springer Berlin Heidelberg, 2001.

[8] David Arney, Miroslav Pajic, Julian M. Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Toward Patient Safety in Closed-Loop Medical Device Systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '10*, pages 139–148. ACM New York, NY, USA, 2010.

[9] Eugene Asarin and Oded Maler. As Soon as Possible: Time Optimal Control for Timed Automata. In Frits W Vaandrager and Jan H Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin Heidelberg, 1999.

[10] Stanley Bak, Karthik Manamcheri, Sayan Mitra, and Marco Caccamo. Sandboxing Controllers for Cyber-Physical Systems. In *Proceedings of the 2nd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '11*, pages 3–12. IEEE, 2011.

[11] Ezio Bartocci, Radu Grosu, Panagiotis Katsaros, C.R. Ramakrishnan, and Scott A. Smolka. Model Repair for Probabilistic Systems. In Parash Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2011.

[12] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou Somers, Egbert Teeselink, Nikola Trcka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 90–105. Springer Berlin Heidelberg, 2010.

[13] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL Implementation Secrets. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2002.

[14] Gerd Behrmann, Patricia Bouyer, Kim G Larsen, and Radek Pelánek. Lower and Upper Bounds in Zone Based Abstractions of Timed Automata. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 312–326. Springer Berlin Heidelberg, 2004.

[15] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal 4.0. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–48. Springer Berlin Heidelberg, 2006. Updated version of the original paper from 2004.

[16] Gerd Behrmann, Kim G Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis using Clock Difference Diagrams. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer Berlin Heidelberg, 1999.

[17] Johan Bengtsson. Reducing Memory Usage in Symbolic State-Space Exploration for Timed Systems. Technical Report 2001-009, Department of Information Technology, Uppsala University, 2001.

[18] Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.

[19] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated Verification of an Audio Control Protocol Using UPPAAL. *The Journal of Logic and Algebraic Programming*, 52-53:163–181, 2002.

[20] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In Davide Sangiorgi and Robert Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500. Springer Berlin Heidelberg, 1998.

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

[22] Paul Bogdan, Siddharth Jain, Kartikeya Goyal, and Radu Marculescu. Implantable Pacemakers Control and Optimization via Fractional Calculus Approaches: A Cyber-Physical Systems Perspective. In *Proceedings of the 3rd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '12*, pages 23–32. IEEE, 2012.

[23] Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring Compact Models of Communication Protocol Entities. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I - ISoLA '10*, pages 658–672. Springer-Verlag Berlin, 2010.

[24] Lei Bu, You Li, Linzhang Wang, Xin Chen, and Xuandong Li. BACH 2: Bounded ReachAbility CHecker for Compositional Linear Hybrid Systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1512–1517. IEEE, 2010.

[25] Lei Bu, Dingbao Xie, Xin Chen, Linzhang Wang, and Xuandong Li. Demo Abstract: BACHOL - Modeling and Verification of Cyber-Physical Systems Online. In *Proceedings of the 3rd ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '12*, pages 222–222. Ieee, 2012.

[26] Peter Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Guangyuan Li, Danny Bøgsted Poulsen, and Amelie Stainer. Monitor-Based Statistical Model Checking for Weighted Metric Temporal Logic. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg, 2012.

[27] Peter Bulychev, Alexandre David, Kim G. Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata. In Herbert Wiklicky and Mieke Massink, editors, *10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL 2012)*, volume 85 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16. Open Publishing Association, 2012.

[28] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[29] Taolue Chen, Marco Diciolla, Marta Kwiatkowska, and Alexandru Mereacre. Quantitative Verification of Implantable Cardiac Pacemakers. In *Real-time Systems Symposium (RTSS 2012), 2012 IEEE 23rd*, pages 263–272. IEEE, 2012.

[30] Taolue Chen, Ernst M. Hahn, Tingting Han, Marta Kwiatkowska, Hongyang Qu, and Lijun Zhang. Model Repair for Markov Decision Processes. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 85–92. IEEE, 2013.

[31] Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer Berlin Heidelberg, 2009.

[32] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Contract-Based Slicing. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag Berlin, 2010.

[33] Alexandre David, Dehui Du, Kim G. Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical Model Checking for Stochastic Hybrid Systems. In *Proceedings First International Workshop on Hybrid Systems and Biology, Newcastle Upon Tyne, 3rd September 2012*, volume 92 of *Electronic Proceedings in Theoretical Computer Science*, pages 122–136. Open Publishing Association, 2012.

[34] Jim Davies, Jeremy Gibbons, Radu Calinescu, Charles Crichton, Steve Harris, and Andrew Tsui. Form Follows Function. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, Lecture Notes in Computer Science, pages 21–38. Springer Berlin Heidelberg, 2011.

[35] Conrado Daws and Stavros Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329. Springer Berlin Heidelberg, 1998.

[36] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. On-Line Monitoring for Temporal Logic Robustness. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 231–246. Springer International Publishing, 2014.

[37] Jin Song Dong, Jing Sun, Jun Sun, Kenji Taguchi, and Xian Zhang. Specifying and Verifying Sensor Networks: an Experiment of Formal Methods. In Shaoying Liu, Tom Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, pages 318–337. Springer Berlin Heidelberg, 2008.

[38] Parasara Sridhar Duggirala and Sayan Mitra. Abstraction Refinement for Stability. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pages 22–31. IEEE, 2011.

[39] Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of Discrete Event Systems: Control Theory Approach. *Electronic Notes in Theoretical Computer Science*, 144(4):21–39, 2006.

[40] Raimund L. Feldmann, Forrest Shull, Christian Denger, Martin Höst, and Christin Lindholm. A Survey of Software Engineering Techniques in Medical Device Development. In *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability - HCMDSS-MD PnP '07 (2007)*, pages 46–54. IEEE, 2007.

[41] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny B. Sipma. Collecting Statistics over Runtime Executions. *Formal Methods in System Design*, 27(3):253–274, 2005.

[42] Dharmalingam Ganesan, Mikael Lindvall, and Rance Cleaveland. Architecture-based Static Analysis of Medical Device Software: Initial Results. In *2011 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability - HCMDSS-MD PnP '11 (2011)*, 2011.

[43] Albert Goldfain, Atanu Roy Chowdhury, Min Xu, Jim DelloStritto, and Jonathan Bona. Semantic Alarms in Medical Device Networks. In *2011 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability - HCMDSS-MD PnP '11 (2011)*, 2011.

[44] Martijn Hendriks. Enhancing UPPAAL by Exploiting Symmetry. Technical Report NIII-R0208, Department of Computer Science, Radboud University Nijmegen, 2002.

[45] Thomas A. Henzinger. The Theory of Hybrid Automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE, 1996.

[46] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time Systems Using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 77–117. Springer Berlin Heidelberg, 2008.

[47] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online Monitoring of Metric Temporal Logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer International Publishing, 2014.

[48] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[49] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[50] Jozef Hooman, Robert Huis in 't Veld, and Mathijs Schuts. Experiences with a Compositional Model Checker in the Healthcare Domain. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, volume 7151 of *Lecture Notes in Computer Science*, pages 93–110. Springer Berlin Heidelberg, 2011.

[51] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qinghua Luo, Aravind Sundaresan, and Grigore Rosu. ROSRV: Runtime Verification for Robots. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 247–254. Springer International Publishing, 2014.

[52] Agata Janowska and Wojciech Penczek. Path Compression in Timed Automata. *Fundamenta Informaticae*, 79(3-4):379–399, 2007.

[53] Henrik E. Jensen, Kim G. Larsen, and Arne Skou. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. In *The Spin Verification System: 2nd International SPIN Workshop, SPIN '96 Proceedings*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 33–47. American Mathematical Society Press (AMS), 1997.

[54] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop Verification of Medical Devices with Model Abstraction and Refinement. *International Journal on Software Tools for Technology Transfer*, 16(2):191–213, 2014.

[55] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Model-Based Closed-Loop Testing of Implantable Pacemakers. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on*, pages 131–140. IEEE, 2011.

[56] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg, 2012.

[57] Andrew King, Dave Arney, Insup Lee, Oleg Sokolsky, John Hatcliff, and Sam Procter. Prototyping Closed Loop Physiologic Control with the Medical Device Coordination Framework. In *Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care*, SEHC '10, pages 1–11. ACM, 2010.

[58] Jeonggil Ko, Jong Hyun Lim, Yin Chen, Rvăzvan Musvăloiu-E, Andreas Terzis, Gerald M. Masson, Tia Gao, Walt Destler, Leo Selavo, and Richard P. Dutton. MEDiSN: Medical Emergency Detection in Sensor Networks. *ACM Transactions on Embedded Computing Systems*, 10(1):1–28, 2010.

[59] Farinaz Koushanfar, Miodrag Potkonjak, and Alberto Sangiovanni-Vincentelli. On-line Fault Detection of Sensor Measurements. In *Sensors, 2003. Proceedings of IEEE*, volume 2, pages 974–979. IEEE, 2003.

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

[61] 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 Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer Berlin Heidelberg, 2001.

[62] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.

[63] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems. *Real-Time Systems*, 25(2-3):255–275, 2003.

[64] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 76–87. IEEE, 1995.

[65] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In Horst Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer Berlin Heidelberg, 1995.

[66] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 575–586. Springer Berlin Heidelberg, 1996.

[67] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[68] Axel Legay, Benoit Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer Berlin Heidelberg, 2010.

[69] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[70] Tao Li, Feng Tan, Qixin Wang, Lei Bu, Jian-Nong Cao, and Xue Liu. From Offline toward Real-Time: A Hybrid Systems Model Checking and CPS Co-design Approach for Medical Device Plug-and-Play (MDPnP). In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 13–22. IEEE, 2012.

[71] Tao Li, Qixin Wang, Feng Tan, Lei Bu, Jian-nong Cao, Xue Liu, Yufei Wang, and Rong Zheng. From Offline Long-Run to Online Short-Run: Exploring A New Approach of Hybrid Systems Model Checking for MDPnP. In *Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability - HCMDSS-MD PnP '11 (2011)*, 2011.

[72] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 281–297. Springer Berlin Heidelberg, 1998.

[73] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Start-Up Mechanism. In *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, pages 235–242. IEEE, 1997.

[74] Xintao Ma. *Online Checking of a Hybrid Laser Tracheotomy Model in UPPAAL-SMC*. Master's thesis, Hamburg University of Technology, 2013.

[75] Xintao Ma, Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. Evaluating On-line Model Checking in UPPAAL-SMC using a Laser Tracheotomy Case Study. In Volker Turau, Marta Kwiatkowska, Rahul Mangharam, and Christoph Weyer, editors, *5th Workshop on Medical Cyber-Physical Systems*, volume 36 of *OpenAccess Series in Informatics (OASIcs)*, pages 100–112. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[76] Dominique Méry and Neeraj Kumar Singh. Formalisation of the Heart based on Conduction of Electrical Impulses and Cellular-Automata. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, volume 7151, pages 140–159. Springer Berlin Heidelberg, 2011.

[77] Dominique Méry and Neeraj Kumar Singh. Technical Report on Formalisation of the Heart using Analysis of Conduction Time and Velocity of the Electrocardiography and Cellular-Automata. Technical Report INRIA-00600339, Henri Poincaré University, 2011.

[78] Stefan Mitsch and André Platzer. ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 199–214. Springer International Publishing, 2014.

[79] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[80] Axel Neuser. *A GUI for Real-time Visualization of On-line Model Checking with UPPAAL*. Bachelor's thesis, Hamburg University of Technology, 2014.

[81] Paul Pettersson. *Modelling and Verification of Real-Time Systems using Timed Automata: Theory and Practice*. PhD thesis, Uppsala University, 1999.

[82] Zhengwei Qi, Alei Liang, Haibing Guan, Ming Wu, and Zheng Zhang. A Hybrid Model Checking and Runtime Monitoring Method for C++ Web Services. In *INC, IMS, and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 745–750. IEEE, 2009. International Conference on Networked Computing (INC), International Conference on Advanced Information Management and Service (IMC), International Conference on Digital Content, Multimedia Technology and its Applications (IDC).

[83] Anders P. Ravn, Jiří Srba, and Saleem Vighio. A Formal Analysis of the Web Services Atomic Transaction Protocol with UPPAAL. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 579–593. Springer Berlin Heidelberg, 2010.

[84] Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. State Space Reconstruction for On-Line Model Checking with UPPAAL. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, pages 21–26. IARIA, 2013.

[85] Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. A Graph-Based Transformation Reduction to Reach UPPAAL States Faster. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 547–562. Springer International Publishing, 2014.

[86] Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. State Space Reconstruction in UPPAAL: An Algorithm and its Proof. *International Journal On Advances in Systems and Measurements*, 7(1-2):91–102, 2014.

[87] William H. Sanders, Tod Courtney, Daniel Deavours, David Daly, Salem Derisavi, and Vinh Lam. Multi-Formalism and Multi-Solution-Method Modeling Frameworks: The Möbius Approach. In Gabriele Kotsis, editor, *Symposium on Performance Evaluation - Stories and Perspectives*, pages 241–256. Österreichische Computer Gesellschaft, 2003.

[88] Gerald Sauter, Henning Dierks, Martin Fränzle, and Michael R. Hansen. Lightweight hybrid model checking facilitating online prediction of temporal properties. In *Proceedings of the 21st Nordic Workshop on Programming Theory, NWPT '09*, pages 20–22. Danmarks Tekniske Universitet, 2009.

[89] Achim Schweikard, Greg Glosser, Mohan Bodduluri, Martin J. Murphy, and John R. Adler. Robotic motion compensation for respiratory movement during radiosurgery. *Computer Aided Surgery*, 5(4):263–277, 2000.

[90] Achim Schweikard, Hiroya Shiomi, and John R. Adler. Respiration tracking in radiosurgery. *Medical Physics*, 31(10):2738–2741, 2004.

[91] Koushik Sen, Griogore Roşu, and Gul Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin Heidelberg, 2004.

[92] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95. ACM, 2000.

[93] Robert A. Thacker, Kevin R. Jones, Chris J. Myers, and Hao Zheng. Automatic Abstraction for Verification of Cyber-Physical Systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 12–21. ACM, 2010.

[94] Frits W. Vaandrager and A. L. de Groot. Analysis of a biphase mark protocol with UPPAAL and PVS. *Formal Aspects of Computing*, 18(4):433–458, 2006.

[95] Fuzhi Wang. *Symbolic Implementation of Model-Checking Probabilistic Timed Automata*. PhD thesis, University of Birmingham, 2006.

[96] Yu Wang and Jack M. Winters. A Dynamic Neuro-Fuzzy Model Providing Bio-State Estimation and Prognosis Prediction for Wearable Intelligent Assistants. *Journal of NeuroEngineering and Rehabilitation*, 2(1):15, 2005.

[97] Piotr Wendykier. JTransforms Library, 2015. Last accessed on 29.01.2015. URL: `https://sites.google.com/site/piotrwendykier/software/jtransforms`.

[98] Hao Xu and Tom Maibaum. An Event-B Approach to Timing Issues Applied to the Generic Insulin Infusion Pump. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, volume 7151, pages 160–176. Springer Berlin Heidelberg, 2012.

[99] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 243–258. Chapman & Hall, Ltd., 1995.

[100] Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[101] Heechul Yun, Po-Liang Wu, Maryam Rahmaniheris, Cheolgi Kim, and Lui Sha. A Reduced Complexity Design Pattern For Distributed Hierarchical Command and Control System. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 42–49. ACM, 2010.

[102] Yuhong Zhao and Franz Rammig. Online Model Checking for Dependable Real-Time Systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 154–161. IEEE, 2012.

List of Tables

List of Figures