



Trustworthy Verification of Realtime Systems

Simon Wimmer





Trustworthy Verification of Realtime Systems

Simon Wimmer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Francisco Javier Esparza Estaun

Prüfende der Dissertation:

1. Prof. Tobias Nipkow, Ph.D.
2. Prof. Dr. Jaco van de Pol

Die Dissertation wurde am 07.10.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.11.2020 angenommen.

Abstract

Timed automata are a popular formalism for modeling real-time systems in safety-critical applications. Model checkers for timed automata are an effective verification method that can automatically prove safety of such a model with respect to a temporal specification. This thesis addresses the question of the trustworthiness of verification results obtained by timed automata model checkers. To this end, I have studied two approaches that try to significantly increase the level of trustworthiness by employing another popular verification method, interactive theorem proving. Interactive theorem provers are considered to be highly trustworthy as their correctness relies only on a small and well-tested program core that checks every logical inference made.

In the first approach, verification, a full model checker for timed automata, MUNTA, was constructed within the theorem prover Isabelle/HOL. Techniques from program refinement were used to obtain efficiently executable code. There is a full proof of correctness for MUNTA within Isabelle/HOL, meaning that verification results obtained by MUNTA can be considered as trustworthy as the correctness of Isabelle/HOL itself. MUNTA delivers reasonable performance and provides features comparable to mainstream model checkers for timed automata.

The second approach, certification, attempts to lift the trustworthiness of regular unverified model checkers to the level of trust put in Isabelle/HOL. The unverified model checker is modified to produce a proof for the verification result, a certificate. An independent checker confirms that the certificate is indeed a valid proof of the model checking result for the given model and property. As the checker is itself derived from MUNTA and thus constructed and verified in Isabelle/HOL, the same level of trustworthiness is achieved. The approach has many advantages: a uniform certificate format is applicable to a large range of model checkers that can employ many optimizations during model checking; certificates can be compressed to speed-up certificate checking; and the certificate checker can be easily parallelized.

Zusammenfassung

Timed Automata sind ein beliebter Formalismus, um Echtzeitsysteme für sicherheitskritische Anwendungen zu modellieren. Model-Checker für Timed Automata sind eine effektive Verifikationsmethode, die automatisch die Sicherheit eines solchen Modells im Bezug auf eine gegebene temporale Spezifikation beweisen kann. Diese Dissertation nimmt sich der Frage der Vertrauenswürdigkeit von Verifikationsresultaten, die von Timed-Automata-Model-Checkern produziert werden, an. Dazu habe ich zwei Ansätze untersucht, die versuchen das Level der Vertrauenswürdigkeit zu erhöhen, indem eine weitere Verifikationsmethode, das interaktive Theorembeweisen, eingesetzt wird. Interaktive Theorembeweiser werden als höchst vertrauenswürdig betrachtet, da ihre Korrektheit nur auf einem kleinen und gut getesteten Programmkern beruht, der jede durchgeführte logische Inferenz überprüft.

Im ersten Ansatz, Verifikation, wurde ein vollständiger Model-Checker für Timed Automata, MUNTA, innerhalb des Theorembeweisers Isabelle/HOL konstruiert. Techniken aus dem Bereich der Programmverfeinerung wurden verwendet, um effizienten ausführbaren Code zu erhalten. Es existiert ein vollständiger Korrektheitsbeweis für MUNTA in Isabelle/HOL, was bedeutet, dass die Verifikationsergebnisse, die von MUNTA produziert werden, genauso vertrauenswürdig sind wie Isabelle/HOL selbst. MUNTA erreicht eine angemessene Performanz und bietet Features an, die vergleichbar zu üblichen Model-Checkern für Timed Automata sind.

Der zweite Ansatz, Zertifizierung, versucht die Vertrauenswürdigkeit von Resultaten von regulären unverifizierten Model-Checkern auf das Level von Vertrauen, das in Isabelle/HOL gesetzt wird, zu heben. Der unverifizierte Model-Checker wird modifiziert, um einen Beweis für das Verifikationsergebnis, ein Zertifikat, zu produzieren. Ein unabhängiger Checker bestätigt, dass das Zertifikat tatsächlich ein korrekter Beweis für das Model-Checking-Ergebnis ist. Da der Checker selbst von MUNTA abgeleitet ist und in Isabelle/HOL verifiziert ist, wird dasselbe Level von Vertrauenswürdigkeit erreicht. Der Ansatz hat viele Vorteile: ein einheitliches Zertifikatsformat ist auf eine große Vielfalt von Model-Checkern, die viele Optimierungen während des Model-Checkens verwenden können, anwendbar; Zertifikate können komprimiert werden, um ihre Überprüfung zu beschleunigen; und der Zertifikat-Checker kann leicht parallelisiert werden.

Acknowledgements

I want to start by extending my gratitude to those who keep producing information on the internet that is available to anyone, often without asking for anything in return. Without the work of these people, I would probably never have picked up computer science and the field of this thesis in particular.

Coming to university already with certain interests and a lot of curiosity to finally understand what I had peeked into, the theory professors at TUM, first and foremost Tobias Nipkow, were just ready to quench my thirst with the “Perlen der Informatik” lecture from week two. With Tobias Nipkow continuing this lecture into the second semester, it is no wonder I later ended up as his doctoral student. I want to thank him for his enthusiasm in teaching, for taking me under his wings already as a young student, and later luring me back from overseas by offering me the opportunity that culminated in this thesis. He has been a kind advisor who always had an open ear for me and who let me pursue my ideas freely, nudging me in the right direction when it was needed (though, sometimes I could probably have needed a stronger push as well). Special credit also goes to Lars Noschinski, who was the teaching assistant in those early “Perlen der Informatik” lectures and who cheerfully introduced me to the wonders of Isabelle from my second semester on.

I want to thank my colleagues at the Chair for Logic and Verification, Bohua, Fabian, Helma, Johannes, Jonas, Julian, Ondřej, Kevin, Lars, Lukas, Manuel, Max, Mohammad, and Peter, for providing a fun and friendly work environment and their never-ending motivation for bringing fresh habits to the group, pursuing new ideas and projects, discussing matters of the earth and the universe, teaching me new things, and sometimes motivating me to do sports or joining me in these activities. Some of these encounters were certainly life-changing. On this note, I specifically want to mention Ondřej and Max who became my dear “Heide” friends. I am very grateful to have met them. Helma needs to be commended for her ongoing friendliness and helpfulness, even when I had once again postponed my bureaucratic duties to the very last day (or later).

My co-authors, Frédéric, Jaco, Johannes, Joshua, and Peter, deserve to be mentioned for their enthusiasm for my ideas and their help in implementing them. Without the work of Peter in particular, this thesis would not have been possible. Also, having worked already as a pair-proving team in research, we proudly represented our group in the fun and engaging “VerifyThis” competition.

Max, Manuel, Frédéric, and Peter have provided helpful comments on earlier versions of this thesis. Thank you!

I thank my friends for being there and coping with me during all these year, particularly those who lived closest to me and therefore suffered the most from this thesis: Benedikt, Fabian, Julia, and Liesa. My final thanks goes to my family for their loving support, which I could always be certain of, even when they could not see me as much as they would have liked to. My mother drew the illustration on the cover.

Contents

Abstract	v
Zusammenfassung	vii
Acknowledgements	ix
Contents	xi
1 Introduction	1
1.1 Research Objective	2
1.2 Summary of Contributions	5
1.3 Outline	9
2 Timed Automata: An Overview	11
2.1 Introduction	11
2.2 Semantics	12
2.3 The Model Checking Task	14
2.4 Regions	15
2.5 Zones and Abstractions	16
2.6 Difference Bound Matrices	18
2.7 Other Model Checking Approaches	19
2.8 Summary	20
3 Formalizing (Probabilistic) Timed Automata	21
3.1 Regions	21
3.2 Zones and Difference Bound Matrices	22
3.3 Extrapolation	23
3.4 Probabilistic Timed Automata	23
4 Constructing a Verified Model Checker for Timed Automata	27
4.1 Refinement	27
4.2 Algorithm Verification	29
4.3 Beyond Reachability	30
4.4 Modeling Formalisms	31
4.5 A Practical Tool	32
5 Certifying Model Checking of Timed Automata	35
5.1 Certificates for Unreachability	35
5.2 Certificates for Büchi Emptiness	37
5.3 Implicit Abstraction	40
5.4 Practical Verified Certificate Checking	41

CONTENTS

6 Conclusion	43
6.1 Related Work	43
6.1.1 Formalization of Timed Automata and Markov Models	43
6.1.2 Verified Model Checking	43
6.1.3 Algorithm Verification and Refinement	44
6.1.4 Certification in Verification	45
6.2 Limitations	46
6.2.1 Simplifying Assumptions	46
6.2.2 Modeling Formalism	47
6.2.3 Temporal Properties	48
6.2.4 Efficiency	48
6.2.5 Trustworthiness	48
6.3 Future Work and Perspectives	49
6.4 General Reflections	51
Bibliography	53
A Formalized Timed Automata	71
B MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper)	89
C Verified Model Checking of Timed Automata	97
D Munta: A Verified Model Checker for Timed Automata	117
E Verified Certification of Reachability Checking for Timed Automata	127
F Certifying Emptiness of Timed Büchi Automata	147

1 Introduction

The advent of computer systems was probably the main technological advance that shaped the world's history over the course of the last 50 years. Nowadays, computers are ubiquitous: most of us carry at least one of them around everywhere we go and there is hardly any sector of life that they have not managed to penetrate yet. Increasingly, we have also trusted them with our lives: they steer our aircraft, control cardiac pacemakers, insulin pumps, ventilators, and other medical devices, and will soon drive our cars. The examples that I just named are all special in their common characteristic that they combine a software system with a component that interacts with the real physical world. Hence they are called *cyber-physical systems (CPS)* [131]. This characteristic sets them apart from regular software systems, like a text processor, your web browser, or an internet platform. While the latter are clearly important, an error in their software does usually not immediately lead to a catastrophe such as the loss of life or the explosion of a rocket.

Unfortunately, CPS have been plagued by fatal software failures from the early times of their existence until today [71, 87]. This is not really surprising as some of these software systems today are the most complex systems that have ever been constructed by human beings [1, 39, 48, 53]. Thus, it maybe startling to think that contrary to common engineering practice in many older disciplines, where construction starts from a detailed blueprint and calculations, the reliability of software is still largely established by simple trial-and-error: some test inputs are invented in one or the other way (ranging from very simplistic ad-hoc methods to sophisticated methods such as model-based testing [105]); they are given to the software and it is checked whether the expected outcome matches the actual outcome (where the outcome could be a concrete output produced by software, a change in its state, an action that is triggered, etc.). If the outcome is deemed incorrect for any of the test cases, then the software is said to have a *bug* that needs to be corrected.

Once the outcome for all test cases is correct, the whole system is assumed to function correctly. It was already noted by one of the pioneers of computer science, Edsger Dijkstra, in his influential Turing Award Lecture “The Humble Programmer” [61] that

program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [.]

This is even more true for CPS as they continuously interact with the physical world (they are part of a “closed loop” [203]) and thus their expected behavior cannot even be described in terms of simple input/output pairs. In “The Humble Programmer”, Dijkstra also suggested a pathway out of this situation [61]:

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

Ever since, this has been the main theme of a subdiscipline of computer science named *verification*. It sets out to give a formal analysis of computer systems that *proves* that the system works correctly under all given circumstances *with mathematical rigor*. Researchers in the field soon came to the conviction that real software is far too complex to be analyzed in this way by hand (e. g. with a mathematical proof of correctness written on paper). This kind

of analysis can nowadays only be found in research papers or university textbooks. Instead, today, verification itself now uses computers to get the job done. That is, verification is usually always understood to be *computer-aided verification*.

With this, the following question immediately arises:

If we use a piece of software (*the verifier*) to guarantee that another computer system works correctly, how can we trust the result of the whole verification process if the verifier itself could have a bug?

This, of course, is an important question to answer, especially if verification is meant to be applied to CPS on which we want to trust our lives in the end. In this thesis, I attempt to answer this question for a certain class of CPS, so-called real-time systems (RTS) [113], by using a combination of different techniques from computer-aided verification.

1.1 Research Objective

In the following, I want to state the research objective that I just outlined in more explicit terms. I will start by summarizing the two particular flavors of computer-aided verification that this thesis is concerned with: interactive theorem proving and model checking.

Out of the two, *interactive theorem proving* [190, 167] is the older and more versatile method. It is built around the idea that formal logics are perfectly suited for being mechanized as computer programs. One such program is a *proof checker*: given a logical formula and a proposed proof for it (in some formal proof calculus), the proof checker either confirms that the given proof is indeed a valid proof of the formula, or it rejects the proof. This approach was pioneered by De Bruijn’s Automath [145].

The logical formula and the proposed proof are usually written by a human. Thus this idea naturally leads to the concept of a *proof assistant*: in addition to providing the capabilities of a proof checker, the program also assists the user while developing the proof. Typically, this always includes feedback on the validity of individual proof steps while the user is spelling out the proof. However, the assistance can also take the form of more sophisticated features such as search tools for finding useful proven theorems, suggesting full proofs automatically, or identifying unprovable statements. In any case, the proof is developed through interaction between the user and the computer. Hence this method is also called *interactive theorem proving*. The terms “interactive theorem prover” and “proof assistant” are used interchangeably in the research community. I will also use both terms in this thesis.

In contrast to interactive theorem provers, a third form of these “logical programs”, *automated theorem provers* [69], only take a formula as their input and try to automatically find a proof for it. With Newell’s “Logic Theorist” [139, 82], this approach was studied even before the proof checker idea came along, as early as in 1956. Automated theorem proving is still an active field of research today but will not play a further role in this thesis. The reason is that these systems are typically not capable of proving anything as complex as I set out to do. While they have been used to resolve open mathematical conjectures [98] or to (semi-automatically) verify programs and algorithms [79], they lack in expressivity. Therefore, to me, there does not seem to be a simple way to apply them to anything that mixes a larger body of mathematical theory and verification of software. Nevertheless, automated theorem provers still played a crucial part in my work: they can be connected to proof assistants to find proofs for intermediate propositions automatically [31, 30, 160] and thus made my work significantly easier.

The main strength of interactive theorem proving is its expressivity. The logics that are typically used are powerful enough to capture computer programs on the one hand but also

to state any mathematical proposition on the other hand. This is also reflected in the broad range of domains that proof assistants have already been applied to: in the verification of an operating system [112] and a C compiler [24], for instance, but also for proving mathematical theorems like Hales’ proof of the Kepler conjecture [86], the Four Colour theorem [75] or the Odd Order theorem [76]. The big weakness of interactive theorem proving (as a verification method) is also apparent: it requires human interaction. This makes it a quite laborious task and prevents it from scaling well to the verification of large systems.

Conversely, *model checking* [11, 53] is a push-button method, i. e. it works fully automatically, but it lacks in expressive power compared to interactive theorem proving. There are many different variants of model checking but they always follow the same basic recipe: the subject of study is a *model* of a system, and one wants to *check* certain properties of the model; the model checker’s task is to answer these queries with “yes” or “no”. The system might be a piece of software or hardware but could also be a board game or a biological process. The *model* is a description of the system in the language of some mathematical formalism. In the beginning, the formalism of choice was finite-state machines but now a large variety with many different features is in use. This variety is also reflected in the granularity of the model. For a computer network communication protocol, for instance, one could choose to model an abstract description of the protocol that captures how different entities exchange messages and when. Or, one could choose to model the C code of the bus drivers implementing the protocol.

In verification, we want to ask questions about this model. Sticking with the communication protocol example, we could e. g. ask:

Is it true that, whenever the sender emits a message that it will reach the recipient eventually?

Or we could also ask:

Will the sender always get a confirmation of delivery within 10 seconds after the message has reached the recipient?

We can see that these questions have a *temporal* component. Therefore, in model checking, these questions are asked in the form of a tailor-made language, as formulas in a so called *temporal logic* [164]. In the words of logic, the model checker’s task can now be reformulated as follows:

Given a *model* and a temporal logic formula, *check* whether the model is indeed a valid model of the formula.

Model checkers have also been applied successfully to a number of complex verification tasks. It may not come as a surprise that there are also modeling formalisms that have been invented specifically to verify CPS, and RTS in particular. One popular such formalism is the concept of *timed automata (TA)*. Since its invention by Alur and Dill in the 90s [2], a rich research area has evolved around it. Different temporal logics have been studied in connection with the formalism [53] and a number of model checkers have been constructed. They have been successfully applied to the verification of numerous real-world examples, ranging from pacemakers [108], over multimedia [90], communication [172], and clock synchronization [169] protocols, to vehicle control software [182].

TA lie at a certain sweet spot between expressivity and decidability within the range of modeling formalisms for CPS. For TA, most important questions can still be answered algorithmically [53], while for any slightly more complex CPS formalism, one can only provide semi-algorithms [181]: if these algorithms give an answer, then it is correct; however, for some

1 Introduction

models they might never terminate, i. e. one may never get an answer. It is for this positive characteristic that I chose TA as the subject of study for my research.

Let us now return to the question that I raised at the end of the initial section. We have looked at two types of verification tools, interactive theorem provers and model checkers, and I have given some examples where these tools have been applied to critical systems in the real world. But how do we know that we have gained any assurance in the safety of the systems by applying these tools? How can we be sure that they themselves do not have a bug, jeopardizing the whole effort? For interactive theorem proving, the answer to this question essentially lies within the realm of logic itself. Just like a logical proof can be built from only a very small set of axioms, the proof assistant can implement these axioms in a small *inference kernel*. Then, every proof has to pass through this kernel to be accepted. This means that we only have to trust that the inference kernel implements the axioms correctly—any other code around it needs not be trusted. This design principle was first implemented in the system *LCF* [78] and is therefore often called the LCF approach. Many interactive theorem provers follow this tradition [178, 151, 77, 55, 88].

For model checkers, the story is different. They usually rely on a well-understood theory with model checking algorithms that are considered correct with respect to this theory. The arguments for the correctness of these algorithms are usually spread out as pen-and-paper proofs in different scientific papers. This is a first potential source of errors. Often proofs are only given in a highly abbreviated form, are omitted, deferred to the appendix, or simply incorrect. In any case, it is often trusted to only a small number of experts in the community to understand the arguments for the correctness of model checking algorithms in detail.

Indeed, TA are one of the most prominent examples where this kind of error occurred. In 2003, Patricia Bouyer noticed that the common model checking algorithms that were used at the time (and are still in use today) were incorrect for the general class of timed automata [35]. This was only years after the first TA model checkers had been constructed and had already been used to verify real-world models. Fortunately, in practice the error could only lead to spurious incorrect behaviors of models being detected, and thus did not lead to any incorrect models being verified. Another notable example is an incorrectness in the widely-used partial order reduction technique which was only discovered last year [171]. This error could indeed have potentially fatal consequences by allowing unsafe models to be deemed correct by the model checker.

More common sources of errors, however, are in the implementation of a model checker. First, these tools are complex pieces of software, and as with any complex software (that is programmed by a human) there is ample room for human error. Second, while the model checking algorithms that are established in research can usually be considered correct, they typically have to be augmented with many small optimizations in the implementation to make the resulting model checker efficient. These optimizations are often very subtle and are frequently done on the spot, without even attempting to give a new correctness proof on paper.

For these reasons, proof assistants are often considered more trustworthy than model checkers. “But is that really so?”, you might contest, “After all the axioms of your logic and their implementation in the inference kernel could also be incorrect”. While that is certainly true, I would still argue that interactive theorem provers reach essentially the highest standard of trustworthiness that could be imagined. First, for one of them, HOL Light [89, 117], another layer of assurance has been added: the inference kernel itself was modeled in interactive theorem provers (HOL Light and HOL 4) and it was proven that it only allows sound logical inferences. Repeating this effort can certainly be imagined for any other proof assistant. But even for interactive theorem provers that lack such a formal proof of correctness,

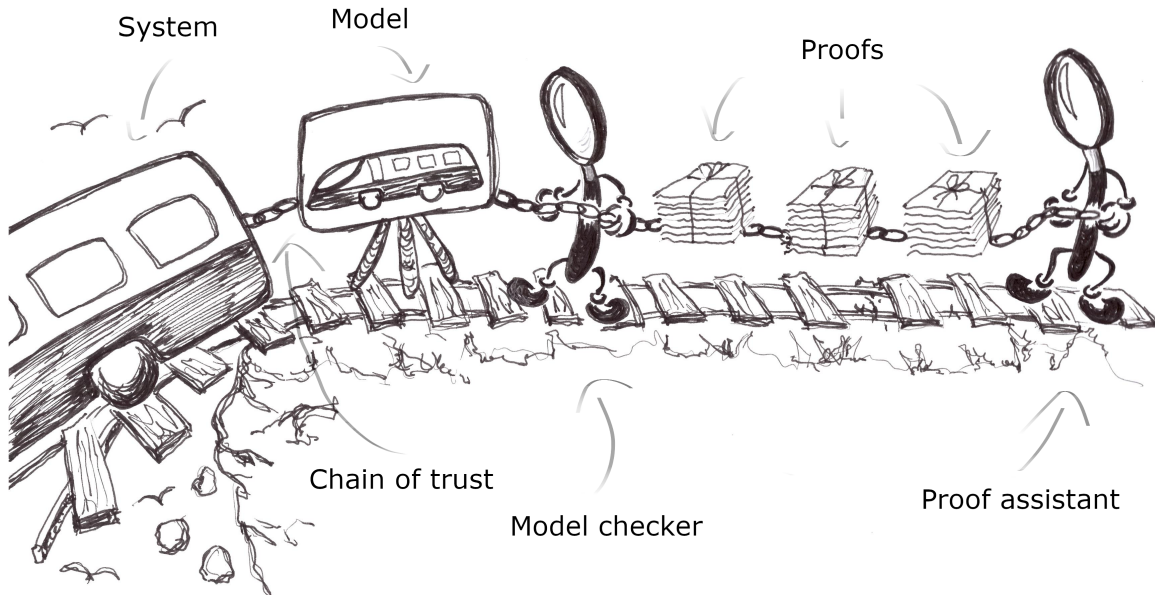


Figure 1.1: Illustration of trust reduction.

the trustworthiness of their kernel is manifested as a community effort. On the one hand, it is assumed that enough people have read and checked the code of the inference kernel. On the other hand, established tools can usually look back at a history in which many have used the tool but in which logical inconsistencies have only been discovered extremely rarely. I consider such an inconsistency much easier to spot than if a model checker gives an incorrect answer for a non-trivial model.

Finally, the most convincing argument is probably given by simply considering the size and complexity of the code that needs to be trusted in order to trust the verification result. To “trust” means here to assume that the code is programmed correctly. This is often referred to as the *trusted code base (TCB)* of a verification tool. For an interactive theorem prover, this may range from a few hundred to a few thousand lines of ML or OCaml code, while for model checkers the TCB may reach tens of thousands of lines of highly optimized C or C++ code. A similar argument can be given on the level of the formalisms. The axioms of a formal logic are typically much easier to be stated and to be checked by a human than the description of a modeling formalism and a temporal logic with corresponding model checking algorithms.

From this discussion, one can now derive the following idea of a “trust reduction”. It is illustrated in Figure 1.1. Our goal is to obtain RTS (the train in the picture) that are as trustworthy as possible (e.g. to keep the train from taking a wrong turn and falling off a cliff). For this, we can model them (or more precisely, e.g. their control unit) as timed automata, and phrase a description of the RTS’s correct behavior as a temporal logic formula. We then use a model checker to ensure that the model really exhibits the intended behavior. We have thus reduced our trust in the RTS to the question of whether we trust the model checker and the models and formulas we came up with. To increase our level of trust further, we can now attempt to reduce our trust in the model checker to trusting an interactive theorem prover instead. The theme of this thesis is to add this last step to the trust reduction chain. As I argued in the last paragraph, this is one of the highest levels of trustworthiness that we can reasonably expect to achieve.

1.2 Summary of Contributions

Now that I have laid out the goal of my research, I want to give a brief overview of what I have done to get there. As I already stated above, my model of choice are TA. My proof

assistant of choice is Isabelle/HOL [152]. Isabelle is a generic theorem prover that can in principle host a range of different object logics. Out of these, HOL is the most widely used flavor, which has also been implemented in many other systems [77, 88]. The logic provides a good tradeoff between expressivity and convenience of use, offering advantages such as fully automated type inference. Its implementation in Isabelle, Isabelle/HOL has been successfully applied to large efforts in verification [112, 86, 175, 65, 138].

There are multiple reasons for my choice of Isabelle/HOL. The most obvious are of social nature: I have been introduced to this tool already during the second semester of my undergraduate studies and later I joined Tobias Nipkow’s research group in Munich that has been at the core of Isabelle development for years. However, Isabelle/HOL was also the most suitable choice beyond these social factors. I see three main points here.

First, Isabelle/HOL arguably offers the highest level of proof automation among all interactive theorem provers. This is mainly due to its powerful term rewriting engine, the *simplifier* [148], and its ability to hand off proof obligations to automated theorem provers and SMT solvers via the *Sledgehammer* tool [31, 30, 160]. However, there are also many other facets, such as offering the most convenient user interface, and a general culture of developing the tool with automation in mind.

Second, Isabelle/HOL offers a mature facility for *code generation* [84, 83]. This component allows one to export code in the functional programming languages Standard ML, OCaml, Haskell, and Scala from definitions in HOL. This feature is essential if one does not only aim to prove some mathematical theory correct but to also obtain practically usable verified software in the end.

Third, the fact that Peter Lammich has also conducted his work on refinement [123, 126, 127] in Isabelle/HOL. Building on this work, we were able to produce efficient highly trustworthy tools for TA model checking. If it were not for this previous work, I would probably have obtained highly trustworthy but terribly inefficient verification tools. A particularly fortunate coincidence was that his work on refinement to imperative code [126] had just emerged when I was looking into extracting verified tools for the first time.

As I outlined at the end of the last section, the overarching goal of the work that is presented in this thesis is to take one further step in the chain of trust reduction. This is, to replace the need to trust timed automata model checkers with trusting Isabelle/HOL instead. Another way to view this is as an effort in greatly reducing the TCB of timed automata model checking. I will now sketch the steps of my journey to achieve this goal.

Theory comes before its implementation. Therefore, before one can consider concrete model checking tools, the first natural step is to ensure that the theory behind them is sound. This was the first part of my work. In the basic recipe, one first defines TA in Isabelle/HOL and assigns a mathematical meaning to them (i. e. one defines their *semantics*). Then, one defines the model checking question. Recall that, usually, this question would be:

Is the given timed automaton a valid model for the given temporal formula?

However, in this first step I simplified the question by restricting the formula to so-called reachability properties. Speaking in terms of verification, reachability properties are highly useful for asking whether the system can ever encounter a fatal error, i. e. whether a fatal system state is *reachable*. In fact, in practice, model checking of timed automata is often restricted to or concerned with reachability properties. Nevertheless, I also provided an avenue to extend my work to more complex temporal properties later on.

So, at first, I proved in Isabelle/HOL that the essential constructions that are used in the most widely used approach for TA model checking are correct [191]. On the one hand, this is the region construction that was already put forward in the seminal paper by Alur and Dill [2].

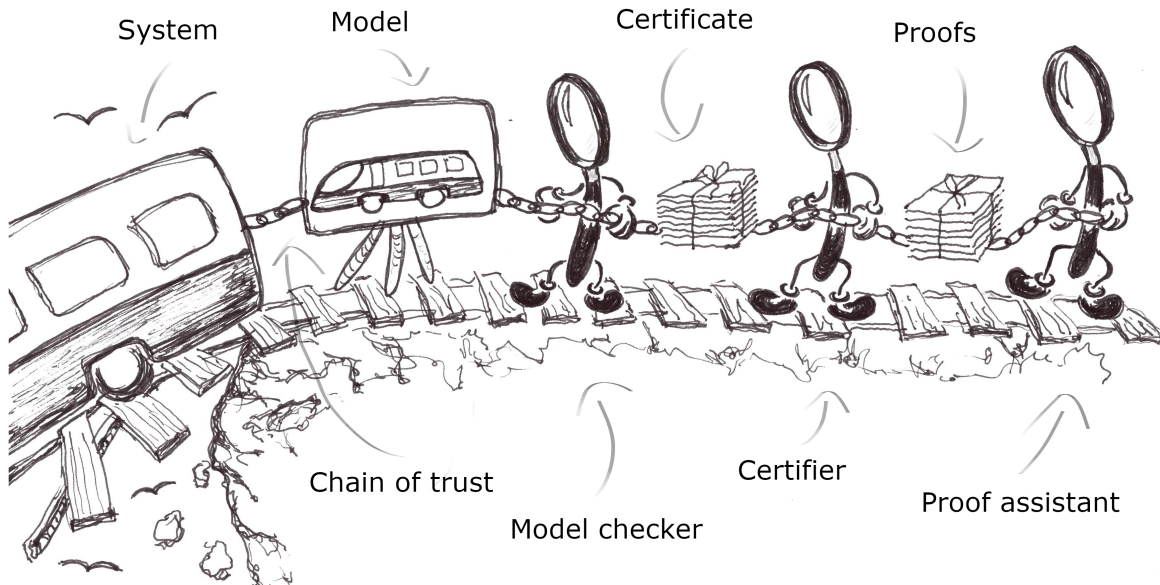


Figure 1.2: Illustration of trust reduction with certification.

On the other hand, there is a more scalable approach that relies on so-called zones to capture the time component of TA states symbolically. It was exactly this approach that was found to be wrong by Patricia Bouyer in 2003 [35]. My work mainly relies on her proofs that repair the classic results on zone-based model checking for the case of diagonal-free timed automata [34]. What makes the formalization of this work in Isabelle/HOL particularly interesting is that her proofs are rather intricate and themselves rely on the correctness of the region construction provided by Alur and Dill. We can now say with high certainty (in so far as we trust Isabelle/HOL) that Patricia Bouyer’s work indeed salvages the theory of zone-based TA model checking and that no further fundamental errors have snuck in.

From the theory I could then move to tools. That is, my next goal was to consider actual model checkers for timed automata. I have considered two main approaches, *verification* and *certification*. In the verification approach (as illustrated in Figure 1.1), the goal is to construct the model checker fully within Isabelle/HOL accompanied by a proof of its correctness. We followed this approach in joint work with Peter Lammich [197]. Starting from the theory that I formalized in the first step, we developed verified abstract implementations of the main algorithms that are similar to the ones that are at work in the popular tool UPPAAL [16]. In the next step, we relied on Peter’s previous work to refine these algorithms in multiple steps to turn them into concrete and efficient programs. Finally, we used Isabelle/HOL’s code generator to obtain an actual executable version of the model checker that is comparable in its core functionalities to UPPAAL. One of the highlights of this work is that we were able to implement the main data structure that is used for the aforementioned zones, *Difference Bound Matrices (DBMs)* [19], imperatively. This way, we can achieve comparable efficiency to unverified model checkers. Later, I improved on this work by equipping the model checker with further capabilities and a graphical user interface. The resulting tool was named MUNTA [192, 196].

In the certification approach¹, one does not simply attempt to supersede existing unverified model checkers but rather tries to validate their work. The idea (as illustrated in Figure 1.2) is that the model checker runs normally but in addition to just producing a yes/no answer, it also emits a proof that this answer is indeed correct, the *certificate*. Another tool, the *certifier* can then validate that the certificate is in fact a valid proof of the answer. If the certifier is in

¹In the literature, “to certify” sometimes refers to the act of verification with the help of a proof assistant. I will only strictly use the terms “to certify” and “certification” for what I describe in the following.

1 Introduction

turn a tool that was proven to be correct in Isabelle/HOL, we can achieve the same level of trustworthiness as for the verification approach, without the need to construct a full model checker in Isabelle/HOL. We sacrifice *completeness*, however: if the model checker’s certificate is rejected by the certifier, we know nothing. Conversely, if the certificate is accepted by the certifier, we know that we can put high trust in the model checker’s answer.

Note that one can always build the trivially correct certifier that simply rejects every input. Thus one also has to argue for its non-triviality at least informally. One such completeness property would be that for every valid pair of a model and a formula, there is a certificate that will be accepted by the certifier. In addition, one could attempt to show that the certifier accepts any valid certificate. Proving any of these results might potentially require significantly more background theory, however. Moreover, even if we have proved both of these completeness properties, in the case that a certificate is rejected, we still cannot infer the reason for that. The cause could be a bug in the model checker or the certificate extraction, or the model might simply not satisfy the formula.

Certification has two main advantages. First, it can be much easier to verify the certifier in a proof assistant. Second, checking that the certificate is correct may be computationally much simpler than full model checking. For TA, we know that the asymptotic worst-case time complexity of certification is the same as of model checking because the problem is PSPACE-complete [2]. Yet, still, it turns out that the certificates can be checked much more efficiently for concrete models than performing full model checking in MUNTA.

Building on the existing work for MUNTA, I have first studied this approach in joint work with Joshua von Mutius for reachability properties [198]. With Frédéric Herbreteau and Jaco van de Pol, we have later extended this work towards so-called *liveness* properties [193], providing a crucial corner stone to handle full *linear temporal logic (LTL)* [164].

To conclude this section, I want to briefly touch on a few important points that are outside the scope of this thesis. Figuratively speaking, the work presented in this thesis extends the chain of trust for the verification process of TA at its tail. However, some potential pitfalls remain at both ends. On one end, we model the system under study (and its environment) using TA and formulate the criteria for its correctness using temporal logic formulas. However, there is no assurance that we do not get the model or the formulas wrong. For the formulas, this problem seems to be fundamental. It is similar to the problems an engineer would run into if they want to construct an aircraft but base their calculations on the viscosity of water rather than air. These problems can probably only be addressed socially, i. e. by humans checking each other’s work. For the model, there is a range of answers that can be given. First of all, the complete model usually falls into two parts: one for the components of the system and one for its environment. For the environment, the typical solution is over-approximation: the model of the environment should allow all behaviors of the environment in the real world and many more. Still, one also has to get this over-approximation right. If we consider autonomous driving, for instance, we may model other cars and humans on the street but we should also not forget about the notorious gigantic glass pane that might be carried across the street.

On the positive side, if we are certain that we got the formulas and the environment component right, we do not need to be concerned about the system anymore: if the model is a valid model for the formula, the system component has to be correct. There is one more aspect to worry about on the model side, however. We may now have a correct model of the system but how do we get a correct system from it? One solution that has been proposed in a broad body of work is to automatically synthesize code that correctly implements the model by construction [7, 23, 200, 85, 157, 38, 115]. Similarly to what is done in this thesis, the trustworthiness of this method could also be increased by verifying or certifying it with

the help of a proof assistant. But even if we now have code that we trust, how do we know that it will be correctly executed? How can we trust the compiler that turns the code into machine code? How can we trust the operating system that runs this code? And how can we trust the underlying hardware? Luckily, all the questions have already been considered by the verification community and are still the subject of active research [24, 177, 112, 52].

At the end of the chain, there is Isabelle/HOL. I briefly want to detail what exactly it means to trust this end of the chain. Essentially, there are two distinct parts: Isabelle/HOL itself and what I write down in Isabelle/HOL. As I argued above, there is a strong case for trusting a proof assistant like Isabelle/HOL, even though its logical inference kernel has not been verified. I will also discuss the correctness of Isabelle/HOL specifically in further detail in the conclusion (cf. Section 6.2). For what I write down in Isabelle/HOL, the strength of proof assistants being grounded in logic shows up again. Essentially, the only piece that needs to be trusted here is that I correctly define what the mathematical meaning of the TA formalism is, i. e. the *TA semantics*. This part is indeed very concise. It is only about 150 lines of Isabelle/HOL code and thus comparable in size and complexity to a pen-and-paper formulation of the semantics. The crux is now that all other theorems, and specifically the correctness theorem for MUNTA and the certifier are formulated with regard to this semantics. If you are ready to trust Isabelle/HOL and my semantics, then you can also trust these correctness theorems.

1.3 Outline

To conclude the introduction of this thesis, I want to give a brief outline of its remaining contents. The thesis is based on six publications. In the next chapter, I will introduce the formalism of TA and summarize the main concepts of TA model checking, which will be crucial to understand the rest of the thesis. The subsequent three chapters will each set the stage for two of the papers by summarizing the main ideas, putting them into perspective with respect to the overall theme, pointing to related work, and considering aspects that I would do differently in retrospective.

The first of these chapters (Chapter 3) focuses on formalizing TA and abstract results on model checking in Isabelle/HOL. This also includes joint work with Johannes Hölzl [195], in which we formalized fundamental results about probabilistic timed automata (PTA). This formalism extends TA to model uncertainty, which is often desirable in practice.

The following chapters treat the approaches of verification (Chapter 4) and certification (Chapter 5). Finally, Chapter 6 concludes by providing some general reflections on the research I conducted, examining related work, pointing to remaining limitations, and laying out some future directions of research that build upon this thesis. The appendix consists of the six papers that constitute the thesis. These papers are:

Paper A Simon Wimmer. “Formalized Timed Automata.” In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Proceedings*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 425–440. DOI: 10.1007/978-3-319-43144-4_26

Paper B Simon Wimmer and Johannes Hölzl. “MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper).” In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 597–603. DOI: 10.1007/978-3-319-94821-8_35

1 Introduction

- Paper C** Simon Wimmer and Peter Lammich. “Verified Model Checking of Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Proceedings, Part I*. ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 61–78. DOI: 10.1007/978-3-319-89960-2_4
- Paper D** Simon Wimmer. “Munta: A Verified Model Checker for Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Proceedings*. Ed. by Étienne André and Mariëlle Stoelinga. Vol. 11750. Lecture Notes in Computer Science. Springer, 2019, pp. 236–243. DOI: 10.1007/978-3-030-29662-9_14
- Paper E** Simon Wimmer and Joshua von Mutius. “Verified Certification of Reachability Checking for Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Proceedings, Part I*. ed. by Armin Biere and David Parker. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 425–443. DOI: 10.1007/978-3-030-45190-5_24
- Paper F** Simon Wimmer, Frédéric Herbreteau, and Jaco van de Pol. “Certifying Emptiness of Timed Büchi Automata.” In: *Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020*. Vol. 12288 LNCS. Lecture Notes in Computer Science. Springer, 2020, pp. 58–75. DOI: 10.1007/978-3-030-57628-8_4

In the remainder, I will only refer to the papers by their identifiers, Paper A and so on.

2 Timed Automata: An Overview

In principle, the papers that constitute this thesis are either written to be self-contained or contain the relevant references to understand the material. However, these publications all needed to make concessions to space requirements, thus discussions of preliminary knowledge are sometimes cut quite short. For this reason and to set a common basis for the whole thesis, this section will explain the most important notions of timed automata and model checking that are relevant for understanding this thesis, without going into excruciating detail. Anyone with a background in computer science or mathematics should be able to understand this material. I will not give an introduction to Isabelle/HOL or other topics pertaining to interactive theorem proving, however. The remaining chapters can largely be understood without this knowledge, and the papers point to introductory material where relevant. To readers interested in an introduction to Isabelle/HOL, I recommend reading the first half of Nipkow and Klein’s “Concrete Semantics” [150].

2.1 Introduction

Model checking needs to start with a model, of course. As I already state above, many different formalisms for the specification of these models have been studied before. Usually, the basic starting point is the notion of a Kripke structure, transition system or finite-state machine, which are augmented with additional features in more sophisticated modeling formalisms. These three formalisms only differ slightly in their precise definition and can be considered to be the same for the purposes of this thesis. Therefore, I will not explicate these differences here and will only use the term *transition system* from now on. The nodes of transition systems will be referred to as *states* and the edges as *transitions*.

Transition systems are essentially annotated graphs where the nodes of the graphs represent states of some system and edges represent transitions between these states. The nodes and edges can be labeled. One purpose of this is to model *communication* between systems. To this end, a separate transition system is used to model each system in a network of communicating systems. Transitions can then be annotated to model message exchanges over pre-defined channels.

In timed automata [3], transition systems are extended with a notion of *clocks* to model physical time. These clocks act as stopwatches. A transition can choose to reset certain clocks when it is taken. While idling in a state, the time reading on all clocks is elapsing at the same speed (i. e. in physical terms all clocks are perfect). These time readings can be made use of on transitions: a transition can have a *guard* which demands that the transition can only be taken when certain conditions on the current values of the clocks are fulfilled.

Figure 2.1 gives an example of two communicating timed automata¹. They only use one channel for communication, which models the physical action of the user pressing the switch. This is often done by designating two special types of *action* labels for ingoing and outgoing actions. In this case, the channel is named *press*, and the input and output actions are *press?* and *press!*, respectively. The automata can *synchronize* on transitions with matching input and output labels, meaning they will take the transition at exactly the same time.

¹Similar examples can be found in the literature, for instance in the tutorial paper by Bengtsson and Yi [19].

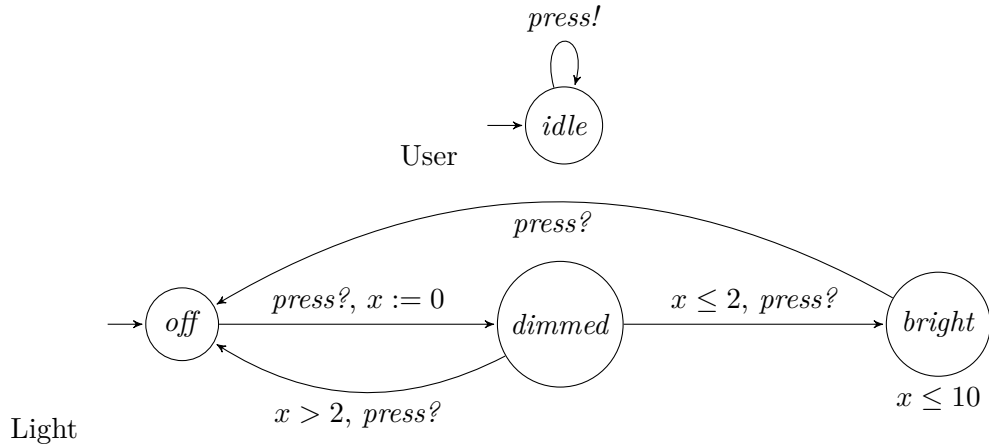


Figure 2.1: An example of two communicating timed automata.

The automata model a simple light switch and its user. The light can either be off, dimmed, or bright. The initial states, *off* for the light and *idle* for the user, are marked by short arrows going in from the left. If the user presses the switch once when the light is off, then the light should be turned on and dimmed. If the user quickly (that is, within two seconds) presses the switch a second time, then the light should turn bright. This behavior is achieved by adding the clock x . It is reset to 0 whenever the transition from *off* to *dimmed*, which turns the light on, is taken. Then, x is used to measure whether two seconds have already passed. While the value of x is at most two, the transition to the bright state is enabled by the guard $x \leq 2$. If a second press is not received within this time, then the transition cannot be taken anymore and instead the transition to the *off* state is enabled by the guard $x > 2$.

The model also features an *invariant* on the *bright* state. Its meaning is that the system can only stay in this state while the condition $x \leq 10$ is true. This could be used to model that the light can only be bright for at most ten seconds before it has to perform an emergency shutoff due to overheating.

Observe that the model of the user is completely *nondeterministic* as the user can press the switch at any time. This is an example of a useful over-approximation of the environment in modelling. A realistic user will never be able to press the button a second time without a delay of say 10ms, for instance. However, any behavior that can be exhibited by a realistic user (as far as the light switch is concerned) is captured by our model.

2.2 Semantics

I will now briefly give the formal semantics (i. e. the precise mathematical meaning) of the timed automata formalism as a basis for further explanations. As we have seen, clock constraints are used to guard states (as invariants) and transitions. Formally, a clock constraint is a conjunction of constraints of the form $x \sim c$ or $x - y \sim c$, where x and y are clocks, c is an integer constant, and \sim is one of $<$, \leq , $=$, \geq , $>$. *Clock valuations* are used to give a semantics to clock constraints. A clock valuation is a mapping from clocks to their current time values. That is, if the clocks are drawn from the set X , then any function of type $X \rightarrow \mathbb{R}_0^+$ is a clock valuation. A clock valuation v satisfies a clock constraint $x \sim c$ iff $v(x) \sim c$ and a clock constraint $x - y \sim c$ iff $v(x) - v(y) \sim c$. I write $v \models x \sim c$ or $v \models x - y \sim c$ to signify this. We need two operations on clock valuations.

Time shift For a clock valuation v and a real d , $v \oplus d$ is defined such that $(v \oplus d)(x) = v(x) + d$ for any clock $x \in X$. That is, the values of all clocks are shifted by time d .

Clock reset For a clock valuation v and a set of clocks $r \subseteq X$, $[r \rightarrow 0]v$ is defined such that:

$$([r \rightarrow 0]v)(x) = \begin{cases} 0 & \text{if } x \in r \\ v(x) & \text{otherwise} \end{cases} .$$

That is, the values of all clocks in r are reset to 0.

Formally, a timed automaton A is defined as a pair $(\mathcal{T}, \mathcal{I})$ where \mathcal{T} is the set of transitions, and \mathcal{I} gives the invariant for each state, i. e. it assigns a clock constraint to each state. I write an individual transition as $A \vdash q \xrightarrow{g,a,r} q'$ where:

- q and q' are a start and target state;
- g , a clock constraint, is the guard of the transition;
- a is an action label (used for communication in networks);
- and r is the set of clocks to be reset when the transition is taken.

A configuration of a timed automaton is a pair (q, v) consisting of a state q and a valuation v . These are the semantic states of a timed automaton. The operational semantics define two kinds of transitions on configurations.

Delay There is a *delay* transition $(q, v) \xrightarrow{d} (q, v \oplus d)$ if $d \geq 0$ and $v \oplus d \models \mathcal{I}(q)$.

Action There is an *action* transition $(q, v) \xrightarrow{a} (q', [r \rightarrow 0]v)$ if $A \vdash q \xrightarrow{g,a,r} q'$, $v \models g$, and $[r \rightarrow 0]v \models \mathcal{I}(q')$.

A *run* of a timed automaton is an arbitrary sequence of delay and action transitions. Note that consecutive delay transitions can always be combined into a single delay transition. Moreover, between two consecutive action transitions, a delay transition with time value 0 can always be inserted. Thus, typically one only considers runs that are a strict interleaving of delay and action transitions, as given by the following transition system:

$$(l, v) \rightarrow (l', v') = (\exists d \geq 0. \exists a v''. (l, v) \xrightarrow{d} (l, v'') \wedge (l, v'') \xrightarrow{a} (l', v')).$$

Some general remarks about this semantics are in order. First, many extensions of timed automata have been studied [34, 43]. Generally, most extensions render model checking questions for TA undecidable [34, 46], with the notable exception of priced timed automata [43]. Thus, TA can be said to sit at a certain sweet spot between expressivity and decidability.

One restriction of TA is particularly important, namely *diagonal-free* TA. In diagonal-free TA, difference constraints of the form $x - y \sim c$ are disallowed (in guards and invariants). This is important because Bouyer showed [35] that the typical DBM-based model checking algorithms for TA are incorrect when difference constraints are allowed. Therefore, the restriction to diagonal-free TA is usually made. Luckily, this does not mean that we will lose in expressivity: any TA can be translated to an equivalent diagonal-free TA [2, 20]. The translation may lead to an exponential blowup in the size of the automaton, however. There are also more practically efficient methods for model checking TA with difference constraints [19, 36, 73]. Still, in practice, mostly TA without difference constraints are used. For these reasons, like many authors in the field, I restricted this thesis solely to the study of diagonal-free TA.

Finally, a particularity to consider in connection with the semantics of formalisms for timed systems in general is the notion of *Zenoness*. A run of a TA is *Zeno* if it makes infinitely many transitions in only a finite amount of time. It is generally agreed upon that such runs should be excluded from consideration because they are not realizable in practice. There are multiple ways to handle this complication. First, any timed automaton can be converted into a *strongly non-Zeno* automaton that does not allow any Zeno runs but preserves all other behaviors. This conversion can lead to an exponential blowup (in the size of the automaton) however [96]. Thus, alternatively, specific model checking methods that try to avoid this exponential blowup in practice can be used [96]. The most common approach is to simply consider automata that allow non-Zeno runs to be incorrect models because they allow unrealizable behaviors. This condition can be checked algorithmically [180, 93]. Therefore, most authors work under the assumptions that all automata are strongly non-Zeno. I follow this convention (where relevant).

Usually, not a single timed automaton but a network of timed automata is considered. I will not give a precise definition for the semantics of such networks but rather allude to intuition. Essentially, such a network of automata is equivalent to a single TA, the *product automaton*. It uses vectors of states, with one dimension to record the current state of each automaton in the network. Each internal transition of a single automaton gives rise to a transition in the product automaton, where only the state of this single automaton is updated in the state vector. Synchronization over a channel with matching input and output transitions yields transitions in the product automaton that update two components of the state vector at the same time. The initial state of the product automaton is the vector of all initial states.

The product automaton for the TA in Figure 2.1 is trivial because the automata synchronize on every transition: it would look exactly like the automaton for the light switch with every state replaced by a state vector that has *idle* as its first component. Similar techniques can be used to add many more features to the formalism, such as shared program variables or other notions of communication.

Finally note that the network in the example has a slight semantic oddity. Because of the invariant on the *bright* state and because the automata have to synchronize on every transition, the user is effectively forced to press the button another time after at most 10 seconds. In a more realistic model, we would therefore add another transition from *bright* to *off*, which does not have to synchronize on *press*. I have omitted this to simplify the following examples.

2.3 The Model Checking Task

We are now ready to formulate model checking questions for TA. I will focus on the aforementioned reachability question for TA for now. It is whether a certain *final* state q_f can be reached from the initial state q_0 . For networks of timed automata, one often considers the question whether a state vector can be reached, in which a single state automaton is in a final state. In the example from Figure 2.1, for instance, we could ask whether the light can ever turn bright, i. e. whether we can reach a state where the light switch is in state *bright*. For this model the answer is “yes” because the user can press the switch with arbitrary speed. If we model the user as in Figure 2.2, however, and add a second clock y and a state *thinking*, then the user is always too slow and the answer is “no”.

For now, the model checking task I will focus on, is to provide an algorithm that can answer such reachability questions. This is not trivial as there are infinitely many clock valuations and thus there are (usually) infinitely many states in the semantics of TA. The rest of this section will give a brief overview of the most prominent approaches to derive such an algorithm. In

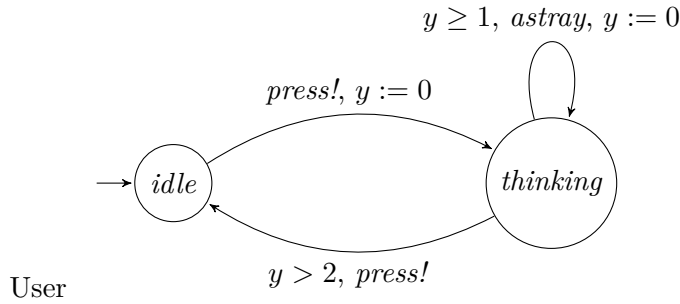


Figure 2.2: A modified version of the user model from Figure 2.1.

general, the model checking task is to check whether the model fulfills a formula from a range of different (timed) temporal logics [53]. I will revisit this question in Chapters 4 and 5.

2.4 Regions

It was already discovered by Alur and Dill [2] that most interesting properties of timed automata, including the reachability task, can be decided algorithmically. At the heart of all these algorithms lies the so-called region construction [2]. In the following, I will briefly sketch the core ideas behind the construction. For a more gentle introduction, see Chapter 9 of Baier and Katoen’s textbook “Principles of Model Checking” [11].

Considering Figure 2.1 again, a first key observation is that we are never interested in values of x for which $x > 10$. The reason is that there is no guard in the automaton which makes a comparison against a larger number than ten. Therefore, we do not need to know whether $x = 10.1$ or $x = 10^{10}$. We are just looking for the fact $x > 10$. Moreover, it is also irrelevant whether $x = 1.4$ or $x = 1.999$. As long as the clock constraints in the TA are only made up of integer constants, valuations with either value of x will always satisfy the same constraints. These observations give rise to a *finite* partitioning of the clock valuations into the *regions*, i. e. distinct sets of clock valuations. These are the *points* $x = 0, x = 1, \dots, x = 10$; *intervals* of the form $0 < x < 1, 1 < x < 2, \dots, 9 < x < 10$; and *unbounded* regions capturing all valuations where the largest constant in the automaton has been exceeded, in this case the single region given by the constraint $x > 10^2$.

From these regions and the original TA, one can now construct a finite transition system, the *region automaton*. The states of this transition system are pairs of a state of the underlying TA and a region. There is a transition between two states of the region automaton if there is a transition in the semantics of the underlying TA between any of the valuations contained in the region. A part of the region automaton for the light switch example is depicted in Figure 2.3. As there are only finitely many regions, the region automaton is also finite. This means that we can easily decide the reachability question from the region automaton: we simply need to check whether a state in the region automaton that contains a final state of the TA is reachable. This task can be handled by standard search algorithms for graphs. In the example, we can see that the state (*bright*, $0 < x < 1$) is reachable in the region automaton and thus we know that the state (*bright*, $x = 0.1$) is also reachable in the TA.

The disadvantage of the region construction is that there are prohibitively many regions when the number of clocks is increased as the number of regions grows exponentially in the number of clocks. This renders the construction rather useless for practical applications.

²An example for two clocks appears in the next section.

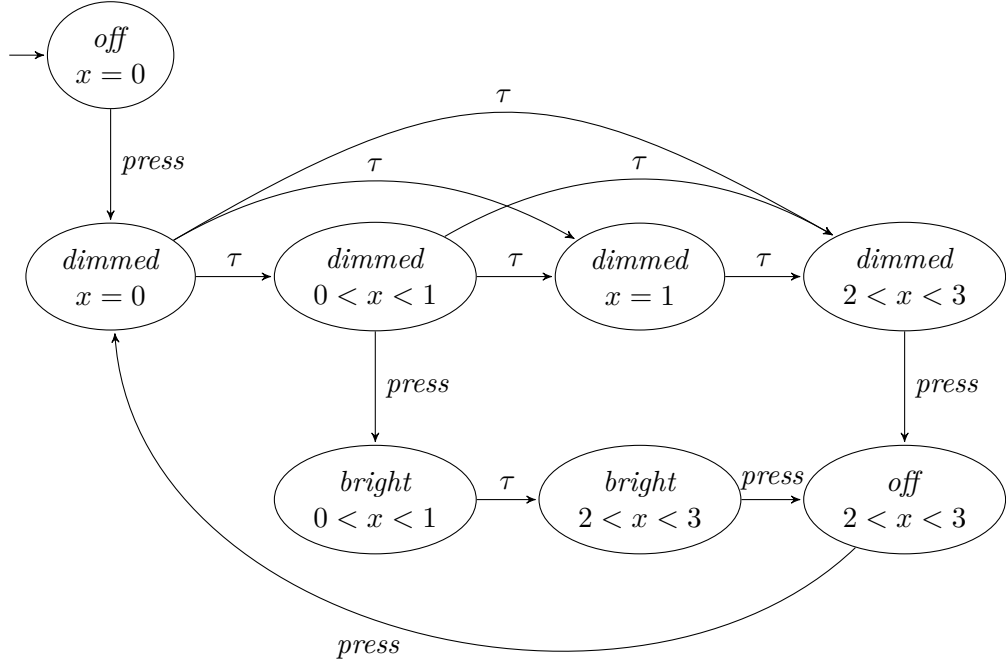


Figure 2.3: Part of the region automaton for the light switch given in Figure 2.1.

2.5 Zones and Abstractions

To overcome the limitations of the region construction, research on TA model checking has focused on finding larger “partitions” of the clock valuations to obtain smaller transition systems on which reachability can be decided. The basic idea is to consider so-called *zones* instead. A zone is a set of valuations that can be described by a clock constraint. This means that every region is also a zone. Also note that every zone is *convex*. Figure 2.4a depicts the zone given by $y \leq x \wedge x - y \leq 2 \wedge y \geq 1$. The state space of a TA can be explored symbolically with zones because it is possible to compute the following operations:

Arbitrary delay $Z^\dagger = \{v \oplus d \mid v \in Z \wedge d \geq 0\}$

Clock reset $[r \rightarrow 0]Z = \{[r \rightarrow 0]v \mid v \in Z\}$

Intersection with a constraint $Z \cap g = \{v \mid v \in Z \wedge v \models g\}$

By “compute” I mean here that these operations that were defined semantically on sets are operations that again yield zones. We are interested in the *strongest postcondition* of a transition $q \xrightarrow{g,a,r} q'$. For a given state Z , it consists of all the valuations v that are reachable from any valuation $u \in Z$ by taking any delay transition followed by the action transition $q \xrightarrow{g,a,r} q'$. The strongest postcondition can be computed as follows:

$$Post(q \xrightarrow{g,a,r} q', Z) = \{[r \rightarrow 0](Z^\dagger \cap I(q) \cap g) \cap I(q')\}.$$

Similarly to the region automaton, we can then define a transition system on zones. The reachable part of this transition system is called the *zone graph*. Figure 2.5 depicts the zone

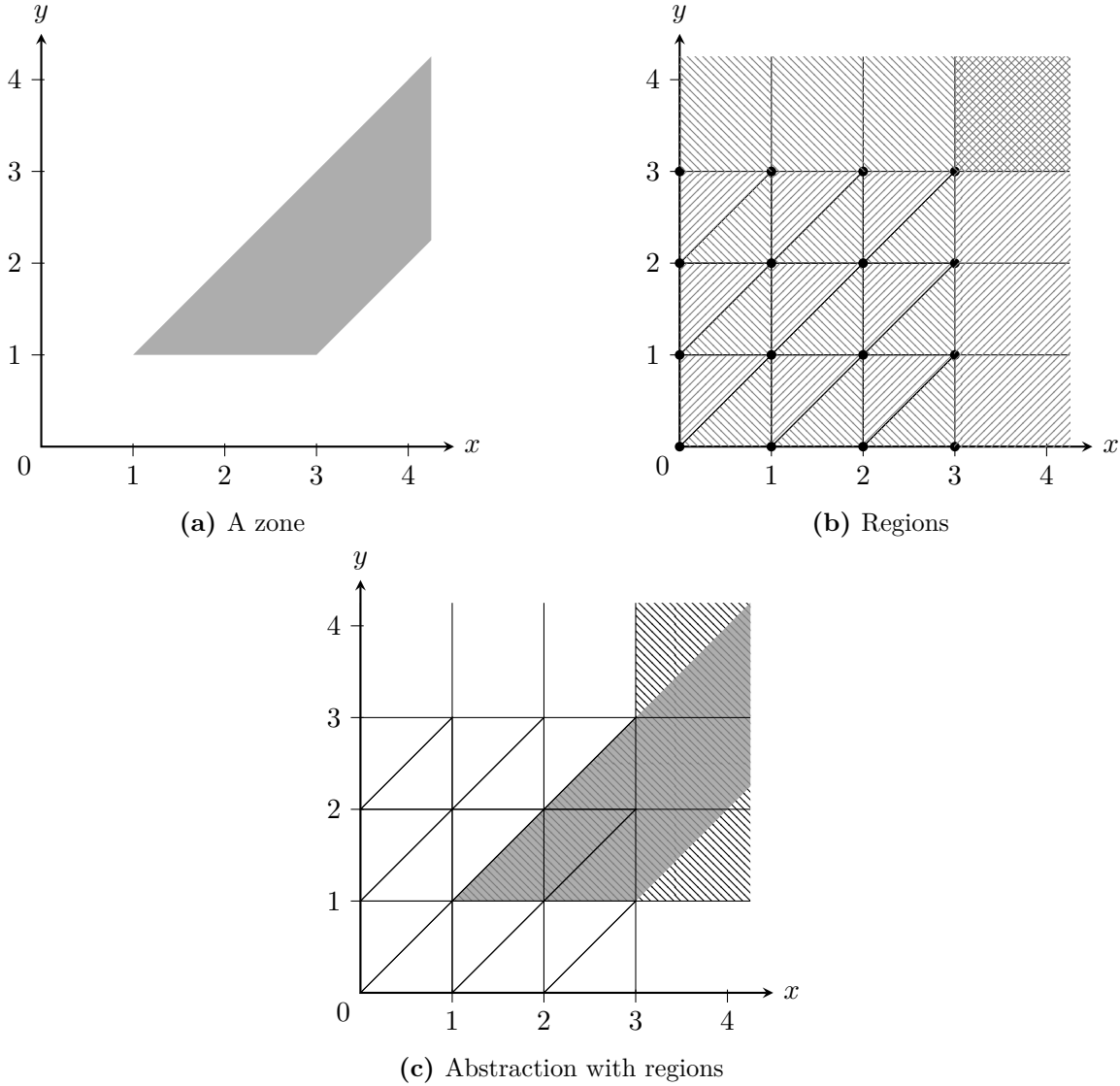


Figure 2.4: Illustration of (a) the zone given by the constraint $y \leq x \wedge x - y \leq 2 \wedge y \geq 1$, (b) the set of regions for two clocks x and y with the upper bound three (given as points, solid line segments, and hashed areas), and (c) the abstraction of the zone with its covering union of regions.

graph for the TA from Figure 2.1³. It can be seen that it is smaller than the region automaton and that we can decide reachability properties just as easily from it.

This need not always be the case, however. Figure 2.6 shows part of the zone graph for the system consisting of the user model from Figure 2.2 and the light switch model from 2.1. We can see that it is infinite, making it useless for algorithmic exploration. To overcome this problem, so-called *abstractions* are used [59]. An abstraction is an operation that computes an over-approximation of a zone. It should have the property that only finitely many such over-approximations can be computed. The zone graph where an abstraction is applied to every node is called the *abstract zone graph*. It is always finite. An abstraction should also be *precise*, in the sense that every run of the TA can be simulated by a path in the abstract zone

³The state component for the user is not shown as there is only one state. I will follow this convention for the remainder of the thesis.

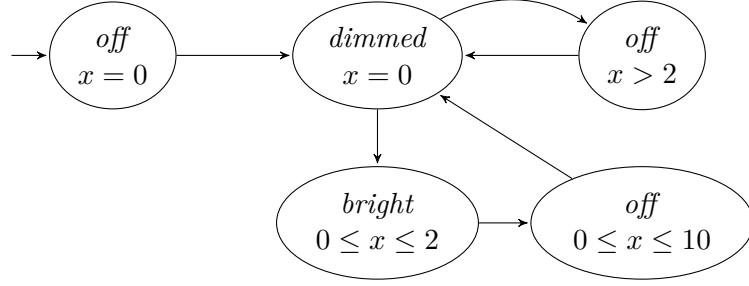


Figure 2.5: The zone graph of the TA from Fig. 2.1.

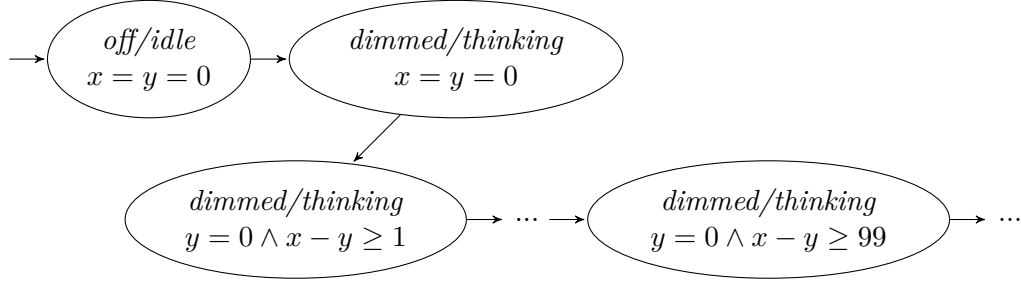


Figure 2.6: Part of the (infinite) zone graph of the TA from Fig. 2.2.

graph (which goes through the same locations) and every path of the abstract zone graph can be instantiated to a corresponding run of the underlying TA.

I will introduce one such abstraction by example. Assuming that the maximal constant in the automaton is 3, Figure 2.4b gives the regions for two clocks x and y . The abstraction computes the union of regions that cover a zone. This is illustrated in Figure 2.4c (for the zone from 2.4a). The abstraction is sound but not very useful⁴: as we can see from the picture, this abstraction is not necessarily convex and thus not a zone. Instead, other abstractions that always yield zones have been introduced [59, 13, 14]. These are often called *extrapolations*. It is these extrapolations for which Bouyer discovered soundness issues [35]. We will revisit the topic of abstractions in Chapters 3 and 5.

2.6 Difference Bound Matrices

In practice, zone-based model checking algorithms for TA represent zones with the data structure of *Difference Bound Matrices (DBMs)*. I will now give a brief explanation of this data structure. Every clock constraint can be seen as a weighted graph with the variables as nodes and where the edges represent difference constraints on these variables. Figure 2.7 illustrates this for the clock constraint $y \leq x \wedge x - y \leq 2 \wedge y \geq 1$. It is first converted into the equivalent set of difference constraints $y - x \leq 0 \wedge x - y \leq 2 \wedge 0 - y \leq -1$. An edge from x to y with weight c is added to the graph if there is a difference constraint of the form $x - y \leq c$. To encode constraints that only compare a single clock to a constant, a special node $\mathbf{0}$ is added to the graph, representing an artificial variable that always has the value 0. Then the graph in Figure 2.7a can be read off directly. Any path in the graph corresponds to a difference constraint that is implied by the original clock constraint. For instance, from the path $\mathbf{0} \xrightarrow{-1} y \xrightarrow{0} x$, we can derive the constraint $0 - x \leq -1$ (or $x \geq 1$).

⁴Note that many years later Herbreteau et al. have found a way to make practical use of it nonetheless, even if only implicitly [95]. This will be relevant in Chapter 5.

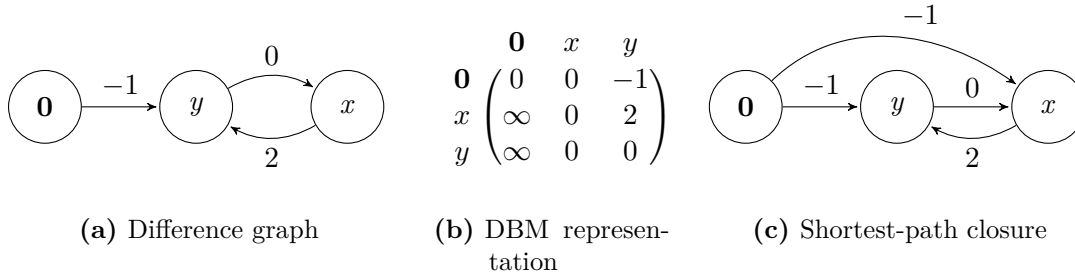


Figure 2.7: A graph representation of the zone $y \leq x \wedge x - y \leq 2 \wedge y \geq 1$ (a), its representation as a DBM (b), and its shortest-path closure (c).

We can also represent such a graph in matrix form as in Figure 2.7b. This matrix form representation is called a DBM [62]. The advantage of this representation is that all the operations that are necessary to compute the strongest postcondition of a transition can be computed efficiently on it. There is also a *normal form* for DBMs, which corresponds to the shortest-path closure of the underlying weighted zone graph. For the example, the shortest-path closure is depicted in Figure 2.7c. On DBMs, it can be computed with the Floyd-Warshall algorithm [19]. Viewed from the angle of difference constraints, the normal form of a DBM represents the set of the tightest difference constraints that can be derived from the original DBM. Normal forms are convenient because the strongest postcondition of a transition can be computed more efficiently on them and the questions of emptiness and subsumption (whether one zone is a subset of another zone) can be decided easily on them. Therefore, model checkers usually maintain the invariant that all DBMs stored are in normal form.

Note that the above presentation has been simplified in that constraints like $x - y < c$ were not considered. To also include this type of constraint, DBMs do not store numbers but entries of the form $(<, c)$, (\leq, c) , and ∞ . Arithmetic on this domain can be defined in a straightforward way. Alternative data structures to DBMs to represent zones such as minimal constraints graphs have been considered but DBMs seem to be most efficient for model checking in practice (while other methods can decrease memory consumption) [19].

2.7 Other Model Checking Approaches

Other model checking methods do away with zones completely, and instead represent the (explored) state-space of a given TA *fully symbolically*. In comparison, the zone-based method can be described as semi-symbolic because the states of TA are represented explicitly in the zone graph, while the time-component is represented symbolically in DBMs. The fully symbolic methods fall into two categories: methods based on tailor-made data structures similar to *binary decision diagrams (BDDs)* [184] and methods based on general-purpose solvers for the *satisfiability modulo theories (SMT)* problem [142, 111]. While both approaches promise some convincing performance advantages, they are usually not used in practice. One reason is that implementations of these methods in “real-world” model checkers like UPPAAL are missing. Another is that, compared to the rich modeling language offered by UPPAAL for instance, the modeling formalism has to be severely restricted to apply these methods. Therefore, I only considered the classic, semi-symbolic approach to timed automata model checking in my work. This is what I will refer to by TA model checking in the remainder of this thesis.

2.8 Summary

The following is a summary of the key points that I tried to convey in this section.

- *Transition systems* are labeled graphs that model a system. Their nodes are called states and their edges are called transitions.
- *Timed automata (TA)* extend transition systems with clocks to model time. Clocks can be reset when a transition is taken. *Clock constraints* are used in *guards* and *invariants*. Guards impose an enabling condition on transitions. Invariants restrict the idling behavior of TA.
- Typical system models consist of networks of TA. *Channels* are used for communication between automata. System models of this kind can be always be compiled into a single equivalent timed automaton.
- The semantics of TA are infinite. Decidability of reachability checking for TA was initially proven by providing a quotient construction, the *region construction*.
- The region construction still yields a very large transition system. Thus the most prevalent model checking method for TA uses *zones* to obtain the (expectedly) smaller *zone graph*. Zones are sets of *valuations* that can be represented by a clock constraints.
- The zone graph is typically not finite. Therefore, *abstractions* are used to calculate a finite set of over-approximations of zones called the *abstract zone graph*. Usually, *extrapolations* are used, which are abstractions that only yield zones as a computation result.
- *Difference Bound Matrices (DBMs)* can be used to efficiently represent zones. Normal forms of DBMs are maintained to efficiently decide the questions of *subsumption* (zone inclusion) and emptiness.

Finally, I want to explicate some terminology on the different notions of states that we have seen above.

- States of transition systems and of TA will both be called “states”. Alternatively, the terms “location” and “discrete state” are used in the literature and can also be found in some of my papers.
- The semantic states of TA will be named “configurations”. In my papers, they are also often described as “concrete states” or “semantic states”.
- The term “symbolic state” can either refer to states of the zone graph or of the abstract zone graph, depending on the context. The abstract zone graph will sometimes be referred to as the “symbolic state space”. All of these terms can be found in my papers.

This concludes the discussion of the most important preliminary knowledge on timed automata and model checking that is necessary to understand this thesis. In the next chapter, I will discuss my work on formalizing most of this material in Isabelle/HOL.

3 Formalizing (Probabilistic) Timed Automata

In interactive-theorem-proving speak, *formalization* refers to defining concepts that pertain to a certain body of mathematics in a proof assistant and to formally prove theorems about these concepts in the proof assistant. Before I could verify any code that belongs to a verified model checker or certifier for TA, I had to be able to talk about TA in Isabelle/HOL. That is, I had to formalize their semantics. It is not immediately clear what this means, however. As we saw above, the abstract definition of the semantics would usually be given for a single automaton, while in practice one always wants to add more sophisticated features—networks with some form of communication at the very least. I see the latter in the realm of concrete model checking applications. Thus, I defer the discussion of this topic to the next chapter. Formalizing the semantics of single TA is rather easy to do, particularly in Isabelle/HOL where one of the proof assistant’s strengths is highlighted: its ability to introduce notation that closely resembles the one used on paper. The main part of the work that is reported on in Paper A is not the formalization of the semantics but the formalization of essentially all the constructions (excluding product constructions) from the last section. This lays the basis for the verification of the tools that I produced later. In the following I will first summarize these results and then elaborate on an extension of this work to probabilistic timed automata (Paper B).

3.1 Regions

For the formalization of most of the material, I relied on Patricia Bouyer’s work [34]. The main reason is that it offers by far the highest mathematical precision that I was able to find anywhere in the literature. This naturally provided the best basis for formalization as the proof assistant forces one to work with utmost mathematical precision. Bouyer does not only define the classic region construction of Alur and Dill [2] but also a refined set of regions, which has the particular property that any union of regions that covers a zone is again a zone. This is later used for the correctness proof of a classic extrapolation.

The main result I proved is that both types of region constructions provide a correct way to decide the reachability problem for TA. This result is not only relevant as a building block for later results about practical TA model checking, but it is also interesting in its own right as a computer-checked proof of this classic and important result for TA.

I will describe one particular aspect of the formalization of this result that turned out to be interesting after finishing the work on the first paper. The region construction (as well as the extrapolations) can be parametrized with additional information about the TA. So far, we have only considered a “cutoff” at the largest constant that appears anywhere in the automaton. Considering the example from Figure 2.2 again, we can see that for clock y we are interested in much fewer regions than for x because the largest constant y is ever compared to is 2. Therefore, a refined construction can use an individual bound for each clock. I already used this variant in my first formalization [191]. The idea can be pushed even further. One way is to consider *local bounds* for each clock [13]. That is, for each state, a different set of bounds can be used depending on which constraints are relevant for the portion of the TA that the state is in. Consider the state *off* in the example from Fig. 2.1. It is clear that the

actual value of x is irrelevant there as there is no invariant on the state and the state’s only outgoing transition has no guard and resets x . Thus, in this extreme case, a bound of $-\infty$ can be used for *off* (meaning that only one region remains, the unbounded region). I later also added this refinement to the Isabelle/HOL formalization. My positive finding was that only miniscule changes were necessary, which one would not necessarily expect after consulting the non-trivial pen-and-paper proof [13].

3.2 Zones and Difference Bound Matrices

The next essential result is the formalization of the zone graph. If you read Paper A, you will find that I do not make a distinction between sets of clock valuations and zones (recall that only the sets of clock valuations that can be represented by a clock constraint are zones). This is also reflected in my formalization of this result. I first formalized a purely semantic version of the zone graph where zones are simply sets of valuations, and transitions compute strongest postconditions. Proving the correctness of this is almost trivial.

In the next step I needed to show that strongest postconditions of actual zones again yield zones. For this, I did not work with zones as represented by clock constraints but rather worked on DBMs directly. I proved that the typical model checking operations on DBMs can be used to compute the strongest postcondition of a zone. DBMs are an equally valid representation of zones and my goal was to prove that the typical DBM-based model checking is correct anyway. This way, duplicated work could be avoided. A key element of this work was the formalization of the Floyd–Warshall algorithm [70, 185], which is used to compute the shortest-path closure of DBMs. I proved the two key facts about the algorithm: shortest path weights are correctly computed and negative cycles are detected by computing a negative entry on the diagonal of the weight matrix. To the best of my knowledge this was the first formalization of this classic algorithm on weighted graphs in a proof assistant. There were however formalizations of the Warshall algorithm for Boolean matrices [158, 21, 22, 186, 67].

An interesting addition that I made later but which is not published anywhere comes from the motivation of *deadlock checking*. The zone-based exploration algorithm I sketched in Section 2.5 explores the zone graph in a forward manner by repeatedly computing the strongest postcondition, *Post*, of symbolic states. Conversely, *Pre*, the weakest precondition of a zone for a given transition can also be computed on DBMs. On the one hand, this is used in (less popular) model checking approaches that explore the state space in a *backwards* manner [204, 206, 66]. On the other hand, the *Pre* operation is important for deadlock checking. There, given a symbolic state, we want to know whether any of the configurations it represents is deadlocked, i. e. whether it has no successor states. The solution is to inspect all outgoing transitions of the given state, and to compute the *Pre* for each transition given its target invariant. The union of all these symbolic states contains all the configurations that are not deadlocked [180]. Thus one just needs to check that the given symbolic state is a subset of this union. This is another non-trivial operation on DBMs, which I have also verified (a description of the operation can be found in Tripakis’ thesis [179]). All in all, I have fully verified the deadlock checking operation, and the operations necessary for backwards exploration of DBMs. Together with the aforementioned operations for computing *Post*, this yields a complete verified DBM library for TA. In the next chapter (in Section 4.1), I will discuss how this work was turned into an efficiently executable verified DBM library.

A complication that shows up in Paper A is the use of clock numberings on DBMs. In concrete automata, as the ones displayed above, clocks use human-readable names, of course. Similarly, in the formalization of TA, I did not specify from which domain the clocks of an automaton are drawn. Conceptually, this is sensible, but it turns into a problem when DBMs

come into play. The reason is that (in the formalization and in implementations but not in the examples above) the rows and columns of DBMs are indexed by natural numbers. Thus clocks somehow need to be identified with natural numbers. I resolved this by defining the semantics of DBMs (in the sense of the sets of clock valuations they represent) with respect to a clock numbering: an injective mapping from clocks to natural numbers. This leads to complications in proofs and additional assumptions about the clock numberings need to be carried around everywhere in the DBM formalization. Initially I thought this would pay off by giving me more flexibility when implementing a model checker for TA. However, it soon became clear that one wants to convert a given model such that all labels are replaced by natural numbers as a preprocessing step anyway. Thus, looking back, I should have dropped the clock numberings altogether, and just assumed that clocks range over the natural numbers whenever DBMs get into play. In fact, I have done so when formalizing the operations for deadlock checking, and some proofs could be drastically simplified compared to the forward versions.

3.3 Extrapolation

The greatest challenge of this formalization effort was to prove the correctness of the classic extrapolation operation on DBMs. As mentioned above, this does not really come as a surprise as establishing the soundness of this operation has had a history of incorrect proofs, until the question was finally resolved by Patricia Bouyer [35, 34]. She showed that the operation is incorrect in the general case but provided a soundness proof for the case of diagonal-free timed automata. Stumbling on her work almost was an epiphany for me—before I had followed a long trail of references to the literature that provided loose proof sketches that I could just never fully understand (and that I was not meant to understand as they were trying to prove a non-theorem). Formalizing her work still was a big effort as the proof of her main theorem¹, stating that the extrapolation of a DBM always yields a subset of the union of regions that covers the DBM, is highly technical on paper and no less so in Isabelle/HOL.

In the course of the work on certification (Paper F), I formalized a way of proving the correctness of the extrapolation that is quite different from Bouyer’s initial work [34]. It stems from the seminal paper on an improved extrapolation that considers different cutoffs for lower and upper bounds of clocks (called *LU*-extrapolation) [15]. With this approach, the main difficulty is still in verifying said main theorem, but it means that it is now also possible to verify the improved extrapolation. I extend my gratitude towards Joshua von Mutius, who adopted the main theorem to the setting of *LU*-extrapolation under my supervision. This extrapolation is still the best (i. e. coarsest) that is used by the popular state-of-the-art model checker UPPAAL.²

3.4 Probabilistic Timed Automata

This section will summarize the work of Paper B, which can be seen as a spin-off of the main topic of this thesis. I will briefly motivate this now. It is often desirable to also reason about aspects of uncertainty. For instance, considering the example of a communication protocol again, we might have empirical knowledge that .1% of all packets sent over a network link are lost. We might then want to incorporate this knowledge into our model, and qualify our formula to ask:

¹I am referring to Proposition 2 of her paper “Forward Analysis of Updatable Timed Automata” [34].

²Although coarser ones exist in principle, they are not regularly used in practice.

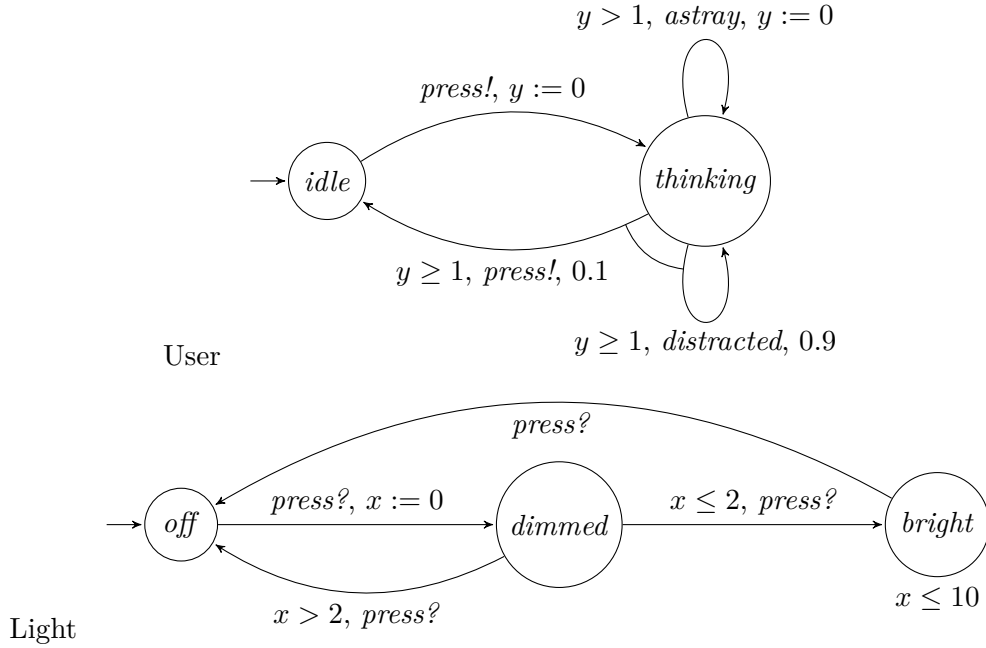


Figure 3.1: A PTA model of the light switch example.

With a probability of 99.9%, will the sender always get a confirmation of delivery within 10 seconds after the message has reached the recipient?

One way to view this is that it may be desirable to reason about uncertainty in a quantitative instead of a qualitative way, i. e. by moving from nondeterminism to probabilities. As in the example above, instead of asking *whether* a certain state can be reached, one would like to ask *with what probability* the state will be reached. Probabilistic timed automata (PTA) are an extension of timed automata to this probabilistic setting. In PTA, every transition still has a single guard but the clocks to be reset and the target state are determined by a probability distribution.

Consider again the light switch model from Figures 2.1 and 2.2. We assume that the user is very distracted today, so whenever they want to take the transition back to the idle state, operating the switch, they only do so with a probability of 0.1; otherwise they get distracted and continue deliberating. A PTA that models this behavior is depicted in Figure 3.1. We can now ask for minimum and maximum reachability probabilities: one can imagine different so-called adversaries or schedulers that choose which transition should be played from a state given the current history of states that were visited. For the example, the adversary that always chooses to play the self-loop in state *thinking* gives the minimum probability 0—just as in the case for the regular TA from Figure 2.2, the *bright* state will never be reached. The scheduler that always waits for exactly one second and then immediately plays the transition back to *idle* (with the other branch looping back to *thinking*) gives the maximum probability of 1: the system may go back to a *dimmed*/*thinking* state many times (via *off*/*idle*) but eventually *bright*/*idle* will be reached.

As we have seen, schedulers are used to resolve the nondeterminism contained in a PTA. This way, the semantics of PTA can effectively be given in terms of Markov chains but I will not delve into these details here and stick to the informal description. What is more, this is not the way in which Johannes Hölzl and me formalized the semantics of PTA. Rather, we directly started from his formalization of *Markov Decision Processes (MDPs)* [102]. MDPs are PTA without clocks. That is, these systems still exhibit nondeterministic and probabilistic

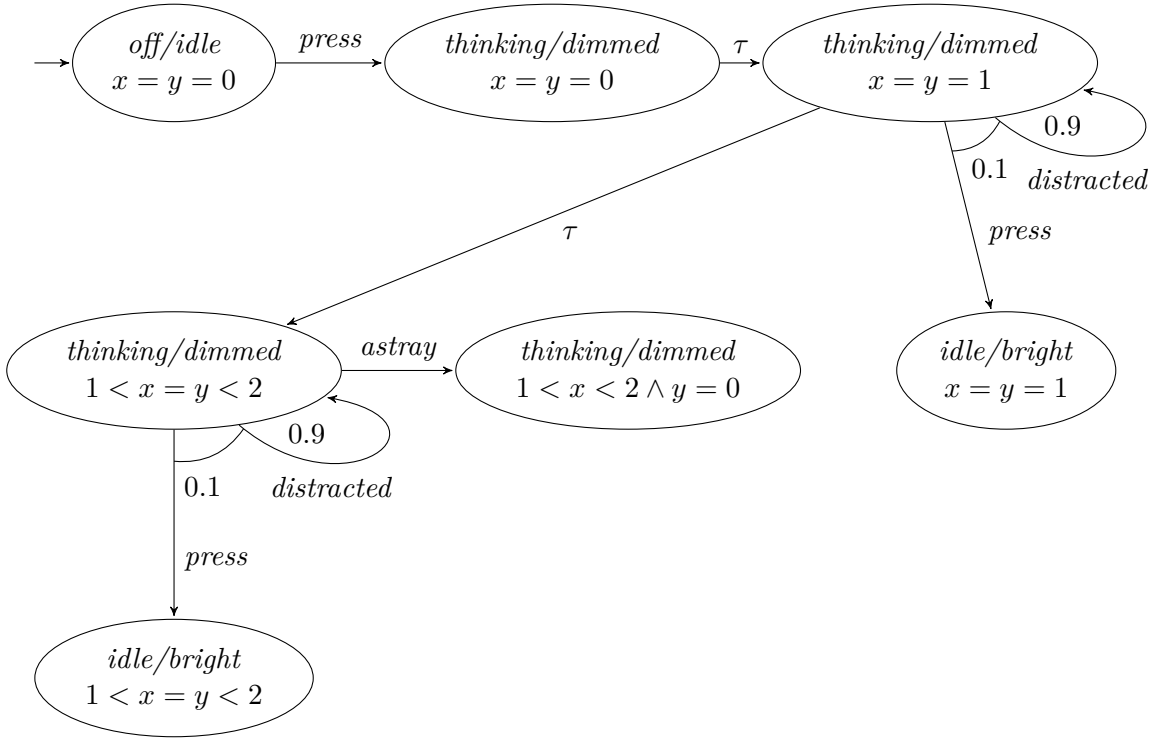


Figure 3.2: Part of the region automaton for the light switch given in Figure 3.1.

behavior but have no realtime component. Naturally, we formalized things the other way round, interpreting PTA as infinite MDPs. Next, we formally proved the essential result for PTA: minimum and maximum reachability probabilities can be computed. This result again invokes the region construction. Using it, we can construct a finite region MDP, as exemplified for the PTA from Figure 3.1 in Figure 3.2. The key property of this region MDP is that the minimum and maximum probabilities for reaching any *bright* state are the same as for the original PTA (as they are for any other reachability query). With existing algorithms for MDPs [11], minimum and maximum reachability properties can then be computed on the region MDP. One of our main contributions in this paper was to prove a probabilistic bisimulation property (using probabilistic couplings) between the original PTA and the MDP. This property is strong enough to derive the result for reachability probabilities from it. The approach we took is interesting in that it is quite different from the more casual arguments that would be made on paper but it also neatly expresses the key reason for the correctness of this construction in a single theorem.

Another interesting feature of our formalization of PTA is that we also considered the question of Zenoness. A scheduler is *Zeno* if it admits *Zeno* runs. Recall that a run is *Zeno* if it passes an infinite number of action transitions in finite time. As these behaviors are considered unrealizable, one often wants to exclude them when calculating minimum and maximum reachability properties. One solution is to restrict oneself only to structurally non-Zeno PTA for which no *Zeno* schedulers exist (as I do for regular TA). Marta Kwiatkowska [121] has also provided a solution where such schedulers can be excluded algorithmically. She gives a criterion, which can be decided algorithmically, that identifies exactly those schedulers of the region MDP which correspond to *Zeno* schedulers of the underlying PTA.³ These

³This criterion goes back to a criterion for regular TA given by Bouajjani et al. [33].

schedulers can then be excluded from the computation of minimum and maximum reachability properties on the region MDP.

Interestingly, in our formalization of this result, the part of the proof that revealed itself to be harder is the one that appears to be easier (or even trivial) on paper. One essentially needs to show that every Zeno scheduler of the PTA is simulated by one on the region MDP, and the other way round. The second part is more intricate as the schedulers identified as Zeno on the region MDP could be realized by many different schedulers of the PTA, including non-Zeno ones. Thus one has to carefully construct the bisimulation such that these schedulers of the MDP always correspond to Zeno schedulers of the PTA.

Overall, I consider it the main strength of this work that it neatly combines our two existing formalizations of MDPs [102] and TA [191] to obtain a formalization of PTA. This is summarized in the formula “ $\text{MDP} + \text{TA} = \text{PTA}$ ” used the paper title. The formula applies for the basic definitions of PTA and the region MDP on the one hand. The formalization of these parts can be considered to be even more concise than a pen-and-paper definition of these concepts. On the other hand, the formula is also reflected in our bisimulation argument, which consequently separates discrete TA-related reasoning from probabilistic MDP-related reasoning.

The result we formalized only proves decidability, of course. As for TA, many practical approaches to model checking PTA rely on zones [120, 153]. We hope that correctness of these methods could also be verified in a similarly elegant way. My preliminary investigations in that direction had two results: some published proofs have turned out to be wrong but the authors could provide corrections for them; and it is clear that our bisimulation argument needs to be strengthened for the correctness proofs of zone-based methods.

4 Constructing a Verified Model Checker for Timed Automata

In the previous chapter, I have reported on my formalization of timed automata and the region construction. The main result, however, was a formal proof of the fact that symbolic DBM-based forward exploration with extrapolation can be used to correctly decide reachability queries for TA. Starting from this, my next goal was to construct a verified model checker for TA. One might ask what there is actually left to do at this point. After all, what I have presented above was roughly a formalization of all the material that one would find in a tutorial on TA model checking, such as the one of Bengtsson and Yi [19]. Normally, the next step could be to just go ahead and implement all these abstract algorithms in a tool. The crux is of course that I want to establish a formal connection between the theory and the tool, resulting in a correctness proof for the tool in Isabelle/HOL. In the formal setting, this is quite some distance to bridge.

The same is also true in the informal world though. A real model checker chiefly needs two things: an expressive modeling language and efficient implementations of the model checking algorithms. There is also quite a gap between these and the theory in the informal setting. It is thus a significant challenge to achieve both of these goals also in a formal setting. The first attempt I made on this task together with Peter Lammich is described in Paper C. Later I made some improvements to this work and added a third set of features that is important for a real model checker: essential convenience features that make a verification tool viable as a practical tool. These include a graphical user interface (GUI), a human-readable input format, and output of diagnostic information. I report on the resulting tool, MUNTA, in Paper D.

In this chapter, I will first summarize the main techniques that we used to get efficient implementations of the DBM algorithms that lie at the heart of TA model checking. They can all be summarized under the same banner: *refinement*. Next, I will outline the verification of the main model checking algorithms. Then, I consider two modeling formalisms that were implemented and sketch how the product construction for them is verified. Finally, the more practical sides of MUNTA are briefly explored.

4.1 Refinement

In the situation that I described above, we have the typical setting for refinement. An abstract specification of an algorithm (forward exploration of TA with zones) is to be turned into a concrete, efficient program (the model checker). A programming paradigm to tackle this task has been proposed by Wirth and others already at the beginning of the 70's: *stepwise refinement* [199]. The idea is that the programmer starts with an abstract specification of an algorithm (say pseudocode), and gradually replaces parts of it with concrete and more and more efficient operations and implementations until they arrive at the desired efficient program. This gets particularly interesting in the setting of verification, where correctness is scrutinized. For the abstract specification of the algorithm it is—hopefully—comparatively easy to prove that it is correct. Then, in the subsequent refinement steps one only has to focus on the correctness of the individual replacements, not the algorithm as a whole. In

$$\begin{array}{ll}
up\ M = (\lambda i\ j. & up_1\ M = (\lambda i\ j. \\
\text{if } i > 0 \text{ then if } j = 0 \text{ then } \infty & \text{if } i > 0 \wedge j = 0 \\
\text{else } \min(M\ i\ 0 + M\ 0\ j)(M\ i\ j) & \text{then } \infty \\
\text{else } M\ i\ j) & \text{else } M\ i\ j) \\
\text{(a)} & \text{(b)} \\
up_2\ M\ n = fold & up_3\ M\ n = imp_for \\
(\lambda i\ M. M((i, 0) := \infty)) & (\lambda i\ M. mtx_set\ (n + 1)\ M\ (i, 0)\ \infty) \\
[1 \dots n]\ M & 1\ (n + 1)\ M \\
\text{(c)} & \text{(d)}
\end{array}$$

Figure 4.1: Refinement stages of the up operation for computing time successors.

practice, this separation of concerns can drastically reduce the burden of the verification task. A particular variant of refinement, where only data structures are replaced (a set might be replaced by a list, or a map might be replaced by a red-black tree, for instance), is *data refinement*. Hoare already studied how to automate this task in a correctness-preserving way in 1972 [100]. Only in recent years, this technique has been made accessible to users of Isabelle/HOL by Lammich and Lochbihler [123, 137, 127]¹. It shall be noted that one implicit refinement step is hidden from the remaining discussion: the extraction of actual SML code via Isabelle/HOL’s code generator [84].

Building on Peter Lammich’s previous work on refinement [123, 126], we have employed refinement in various ways to construct MUNTA, but there are also instances where the idea was used in a more ad-hoc manner. Refinement mainly shows up in two different places: refinement of DBMs and of graph search algorithms for model checking. The former will be the content of the rest of this section, while I will touch on the latter in the next section.

Figure 4.1 is taken from Paper C and simplified for readability. It displays the refinement stages of the up operation on DBMs.² This will serve as a prototypical example for the refinement chain that is applied to all DBM operations I formalized. At the starting point, the formalization of the abstract semantics, I had proven that up correctly computes the delay operation \uparrow of the zone represented by the DBM. We used stepwise refinement to move through implementations up_1 and up_2 to arrive at the goal up_3 . In the first three implementations, DBMs are represented in a mathematical way as mappings from a pair of indices to a DBM entry. The last implementation is an imperative program using destructive updates to an array and resembles what can be found in typical TA model checkers. The following refinement steps come in between.

Semantic Refinement In the purely abstract, mathematical formalization, there are certain features that cannot be implemented. For instance, constants in automata and DBMs are given as real numbers, and DBMs can have infinite dimensionality. Thus in a first step, this semantic “freedom” is restricted by fixing a set of clocks and representing constants by integers. To automate the latter step, parametric reasoning with Isabelle’s *transfer* tool is used [103].

Algorithmic Refinement The operation up (and others) can be optimized, by introducing the additional assumption that the given DBM is in normal form, yielding up_1 . This type of refinement is carried out manually. Later I have added further such “algorithmic”

¹Similar technology is also available in Coq [54, 60].

²The figure contains some specific Isabelle/HOL syntax, which I will not explain here as it is not important for understanding the rest of this thesis.

refinements to stop computation early when a negative diagonal (i. e. empty zone) is detected, or to avoid full runs of the Floyd-Warshall algorithm under certain circumstances. These refinements are carried out with a certain degree of automation using fine-tuned rule sets for standard Isabelle tactics.

Ad-hoc functional implementation The computations described by up and up_1 are still very implicit as they only specify what the resulting matrix will look like but they do not give us a computational rule to obtain this matrix. This is introduced in the form of a functional program in up_2 . These functional programs are written by hand but their proofs always follow the same structure and are highly automated.

Synthesis of an imperative implementation The functional program up_2 could now in principle be turned into an executable program by Isabelle’s code generator, but we want to go one step further to make it more efficient. Naturally, a realistic model checker would represent DBMs as arrays on the heap, and compute an operation like up by applying destructive updates to the given array. With the help of Imperative HOL [45], such programs can also be expressed in Isabelle/HOL. These can later be translated by the code generator to imperative features of the functional programming languages it can export code to. To obtain up_3 from up_2 , two things need to happen. The *fold* operation needs to be replaced with another looping construct and data refinement needs to be applied to the DBM to replace it with an array. Both steps can be carried out automatically (after some setup) by Peter Lammich’s *Seppref* tool [126]. Using it, we automatically synthesize up_3 from up_2 . Note that there is quite some non-trivial reasoning involved “under the hood” to justify the use of destructive updates. Internally, this reasoning is based on separation logic [166, 165].

4.2 Algorithm Verification

We are not done yet after implementing the DBM algorithms efficiently. The other core part of TA model checkers are search algorithms and related data structures that govern the forward exploration of the search space. One such search algorithm is the cyclicity checker that finds a pre-cycle, i. e. a path from a state to another state that subsumes it, in a given transition system (or reports the absence thereof). In a finite transition system, the existence of a pre-cycle implies the existence of an actual cycle. The pseudocode is displayed in Algorithm 1. It is phrased as a recursive depth-first search (DFS) algorithm. It uses a stack (ST) to keep track of the nodes that are on the current path and a *passed set* (P) to keep track of the nodes that have already been fully explored. If a subsumption to the stack is found, a pre-cycle has been identified. If the current state is subsumed by the passed set, it can safely be discarded: any pre-cycle that could be found from the subsumed state could also be simulated by a pre-cycle from the subsuming state. But as the subsuming state has already been added to the passed set, no pre-cycle starting from it can exist.

There are some interesting aspects to note about the algorithm. First, it is quite similar to Listing 1.1 in Paper D but the latter is (nearly) actual Isabelle/HOL code! In fact, the pseudocode is also very similar to the original variant given by Behrmann et al. [17]. Second, the algorithm is highly parametric and is defined in terms of only a few abstract parameters, a transition system \rightarrow and a subsumption relation \preceq . It can thus be applied beyond TA model checking. The original algorithm is stated in the nondeterminism monad of the Refinement Framework [123]. The framework gives us some assistance in verifying the algorithm by providing a verification condition generator. Note that hints can be provided to this tool by adding assertions as in line 12.

Algorithm 1 Cyclicity Checker

```

1: procedure DETECT-CYCLE( $P, ST, v$ ) ▷ Initially:  $P, ST$  empty
2:   if  $\exists v' \in ST. v' \preceq v$  then
3:     return ( $P, ST, True$ ) ▷ Pre-cycle detected
4:   else if  $\exists v' \in P. v \preceq v'$  then
5:     return ( $P, ST, False$ ) ▷ Subsumption to passed set
6:   else
7:      $push(v, ST)$ 
8:      $r := False$ 
9:     for all  $\{v' \mid v \rightarrow v'\}$  do ▷ Check all successors
10:       $P, ST', r := DETECT-CYCLE(P, ST, v')$ 
11:      if  $r$  then break
12:     assert  $ST' = ST$ 
13:      $pop(v, ST)$ 
14:      $P := P \cup \{v\}$  ▷ Add  $v$  to passed set since its subtree is fully explored
15:     return ( $P, ST, r$ )

```

After verifying this abstract implementation, we applied a number of refinement steps to obtain a version that uses an efficient data structure for managing the passed list. For this, we use a hash map that maps states of the TA to a list of DBMs that have been discovered for it. The crux is that the DBMs are represented as arrays on the heap, and thus the hash map has to store heap content. The involved separation logic tricks that we had to develop for this made for an interesting verification task in itself. The resulting implementation is quite close to our target, the “unified passed-waiting list” that is used by UPPAAL [18]. Ours, however, still has a higher memory footprint as DBMs cannot be shared between the state and the heap. Achieving this would still be an interesting task for future work but only little work has been done on verifying shared data structures in Isabelle/HOL. While this data structure is tailored towards the needs of TA model checking, the final version of the cyclicity checker is still parametric in the subsumption relation and the transition system. In the course of our work, we have developed a library of such verified, efficient search algorithms that could also find use in other applications.

4.3 Beyond Reachability

The attentive reader may wonder why I presented an algorithm checking for cycles in the first place in the last section. Indeed, for reachability questions, MUNTA uses a similar DFS algorithm that just looks for final states but does not need to identify cycles. Beyond reachability, MUNTA can also check properties from a small subset of the temporal logic CTL. These are the same as the ones supported by UPPAAL. The cyclicity checker is used as a component for checking these properties. Supporting these properties involved two main challenges. First, to verify the necessary search algorithms as discussed in the previous section.

Second, to extend the abstract formalization to show that zone-based exploration with extrapolations can also be used to correctly model check these properties. The main result one has to prove here is that the abstract zone graph has a cycle through a state with a property P if and only if the underlying TA has a run on which P holds infinitely often. To obtain this non-trivial result, I followed the work by Bouajjani et al. [33] and formally filled out missing pieces that one could consider “folklore theorems” in an informal setting. Only later did it come to my attention that Li has used a quite different route to obtain the same result

[133]. Li’s proof even yields a stronger variant of the result that applies to a larger number of abstractions. I have also formalized this work now, and it turned out to be noticeably easier while yielding the stronger result.

4.4 Modeling Formalisms

All the material that I have discussed in this chapter so far can be seen as just being related to model checking of a single automaton. Consequently, this is also how it is formalized in Isabelle/HOL. As one is typically not interested in model checking of a single automaton, a more powerful modeling formalism needs to be built on top. “On top” is meant quite literally here, in the sense that model checking of this formalism should be added modularly without touching the underlying formalization for model checking of a single automaton. As in the informal presentation I gave in Section 2.2, the reduction from the modeling formalism to the single automaton is achieved by a product construction.

I have considered two modeling formalisms: one that tries to mimic the formalism of UPPAAL, and one that is more bare bones but is similarly found in tools such as Rabbit [27], Red [184], TChecker [92], or Prism [119]. The former is described in Paper C, while the latter is touched upon in the tool paper on MUNTA (Paper D).

Apart from networks of timed automata with communication over channels, UPPAAL supports a large variety of features such as urgent and committed locations, various constraints that can be imposed on communication, and shared state that can consist of bounded integers and bounded integer arrays. In particular, the shared state can be manipulated with a C-like programming language when a transition is taken. This yields some difficulties. First, formalizing the full language in Isabelle/HOL would be quite laborious. Second, it is not immediately clear that the input still represents a timed automaton as programs may not terminate. I attempted to tackle the first issue by considering only an intermediate bytecode language that UPPAAL generates from the C-like language as a pre-processing step. To address the second issue, I added fuel to all executions of programs, i. e. by executing each program only for a certain maximum number of steps.

Still, this approach has some issues. First, the bytecode language is not publicly documented, thus I had to come up with a semantics for it by reverse engineering. Second, there is a semantic gap between the UPPAAL input and the bytecode input. In this gap lies the UPPAAL bytecode compiler, which would now need to be trusted. Of course, one could imagine adding a verified compiler of this kind later. That would be laborious, however, and directly interpreting the C-like language would then be the easier option. Second and foremost, the programs can also be used as guards and invariants, i. e. they cannot only refer to the discrete state but also to *clocks*! Hence, even if termination is disregarded (by introducing a fuel for execution), not every model with bytecode annotation yields a valid TA semantically. Intuitively, the issue is that unconstrained use of clocks in the programs would yield disjunctions of clock constraints on transitions and in invariants. As a remedy for this problem, I have applied a crude program analysis that allows only very restricted use of clocks in programs, but which is sufficient to allow for the programs used in standard examples. It shall be noted that UPPAAL has to solve the same problem and also rejects many programs. However, it remains unclear to me what analysis is used by UPPAAL exactly.

Due to these issues with the UPPAAL-style formalism, I later decided to settle for a less ambitious approach. There, I used a formalism that allows for shared state in the form of a number of integer variables just as before. The language to manipulate this state is more simplistic, however. This language allows for simple expressions of arithmetic and Boolean expressions. These expressions can be used in guards and invariants that refer only to the

shared state, and on the right-hand side of updates to the shared-state variables. Updates to clocks and clock constraints are strictly separated from the shared state part.

While this language is less powerful than what UPPAAL offers, it is similar to what many other tools use as well (as mentioned above). Moreover, it is sufficient for all standard TA model checking benchmarks that I have found so far. It can also be argued that for a verified tool like MUNTA, which should serve as a reference implementation or as a certifier for a possibly large range of other model checkers, it is reasonable to settle for a simple formalism that just captures the essence of typical modeling formalisms. The benefits gained are a simple and clear definition of the formalism’s semantics, and a simple and portable input format.

Nevertheless, this input language offers some additional features compared to the first attempt. These are additional synchronization primitives taken from UPPAAL [19], which are relevant as convenience features for practical models and for *state space reduction*: urgent and committed states, and broadcast transitions. Two of these features, committed states and broadcast transitions are directly implemented via the product construction. The third, urgent states, is added as a pre-preprocessing step that adds an additional clock to the automaton. This is beneficial as the computational overhead introduced is small and the size of the symbolic search space is only increased slightly due to abstractions, while it avoids additional complexity of the product construction.

A particularly interesting set of models that make heavy use of these features are the *Penn pacemaker models* [108]. They model different variants of a pacemaker with different feature sets. Their correctness is verified with respect to a heart model that can essentially act completely nondeterministically. These models serve as a basis for generating code to operate a pacemaker. I was able to fully verify them with MUNTA.

I have now discussed the two modeling formalisms that are supported by MUNTA. What is still missing is the product construction to be able to actually model check these formalisms. For both, I used the same idea, which is presented in Paper C. At the interface for the verified model checker for a single automaton that was described in the last sections, the automaton is given as two HOL functions: one for the invariants, mapping each state to a clock constraint; and one for the transitions, mapping each state to a set of outgoing transitions. The idea is to give two descriptions of the product automaton. One is a highly abstract mathematical description, and the other are the two HOL functions, which implement an efficiently executable version. Then one proves that the two descriptions coincide.

The trick of this specification as two HOL functions is that the product is automatically constructed *on the fly*. That is, an explicit product automaton is never constructed but its invariants and transitions are computed as the model checker explores the state space further and further. This is paramount for practical model checking. Unfortunately, the formal proof of coincidence of the two representations is highly laborious while not conveying any deep mathematical insights. I improved this in the second attempt (mostly out of need, as additional synchronization primitives complicate the constructions) by adding an intermediate refinement stage and introducing specialized tactics. Yet, I am still not satisfied with the complexity of these proofs compared to how simple the product construction is intuitively.

4.5 A Practical Tool

In the last sections, I discussed everything that is needed for a verified prototype model checker like the one we presented in the Paper C. To obtain something practical like MUNTA [196], some additions were necessary (Paper D). First, there is the ability to check models for deadlocks. On the one hand, this is necessary because all (other) correctness theorems for MUNTA rely on the assumption that the given input is deadlock free. On the other hand, in a

Munta Verified Timed Automata Model Checker

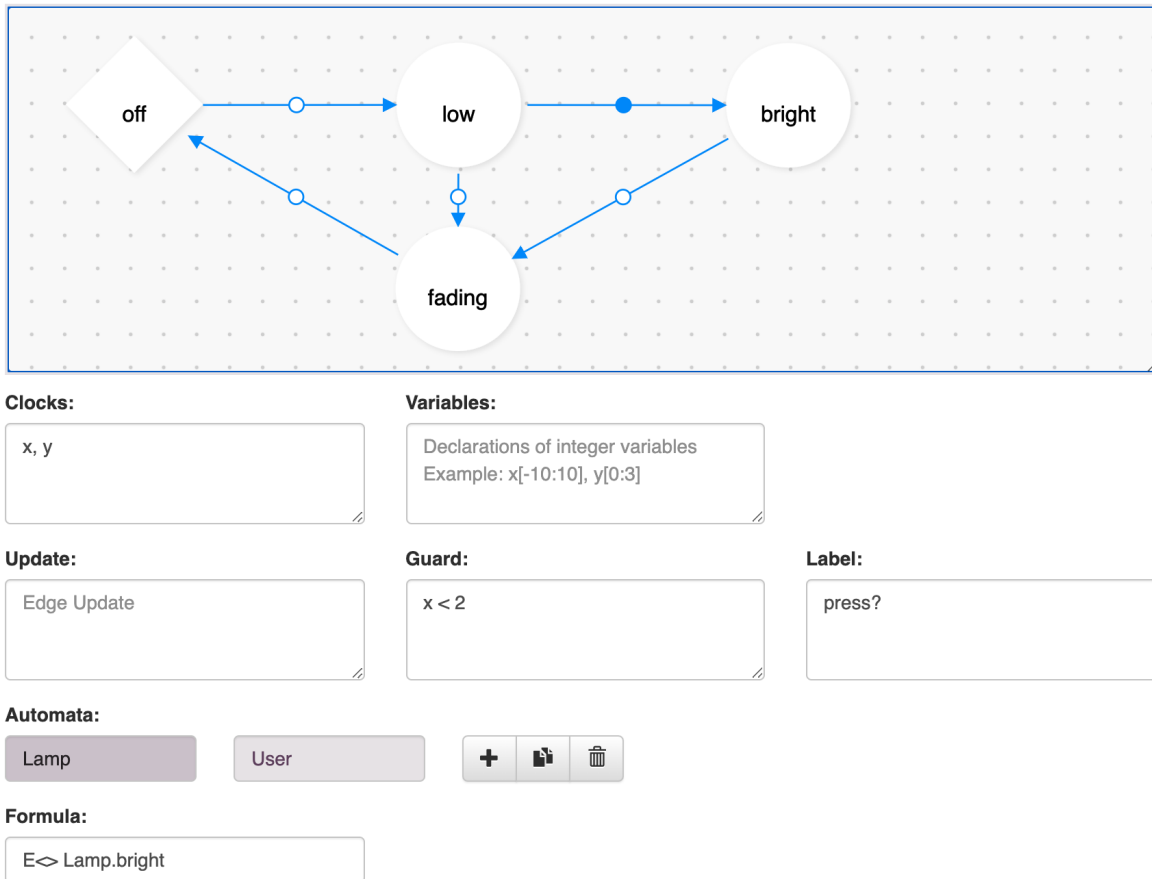


Figure 4.2: Screenshot of MUNTA's GUI showing a transition of a modified light switch example being selected for editing.

practical interaction with a model checker, absence of deadlocks is the baseline property that a user typically wants to check first to ensure that the model is sensible. I already discussed the required changes in Section 3.2.

The other additions fall into the categories of input and output. On the input side, I mainly introduced the new modeling language, together with a simple, portable, human-readable, JSON-based input format for it. On the output side, I added a number of convenience features: diagnostic information like reporting the number of explored states or emitting the whole set of explored states, and human-readable error messages for parsing and pre-processing input. Most prominently, both sides were tied up with a graphical user interface (GUI). A screenshot of the interface is given in Figure 4.2. The GUI runs in a web browser, and MUNTA itself can even be fully used inside the browser. This functionality is enabled by exporting code to OCaml instead of SML, which can again be compiled to JavaScript.

For all these features, the question of correctness immediately arises. In reporting errors, there is no issue because MUNTA does not claim a result in case of an error. To implement diagnostics, I use a simple logical trick to add them without jeopardizing correctness. Regarding the GUI and parsing, the state of affairs is more complicated. I use the same idea of a *parse-print-parse* loop to *validate* the correctness of both components. In this approach, a given input is parsed to an internal representation first, then printed out again, and parsed

back to an internal representation. It then is checked whether both internal representations match up. The same approach is used by the GUI: it converts the input to an internal JSON representation, displays it back to the user to double-check, and compares the JSON representation of that input against the original.

Of course, validation has only little value compared to verification. However, for a GUI, I do not see much that can be done. At the end, a human always has to ensure that the given input is really what they have in mind. Nevertheless, it would be reasonable to fix the JSON representation as a ground truth and to replace the parse-print-parse loop with a verified parser. In principle, this is possible as a significant body of work on verified parsing exists [12, 114, 109, 6, 129, 32, 64]. The issue is more of technical nature: there is currently no off-the-shelf solution for verified parsing in Isabelle/HOL. When it will finally emerge, it should certainly be used to equip MUNTA with a verified parser.

I have discussed two types of components of MUNTA so far. First, fully verified pieces of code that have been synthesized by the code generator (the greater part). Second, completely unverified pieces that are only validated. There is also a third type, *certified* components. These are bits of unverified code whose computation result is checked (internally) by a verified checker. For instance, I use this to compute local clock ceilings as a parameter for extrapolation [13] by running an unverified graph algorithm first and then checking that the computed parameters are valid.

I ran experiments with MUNTA on a set of standard benchmarks and compared the results to UPPAAL. They are reported on in Paper C. Generally, MUNTA is much slower than UPPAAL, which is to be expected as UPPAAL is a highly optimized tool implemented in C and C++. If the throughput is considered, the number of states explored per time unit, the results look encouraging: MUNTA's throughput is usually within an order of magnitude of UPPAAL's throughput. Furthermore, MUNTA is able to check medium-sized benchmarks in reasonable time. In this light, I consider MUNTA to be a usable reference tool for TA model checking.

5 Certifying Model Checking of Timed Automata

In the last section, I have discussed verification of a TA model checker as a way to get highly trustworthy model checking results. As mentioned in the introduction, I have also considered the alternative approach of certification, which will be the subject of this chapter. To recapitulate, in certification, an unverified model checker produces a result and a proof that the reported result is correct, the certificate. An independent tool, the certifier, is then given the model, the proposed property, and the certificate and checks that the certificate is indeed a valid proof of the fact that the property holds for the model. If correctness of the certifier is verified, we achieve the same level of trustworthiness as when verifying a full model checker. This is because the proof assistant still sits at the end of the trust reduction chain.

As hinted on in the introduction, certification has many advantages over full verification. First, the effort of verification can be significantly reduced by only verifying a certifier. For TA, the most prominent example of this is extrapolation. As discussed in Chapter 3, verifying the classic extrapolation operation was a highly laborious task. With certification, it is irrelevant to know which extrapolation was used during model checking. Instead it suffices to check that extrapolation always yields a larger state. This allows us to use a more sophisticated extrapolation technique [15] without verifying it. Second, the certifier can easily be parallelized (cf. Section 5.4). Finally, since TA model checking can exploit subsumptions, the number of states that are explored during model checking can be much larger than the number of states that is needed for a proof of the property. In this way, the task of certification can be computationally much cheaper than model checking. This is beneficial as the verified tool is always expected to be slower. Note however that complexity theory tells us that, in the worst case, checking certificates cannot be a computationally easier than TA model checking because the problem is PSPACE-complete [2].

In the following, I will first introduce certificates for reachability properties, mostly by example. Then, I will move on to liveness properties. These are characterized as so-called Büchi properties. I will discuss how the certificates for reachability can be extended to handle these properties. Next, I will outline an interesting result about both approaches, namely that they can be made compatible with the implicit abstraction technique of Herbreteau et al. [91, 95]. Finally, I will consider some important aspects of constructing a tool chain from an unverified model checker to the verified certifier in practice. The work of Joshua von Mutius and myself on reachability and the construction of a practical verified certifier is presented in Paper E. I later extended this to Büchi properties and the implicit abstraction techniques with Frédéric Herbreteau and Jaco van de Pol. This is reported on in Paper F.

5.1 Certificates for Unreachability

For reachability properties, a model checker can give two answers: either a state fulfilling the property is reachable or not. The positive result usually indicates a problem with the model signifying that a *bad* state is reachable. In this case, the model checker can assist the user in fixing the model by providing a *counterexample trace*, a trace of configurations yielding from

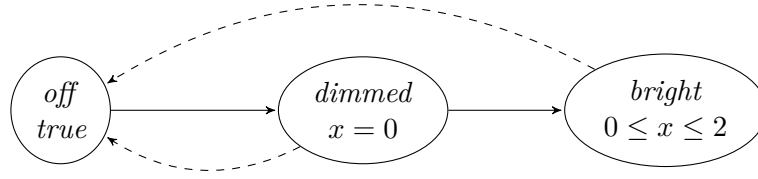


Figure 5.1: A reachability invariant for the TA from Fig. 2.1.

the initial state to a bad state. Counterexample traces can directly be used as certificates for reachability. Constructing a verified checker for such traces would certainly be feasible (and not very hard). The negative result, on the other hand, usually means a proof of safety: a bad state is not reachable and therefore the system is safe. Consequently, one wants to be able to put a high level of confidence in this type of result. Thus, in the following, I will focus on certificates for *unreachability*.

In my work, certificates for unreachability are simply a set of symbolic states, i. e. pairs of a state of the automaton and a zone. Figure 5.1 shows an example for the model from Figure 2.1. We can see that compared to the model's zone graph from Figure 2.5, there are fewer symbolic states because all the *off* states have been replaced by a single one with the trivial unbounded zone (*true*). Solid edges represent regular edges of the zone graph, while dashed edges represent *subsumptions*: if there is an edge from (q, Z) to (q', Z') , then there exists a transition $q \xrightarrow{g,a,r} q'$ such that $Post(q \xrightarrow{g,a,r} q', Z) \subseteq Z'$. The graph in Figure 5.1 is a *reachability invariant*: for any symbolic successor of any node in the graph, the successor is either contained in the graph itself (solid edge), or it is subsumed by a larger node in the graph (dashed edge). A reachability invariant can thus be thought of as an inductive invariant of the zone graph. The key property of such a reachability invariant is that it can simulate any run of the underlying automaton. That is, if there is a run from a configuration (q, v) to (q', v') with $v \in Z$, and (q, Z) is part of the reachability invariant I (i. e. $(q, Z) \in I$), then there is also run from (q, Z) to some $(q', Z') \in I$ with $v' \in Z'$ (using subsumptions or edges of the zone graph).

This directly gives a way to certify unreachability. A certificate for unreachability is a set of symbolic states S such that:

- (1) S is a reachability invariant;
- (2) the initial configuration (l_0, u_0) is *covered* by S , i. e. there is a node $(l_0, Z_0) \in S$ with $u_0 \in Z_0$;
- (3) and no state in S fulfills the reachability predicate ϕ .

If there was a run in the automaton from (l_0, u_0) to (l, u) such that $\phi(l)$ holds, then there would also be a run from (l_0, Z_0) to some $(l, Z) \in S$ with $u \in Z$ (by the first two conditions and the simulation property of reachability invariants). This contradicts the third condition, and hence the certificate is a valid proof of unreachability. Note that the edges of the reachability invariant do not need to be a part of the certificate. The certifier has to check that they exist anyway by computing the symbolic successors of any (symbolic) state in S , and checking that each one of them is subsumed by some other (symbolic) state in S (which could also be the successor state itself).

The graph from Figure 5.1 is *not* quite a valid certificate of unreachability. As every state is reachable, condition (3) cannot be verified for a meaningful predicate ϕ . In contrast, Figure 5.2 presents a valid certificate for the modified light switch model from Figure 2.2. It proves

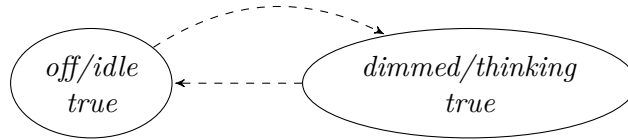


Figure 5.2: An unreachability certificate for the TA from Fig. 2.2.

that the lamp can never turn bright. The certificate needs only two symbolic states, while the full zone graph is even infinite, and the number of symbolic states explored by a model checker using extrapolation would also be much larger.

An important property of these certificates is that they are also complete. That is, for every model and property ϕ , if no state satisfying ϕ is reachable, then there is a valid finite certificate proving this fact. This is because from what was formalized in Chapter 3, we know that the zone graph with the classic extrapolation is finite and complete (every run can be simulated). It is also a reachability invariant: every successor state is subsumed by its own extrapolation. Hence the set of symbolic states explored by a TA model checker always forms a reachability invariant. If the exploration starts from the correct initial symbolic state, and the analysis deems no accepting state reachable, then the set of explored symbolic states is a correct certificate. In practice, one does not need to emit the full set of explored symbolic states as a certificate but it is sufficient to consider only the symbolic states that are stored at the end of exploration: they can subsume some symbolic states that were explored earlier on and then discarded later.

One salient feature of these certificates is that checking their validity only consists of a number of local checks that can be run independently for each state in the certificate, i. e. checking that the state is not accepting and that all its successors are subsumed by the certificate. Therefore the certificate checking process can be completely parallelized, which is not possible for model checking. Moreover, the certificate checker does not need to have any knowledge about the extrapolation that was used. As long as the extrapolation only enlarges zones, it will produce valid subsumptions, and that is all the certifier needs to check. Overall, the certificates I presented in this section provide a flexible and easily checkable way to prove unreachability in TA. To the best of my knowledge, I was the first to have ever studied this for TA¹.

5.2 Certificates for Büchi Emptiness

In the previous section, I have presented a method to certify unreachability in TA. This was presented in Paper E. In Paper F, we extended this work towards liveness properties. In contrast to reachability, which we can use to ensure that “something bad will not happen” (safety), the idea of liveness properties is to ensure that “something good will happen eventually” and variants thereof. In section 3.2, I have already discussed the most basic liveness property, deadlock freedom: the system will never get stuck. To give an example of a more complex liveness property, in a communication protocol, for instance, we could be interested in whether a message will *always* be sent *eventually*, even if it needs to be retried many times. In the example of the light switch, we could ask whether the light will *always* turn bright *eventually*. That is, in every (infinite) run of the automaton, no matter what happened before, will the light *always* turn bright again at some later point (*eventually*)? In the language of linear temporal logic (LTL) [164], the properties that follow this *always–eventually* structure can be

¹For related work on certifying other types of model checking, see Section 6.1.

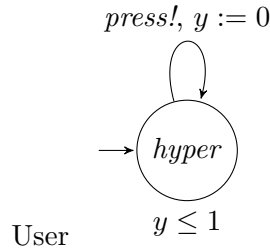


Figure 5.3: A modified version of the user model from Figure 2.1.

written as $\mathbf{GF} \phi$, for instance $\mathbf{GF} \textit{Light.bright}$. The properties are interpreted over all runs of an automaton, and \mathbf{G} stands for always (or globally), while \mathbf{F} stands for eventually (or future). The automata from Figures 2.1 and 2.2 both violate the formula $\mathbf{GF} \textit{Light.bright}$ because the user is not guaranteed to always be fast enough to turn on the light again at any given point in time. Figure 5.3 shows yet another modification of the user automaton. As the user now presses the switch at least every second, the light will always turn bright again and again, hence $\mathbf{GF} \textit{Light.bright}$ is satisfied. In general, LTL formulas can capture many interesting liveness properties [11].

It is a key result of research on model checking that any LTL formula can be converted into an equivalent *non-deterministic Büchi automaton (NBA)* [183]. An NBA is simply a nondeterministic finite automaton (a TA without clocks) where accepting states are interpreted as a Büchi condition. A run is Büchi if it visits an accepting state infinitely often. The typical approach of automata-based model checking of LTL formulas goes as follows [118]: the given LTL formula is first negated; the negated formula is then converted to an equivalent NBA; and finally the product with the transition system under question is constructed. The resulting product NBA then has a Büchi run if and only if the original transition system does not satisfy the formula. This approach can also be applied to TA. A TA where the accepting states are interpreted as Büchi condition is called a *timed Büchi automaton (TBA)*. At the core of the automata-based approach to model checking LTL properties of TA thus lies the problem of deciding whether the language of a given TBA is empty, i. e. whether the TBA does not have a Büchi run. Analogously to reachability, we are interested in certificates for the emptiness of TBA. Since if the constructed TBA is empty, it means the original TA fulfills the given LTL formula.

Figure 5.4 illustrates the idea for the automaton from Figure 5.3². The *Monitor* automaton is an NBA that monitors violations of $\mathbf{GF} \textit{Light.bright}$. A violation is a run in which the light turns bright finitely many times and then never turns bright again. To be able to monitor when the light turns bright, the automaton for the light was also modified. The state *bright'* is *urgent*: due to the constraint $t \leq 0$ no time can pass in the state, and the signal *turn-bright* is sent out immediately. The monitor can decide nondeterministically when it has seen the lamp turn bright for the last time, moving to the only accepting state *bad* (marked by a double circle). If the lamp would indeed never turn bright again, the monitor would stay in the *bad* state forever, constituting an accepting Büchi run of the product automaton. This signals a violation of the formula.

The emptiness of NBA can be decided by identifying so-called lassos, i. e. paths leading up to a cycle through an accepting state [118]. The NBA has a Büchi run if and only if it admits a lasso. For TBA, it can be shown that identifying lassos in the abstract zone graph also suffices to decide emptiness (when common extrapolations are used) [33, 133]. However, for efficiency, one also wants to exploit subsumptions when looking for lassos. The main

²To simplify the presentation, the constructions given here deviate slightly from typical approaches.

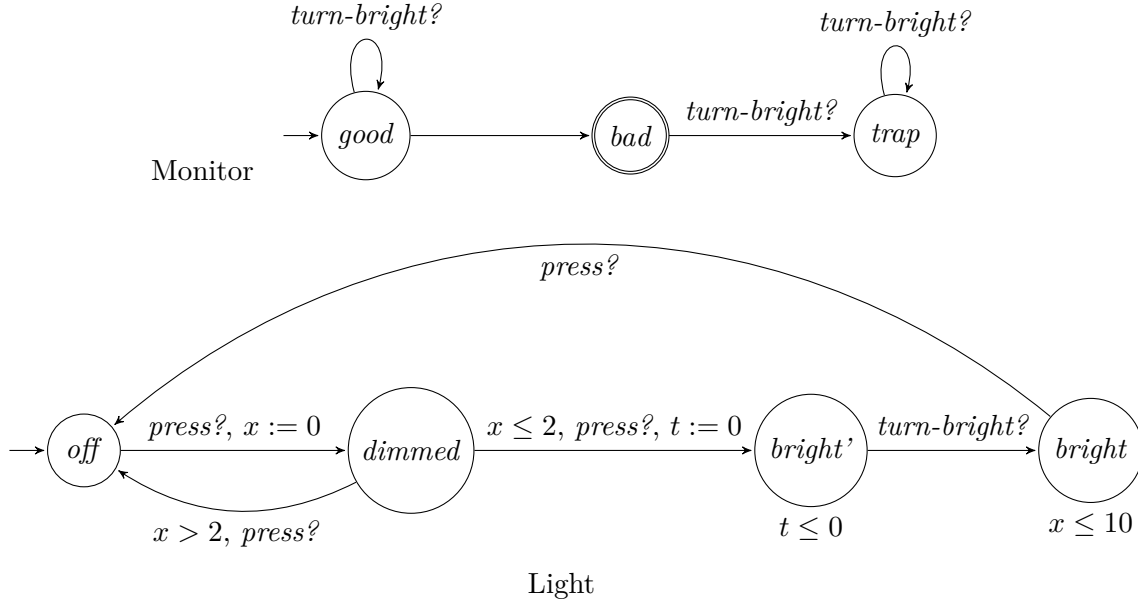


Figure 5.4: A modified version of the light model from Figure 2.1 with a monitor NBA.

algorithms for identifying lassos in finite directed graphs [170], nested depth-first search and decomposition into strongly connected components, can be adopted to exploit subsumptions [122, 94]. However, doing so correctly is much more intricate than for the model checking algorithms discussed so far. Moreover, it has been shown that deciding emptiness of TBA is inherently more difficult than deciding reachability for TA with zone-based methods [94]. For these reasons, it is not immediately clear whether one can expect to get a similarly simple way of certifying Büchi emptiness as we did for certifying unreachability in the last section. We were able to answer this question positively, however. The certificates for unreachability can be augmented such that all positive features are retained: certificate checking is still local and can easily be parallelized, the certificate checker is agnostic to the concrete extrapolation used, and the same type of certificate can be used for all state-of-the-art algorithms for checking emptiness of TBA.

The main idea of our certificates is to extend the certificates for unreachability with a special type of topological numbering. To this end, properties (2) and (3) of the certificates remain unchanged. For property (1), we also need to ensure consistency of the subsumptions with the proposed topological numbering. For any successor of any (symbolic) state in the certificate, we need to ensure that it is subsumed by another (symbolic) state in the certificate (the regular requirement for a reachability invariant), and in addition that the subsuming state's number is at most as high as the source state's number. Moreover, if the source state is accepting, the subsuming state's number must be strictly smaller. Figure 5.5 gives a certificate for the example from Figures 5.3 and 5.4. A topological numbering is given next to the nodes.

The key idea is that the topological numbering property essentially proves that there is no cycle through an accepting state in the certificate consisting of subsumptions and actual edges of the zone graph. From the properties of the reachability invariant, one can show that any Büchi run in the underlying TBA would yield such a cycle in the certificate. As there is no such cycle, the TBA has to be empty. Herbreteau et al. had already observed that not creating such cycles is the criterion for using subsumptions soundly when model checking TBA [94]. We studied these certificates in an abstract way for self-simulating transition systems, a slight relaxation of well-structured transition systems [68], which the zone graphs of TA are an

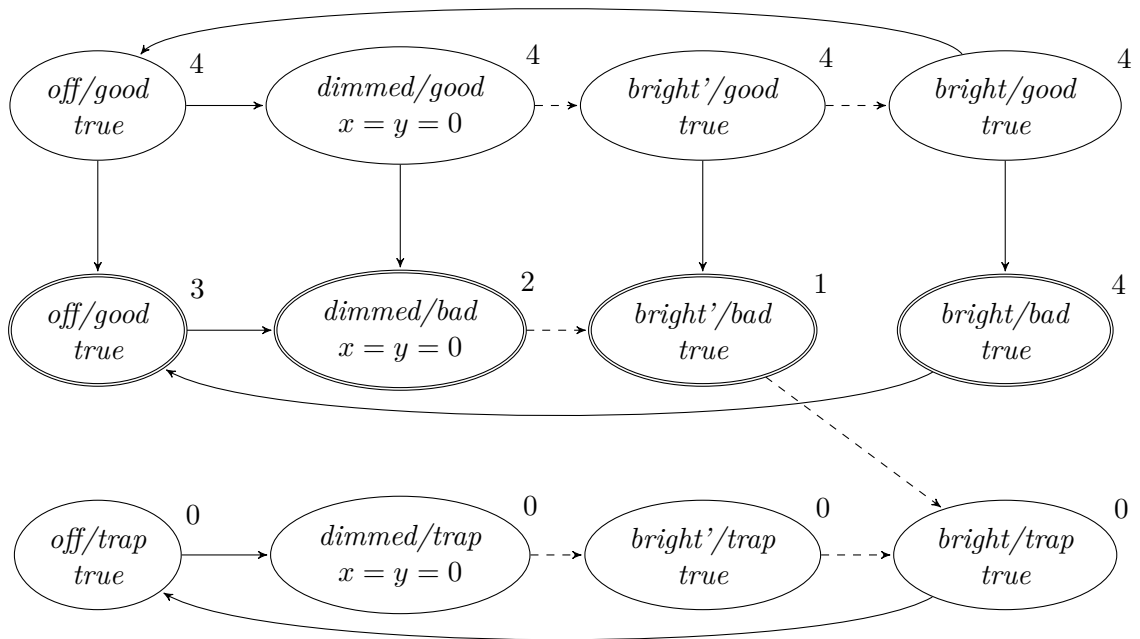


Figure 5.5: A certificate of emptiness for the TBA from Figures 5.3 and 5.4.

instance of. We (formally) proved soundness of the certificates, and also showed (informally) that they are complete for TBA, i.e. that for an empty TBA, a certificate can always be constructed.

5.3 Implicit Abstraction

So far, I have stressed that one of the valuable features of our certification approaches is that they are agnostic to the concrete extrapolation used. The second key result of Paper F is that, when giving up this agnosticism, valid certificates can even be given even when using abstractions that are not extrapolations. Recall that an abstraction α yields overapproximations of zones ($Z \subseteq \alpha(Z)$) and that its range is finite. An extrapolation is an abstraction that only yields zones from zones. I am only aware of one practical model checking approach that uses abstractions which are not extrapolations. The approach was developed by Herbreteau et al. [91, 95]. They do not apply abstractions to zones explicitly but rather apply them implicitly in subsumptions. That is, whenever they check whether a newly discovered symbolic state (l, Z) needs to be explored or can be discarded, they do not check whether a symbolic state (l, Z') with $Z \subseteq Z'$ has already been discovered, but instead they check whether $Z \subseteq \alpha_{LU}(Z')$. Here, α_{LU} is an abstraction that was already defined in the aforementioned paper on LU -extrapolation [14]. It is always at least as big as the LU -extrapolation, yielding a smaller symbolic search space (as illustrated in Figure 5.6). However, it does not necessarily yield a zone and therefore it is not an extrapolation that could be used in the regular way. The main result of Herbreteau et al. is that the subsumption check $Z \subseteq \alpha_{LU}(Z')$ can be computed as efficiently as a regular subsumption, meaning it can be practically used in model checking.

We showed that both certification approaches are also compatible with this technique of implicit abstraction. More specifically, we proved that for a certain class of abstractions α (which arise from time-abstract simulations and of which α_{LU} is an instance), the certificate checker just needs to replace regular subsumption $Z \subseteq Z'$ with the subsumption with implicit

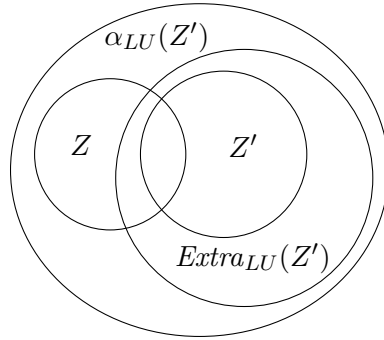


Figure 5.6: Illustration of implicit abstraction. The zone Z' and its LU -extrapolation, $Extra_{LU}(Z')$, are not large enough to subsume Z but the LU -abstraction of Z' , $\alpha_{LU}(Z')$, is.

abstraction $Z \subseteq \alpha(Z')$. Compared to the situation of the last sections, one also needs to be able to compute this operation, and to prove it correct to obtain a verified certifier.

5.4 Practical Verified Certificate Checking

In the last sections, I have summarized the theoretical insights of our work on certification of unreachability of TA and emptiness of TBA. In this section, I will report on how we turned these into a practical certifier that was tested against real model checkers. To obtain a verified certificate checker, the main effort was to formalize the aforementioned results in Isabelle/HOL. Starting from MUNTA, only little work had to be done to obtain an executable tool, as MUNTA already implements the whole pipeline of parsing a model, applying a product construction, and exploring the resulting automaton’s zone graph. After formalizing our (non-trivial) abstract results, it was almost trivial to adopt the certifier for unreachability to Büchi emptiness since only the topological numbering property needed to be checked in addition.

In Paper E we presented a number of optimizations for this verified certifier. The most notable is parallelization. As stated above, our certification algorithms are very well suited for this because only local properties on symbolic states need to be checked. To exploit this property without jeopardizing soundness, I developed techniques to soundly embed imperative code (in Imperative HOL) back into regular HOL (which can be thought of as a functional programming language). These rely on ideas similar to Haskell’s ST monad [130]. Then, parallelization is introduced by instructing the code generator to replace the map operator, which applies a function f to every element in a list, with a parallelized version. This is correct because any “pure” functional program is side-effect free. Other optimizations include using a machine word integer representation for DBMs (instead of standard arbitrary precision integers) and specifying optimized code equations for essential model checking operations.

We conducted a number of experiments with the verified certifier. In Paper E, we produced certificates of unreachability for a standard set of benchmark with our own unverified model checker, MLUNTA. The results showed that checking certificates (with the verified tool) alone is quite consistently an order of magnitude faster than performing full verified model checking with MUNTA. This is encouraging as it shows that the certification result can drastically reduce the computational burden of obtaining highly dependable verification results for TA. On the other hand, full model checking with the unverified tool UPPAAL often still outperforms verified certification significantly, sometimes (but rarely) even by up to an order of magnitude. The speedup achieved through parallelization is rather dissatisfactory. For two CPU cores, a doubling in speed is mostly nearly achieved but, when adding more cores, the effect quickly

diminishes. We did not find a convincing explanation for this. I suspect the reasons are to be found in the thread model of the Poly/ML³ compiler, which we use to compile the generated SML program.

In addition, we experimented with *certificate compression* for unreachability certificates. In our setting this means to reduce the number of symbolic states in the certificate after model checking. We tried different heuristics based on subsumption between zones and computing the convex hull of multiple zones. It is always ensured that the reachability invariant property of the certificate is maintained. The compression factor (the number of states after compression divided by the initial number of states) varies by model but one technique consistently achieved a compression factor of at least 40% for all the benchmark models. By chance, we have also achieved a compression factor of more than 95% for one model (instead of 43% for the otherwise best technique), showing that there is still a lot of potential for improvement on these techniques. In practice, certificate compression was not very useful, however, as it is computationally quite expensive with MLUNTA. Nevertheless, if this task was performed by a highly optimized tool like UPPAAL [16] or TCHECKER [92], it could significantly bring down the overall time of verification for the tandem consisting of an unverified model checker and the verified certificate checker.

In our second paper, we produced certificates for the emptiness of TBA with “real” model checkers, TCHECKER and IMITATOR [8]. In both cases, topological numbers can be computed by running an SCC decomposition on the graph which has as its nodes the symbolic states explored by the model checkers and as its edges the actual edges of the zone graph and any subsumptions that were used. In the paper, we discuss some further technical aspects for the construction of this pipeline of verification tools. The results show that the performance of checking certificates for the emptiness of TBA is similarly encouraging as in the case of unreachability (which is to be expected given the similarity of the checkers).

³<https://www.polyml.org/>

6 Conclusion

To conclude this thesis, I will summarize the most important lines of work related to mine. Next, I will point out some important limitations of my work, and subsequently I will lay out some ideas and perspectives for future work. Finally, I will muse on some general wisdom I draw from my work.

6.1 Related Work

As all of the work presented in this thesis is closely related, I give an overview of the most important literature that is related to my thesis as a whole here. Additionally, more specific references can be found in my papers. I exclude the two main bodies of work that lie at the foundation of my thesis, interactive theorem proving and model checking of timed automata. The interested reader shall instead be referred to a number of extensive survey papers and monographs on the respective topics ([190, 161, 150, 167] and [19, 11, 53]). Before summarizing the work that is more specifically related to mine, I want to point out that André Platzer has researched the trustworthy verification of CPS in a broad body of research, which has recently culminated in a monograph on the topic [163]. His approach is more general in that it can cover a large range of hybrid systems, not only timed automata. Similarly to my work, a high degree of trustworthiness is achieved by employing a special-purpose proof assistant for verification. The difference is that in his work, the system is verified directly in the proof assistant (in the usual interaction with the system). In contrast, my work focuses on increasing the trustworthiness of automated verification tools.

6.1.1 Formalization of Timed Automata and Markov Models

Various groups have formalized TA and the related formalism of p-automata [4] in Coq, PVS [156], and Isabelle/HOL, and have proved properties about concrete automata on this basis [159, 72, 47, 10, 201]. All these formalizations only provide little material on the theory of model checking TA. Advancing on this, Xu and Miao used PVS to formalize TA and the region construction [202] and made some attempts towards formalizing DBMs. Together with Han, they later formalized a simple predicate abstraction technique for TA [205] that can be used to prove properties of single concrete TA. None of these works go as far as proving the fundamental correctness results of DBM-based model checking, particularly using extrapolations.

The most extensive formalization of MDPs is by Johannes Hölzl in Isabelle/HOL [102], which built the basis for our formalization of PTA. Liu et al. formalize finite-state discrete-time Markov chains [135, 136] but not MDPs. I am not aware of any previous formalization of PTA.

6.1.2 Verified Model Checking

The most important forerunner of MUNTA is the CAVA project [65, 41]. It is a fully executable model checker for LTL properties of finite-state transition systems. It follows the automata-theoretic approach, which I have briefly described in Section 5.2, and implements partial-order

reduction. CAVA is a real forerunner of MUNTA in that the refinement techniques that we relied on for constructing MUNTA were in large parts developed in conjunction with CAVA. Apart from making use of these refinement techniques, MUNTA was developed independently of CAVA. CAVA outshines MUNTA in its support for the fully-fledged specification language Promela [146] and for the full set of LTL properties. MUNTA, on the other hand, supports the more powerful realtime (and infinite-state) formalism of TA, is in part implemented imperatively and thus more efficient, and provides more features for practical use such as a GUI. The only other verification of a model checker that I am aware of is by Sprenger for a modal μ -calculus checker in Coq [174]. The model checker does not seem to be focused on an efficient implementation or on supporting a practical modeling formalism, however.

In the course of the CAVA project, other parts of theory in the area of the automata-based model checking approach have been verified. Brunner has verified a tool for the complementation of finite Büchi automata [40], and Seidl et al. have verified a modular construction that can convert LTL formulae to equivalent Rabin and Büchi automata [42]. These tools are not model checkers per se but could be used as components of verified model checkers. I will outline a possible combination of the latter work with MUNTA below (in Section 6.3).

6.1.3 Algorithm Verification and Refinement

At the core of our constructions to make MUNTA an efficient and partly imperative verified model checker lie the refinement techniques developed by Peter Lammich for Isabelle/HOL. These started with a framework for stepwise refinement of nondeterministic programs (the Isabelle refinement framework) [128], followed by a tool for automatically refining programs to an efficient functional implementation (Autoref) [123], and were concluded with a tool for automatic refinement to Imperative HOL (Sepref) [126], which built the basis for our work. I have already mentioned (in Section 4.1) the work of Andreas Lochbihler on data refinement to efficient functional programs [137], and the paper where Lammich and Lochbihler compare their approaches [127].

The frameworks CoqEAL [54] and Fiat [60] provide similar functionalities to Autoref in Coq. Recently, Fiat was extended to a tool chain that also supports refinement to imperative programs similarly to Sepref [162]. While it has not been applied to complex algorithms yet, it has the nice property that it is complemented with a proof-producing compiler to an assembler language. Such a complete tool chain could also be achieved by connecting Lammich’s recent work on refinement to the LLVM intermediate language [125] with a verified compiler for this intermediate language.

In contrast to the top-down approach provided by refinement techniques, Charguéraud takes a bottom-up approach to the verification of imperative programs [51, 49]: an OCaml program is first translated to a so-called characteristic formula, which is then used as a basis for verification. Charguéraud’s work is similar to the works of Lammich and Lochbihler in that he verifies similar algorithms and data structures. To verify the imperative hash table implementation which lies at the core of MUNTA’s state space exploration algorithms, we borrowed some core ideas from Charguéraud’s work on higher-order representation predicates for separation logic [50].

Many other techniques for verification of algorithms and programs with the help of proof assistants exist. As they are only remotely connected to the work presented in this thesis, I refer the interested reader to a recent survey by Nipkow et al. [149]. It gives an overview of different approaches towards verifying algorithms (but not necessarily efficient implementations) in the ITP world and an extensive list of such verified algorithms.

6.1.4 Certification in Verification

The idea of programs that produce independently checkable certificates spans a large range of areas of computer science: from the definition of the famous complexity class NP, over tactics for proof assistants, to self-explaining artificial intelligence. Therefore, I will only focus on research where certification was applied within the verification community.

Certificate checking for *SAT*, the problem of satisfiability of Boolean formulas, arguably represents the most important stream of work in this area. On the one hand, producing and checking certificates for satisfiable Boolean formulas is nearly trivial. On the other hand, certifying the *unsatisfiability* of a formula is much harder and many approaches have been proposed [189, 188, 173]. The SAT competition¹, which is an important driver for research in the area, has been demanding that tools output certificates since 2013. The verification of (UN)SAT certificate checkers with proof assistants has also been studied [56, 57, 124] and convincing results have been obtained [124]: for the same formula, verified certificate checking for unsatisfiability can be (much) faster than unverified SAT solving.

The IsaFoR/CeTA project [175] embodies many applications of the certification approach to tools for automatic termination checking and complexity analysis of programs. It is fully verified in Isabelle/HOL. Besson et al. constructed an analyser for byte code programs, which can be checked by a verified checker [25]. Noschinski et al. verified certificate checkers for graph algorithms [154].

Proof tactics in proof assistants are also often based on the principle of certification [167]. Given a proof goal, they might attempt to run an unverified decision procedure or calculation first. In the case of a success, the unverified code produces a certificate that is then “replayed” by composing elementary proof rules (or tactics). A notable instance of this are tactics for replaying proofs that are produced by solvers for the satisfiability modulo theories (SMT) problem and other automated theorem provers. These tactics give rise to tools like Sledgehammer [31, 30], HOL(y)Hammer [110] and CoqHammer [58], which allow the user of an interactive theorem prover (ITP) to hand off a current proof goal to a range of external ATPs (usually involving a lossy translation). If an external ATP succeeds at proving the goal, it is asked for a certificate, which is then replayed in the ITP. Similarly, Li et al. replayed certificates that were produced by a computer algebra system to decide univariate polynomial problems in Isabelle/HOL [134]. In a different approach, the primary purpose of the tactic is to perform a computation and the certificate is a proof that is constructed by the tactic during the calculation. Examples of this can be found in a large part of the aforementioned work on refinement, in proof-producing compilation (in the bootstrapping process of the CakeML compiler [177] or the Cogent compiler [5]), and Manuel Eberl’s tactic for automatically proving limits of real functions in Isabelle/HOL [63].

In the context of software verification, the idea of producing certificates for the correctness of a program has been broadly studied [106, 26] but none of the checkers have been verified. In hardware model checking, approaches for certifying the results of BDD-based and IC3-based model checkers with the help of SAT solvers have been developed [116, 207]. Again, no checkers have been verified but a connection to the existing verified checkers for UNSAT certificates is conceivable [99, 124]. The work of Griggio et al. is also noteworthy [80]. They provide a proof calculus for LTL formulas on finite transition systems, and a procedure for extracting proofs in this calculus from a typical model checker based on the IC3 approach. They also provide a prototype implementation of a certifier for such proofs, which delivers reasonable performance. This certifier is not verified however, and their technique is not applicable to more advanced formalisms like TA. Older work on certification has focused on the theory of

¹<http://satcompetition.org/>

the μ -calculus [144, 101]. It is of more theoretical nature and has not been applied to concrete verification tools. To my knowledge, no one has worked on a verified checker for certificates produced by a model checker or on producing certificates from TA model checking before.

6.2 Limitations

Before I remark on different potential directions for future research, I want to discuss some limitations of the work presented in this thesis. Roughly, there are two types. On the one hand, these are limitations of the usefulness of the verification methods covered by my work. On the other hand, the degree of trustworthiness of these methods has certain limits.

6.2.1 Simplifying Assumptions

On the model checking side, one important limitation are the simplifying assumptions I made about TA throughout. These were that TA are diagonal-free and strongly non-Zeno.

Diagonal Constraints As discussed above (Section 2.2), any TA can always be converted into an equivalent diagonal-free automaton but for the price of a potentially exponential blowup (in the number of diagonal constraints) [20]. Nevertheless, the restriction is very often made as most practical models fall into the class of diagonal-free TA. Moreover, most research on TA model checking has also focused on this class. Implementing and verifying the transformation to diagonal-free TA in MUNTA would be a very conceivable feat. Apart from this, there are currently two relevant approaches to handle diagonal constraints directly during model checking.

The older approach implemented in UPPAAL [19] amounts to splitting zones according to diagonal constraints, into a zone implying the constraint and another implying its negation. This is done for every diagonal constraint of a given TA and any zone stored during extrapolation. Therefore, this approach essentially moves the exponential blowup from the automaton to the zone splitting process. Moreover, it is only compatible with older, less coarse extrapolations and does not allow one to use local clock ceilings. I do not see a theoretical barrier for adding this technique to both, MUNTA’s capability as a model checker and the certification methods we studied for reachability. For emptiness of TBA, to my knowledge, this method has never been considered, and I have not investigated whether it can be applied soundly to the problem.

A recently developed technique by Gastin et al. [73] builds on the ideas of Herbreteau et al. [95] on using abstractions implicitly, which I discussed in Section 5.3. The approach appears to be significantly superior over the older approach in practice. As we have already studied certification techniques based on implicit abstractions in Paper F, an extension to this method should not provide many obstacles *if* the correctness of the specific subsumption check with implicit abstraction and its implementation has been proved. In fact, this should be rather easy once the same task has been finished for the work of Herbreteau et al. [95]. However, given the complexity of the proofs on paper, doing so seems quite laborious. An extension of MUNTA’s model checking capabilities with implicit abstractions also seems quite feasible after verifying correctness of the subsumption check. As before, the work by Gastin et al. does not consider emptiness of TBA, and I am not aware whether it is directly applicable to the problem.

Zenoness For reachability problems, only finite runs are considered and thus Zeno runs cannot occur. A given TBA can be transformed such that all accepting runs are non-Zeno

while keeping all other behaviors intact [180]. This transformation incurs an exponential blowup in the number of clocks [96]. Similarly to the case of TA with diagonal constraints, implementing and verifying the construction in MUNTA would be a goal within close reach.

Herbretreau et al. [96] have developed an extension of the regular zone graph construction (with abstractions) for deciding the emptiness of TBA when only non-Zeno runs are regarded as accepting. Their method is efficient in that the number of reachable zones is only increased linearly in the number of clocks in the worst-case, and their proposed algorithm can often avoid the blowup altogether. However, their algorithm does not make use of subsumptions, which is usually considered of high importance for the efficiency of TA model checking algorithms². Nevertheless, extending MUNTA with this method would not pose any fundamental problems but would possibly be burdensome. Similarly, on an abstract level, producing and checking certificates for this technique would be rather straightforward but incur some (but less) work on the implementation level.

Like deadlocks, Zeno TA could be assumed to be a modeling error. Unlike for deadlocks however, MUNTA does not yet provide a way to identify such automata. The question of detecting whether a TA has a Zeno run has been studied more intensively than excluding such runs directly from model checking. The approaches are similarly intricate and a silver bullet does not seem to exist at present. One line of work studied different sufficient static conditions of TA for strong non-Zenoness that can be checked efficiently [180, 74]. All of these are certainly simple enough to be verified in Isabelle/HOL but would make little use of the work developed in this thesis.

In another approach, a given network of TA is first extended with an additional clock [180, 74, 133]. On this automaton, one can then prove non-existence of Zeno runs by checking that a certain “leads to” property holds for the network. As MUNTA already supports this type of property, implementing the construction on top of MUNTA would be rather easy. Again, the (main) downside of this approach would be the exponential worst-case blowup incurred by adding the additional clock [93].

Herbretreau and Srivathsan studied methods to avoid adding this extra clock [93]. Their surprising finding was that the typical *LU*-extrapolation is no longer efficient in this setting. As a remedy, they provide slight modifications of the extrapolation together with algorithms based on the previous extended zone graph construction for excluding Zeno runs [96]. Again, their algorithms do not make use of subsumptions but could be added to MUNTA. The effort would be similar to the aforementioned techniques that are also based on the extended zone graph construction (and synergies could be exploited if both methods were to be added). I consider it an open question how certification of this method could be fully achieved.

Finally, it should also be noted that even the popular model checker UPPAAL does not provide methods that can take Zeno runs into account. However, the techniques based on adding an additional clock can at least be used rather easily after manual intervention.

6.2.2 Modeling Formalism

Another limitation, which I already discussed in Section 4.4, is the expressiveness of the modeling formalisms. To summarize, for now I have opted for a simple but interesting enough formalism that is similar to what is provided by many other model checkers. We have shown (in Paper C and Paper D) that it is possible to support a richer formalism like the one of UPPAAL in principle. Doing so is laborious though, due to the complexity of the product

²Note that the algorithms we targeted with certification in Paper F [122, 94] can exploit subsumptions but were developed only for strongly non-Zeno TBA.

construction and semantic oddities that arise when using free-form programs as annotations on states and transitions.

6.2.3 Temporal Properties

On the semantic side, the other big limitation are the types of properties that are supported by the methods I studied. Compared to UPPAAL, MUNTA supports a very similar set of properties but formulas cannot refer to clock values, i. e. they are from the temporal logic CTL instead of TCTL. It would certainly be possible to lift this limitation, although it would present a larger refactoring effort for MUNTA. The reason is that zones would now need to be split accordingly to the clock constraints occurring in the formula. The model checker Kronos [206] supports the full logic TCTL. Extending MUNTA with the automata-theoretic construction of Bouajjani et al. [33] to support full TCTL should be feasible when employing the methods developed in this thesis (and adding the aforementioned zone splitting).

6.2.4 Efficiency

The applicability of my work could also be inhibited by the efficiency and scalability of the developed algorithms and tools. For verified model checking, compared to UPPAAL, MUNTA can only get within an order of magnitude in throughput, the number of states checked per second (cf. Paper C). The overall model checking time can even compare less favorably when UPPAAL is allowed to use its best exploration methods, as it can then explore significantly fewer states than MUNTA, depending on the model. I thus conjecture that MUNTA is sufficiently efficient to be applied to medium-sized models but will be prohibitively slow for large models. This can still be seen as a good stepping stone for a verified model checker. Moreover, MUNTA can also quickly verify the realistic Penn pacemaker models [108].

With certification, we have pushed the performance boundaries much further. First, the performance of extrapolations and other possible search-space reduction techniques such as the choice of search order [97] is not a concern anymore. Second, throughput was greatly improved by certificate checking, and the process can even be parallelized (while the positive effect of this quickly diminishes beyond two cores). Thus I conjecture that the certification methods can also scale well to the largest models that can be handled by state-of-the-art model checkers.

6.2.5 Trustworthiness

To conclude this discussion, I want to turn to the issue of trustworthiness. This is highly important as it was the main goal I initially set out to achieve: to make verification of realtime systems as dependable as possible. As the verification is carried out by a computer, there are certain factors that could lead to a wrong result and that belong to the “physical” world. These are errors of the computer hardware or user errors such as misreading a verification result. The most reasonable way to rule out this kind of error seems to me to simply to repeat the verification in different settings.

All other sources of errors are confined to the “software stack”. An important notion pertaining to this is the *trusted code base (TCB)* of a verification tool. Recall that this comprises all code that has not been verified but which is simply assumed to be correct and thus “trusted”. For MUNTA, this consists of³:

- (a) parts of MUNTA that are trusted code,

³For a diagram of MUNTA’s architecture see Paper D.

- (b) my formalization of the modeling language semantics,
- (c) the formalization of Imperative HOL and its corresponding separation logic,
- (d) Isabelle/HOL’s logical kernel,
- (e) Isabelle/HOL’s code generator,
- (f) the target language’s compiler and runtime system,
- (g) and the underlying operating system.

As I argued in the introduction, it is reasonable to trust the correctness of (d), which is also regularly assumed within the research community. The trustworthiness of (a) is discussed in my tool paper about MUNTA (Paper D). Regarding (b) and (c), I believe that trust in these components must always be gained via manual inspection. On the positive side, these TCBs are quite small, and Imperative HOL has been applied in many projects, which can help to increase the level of confidence in it. The semantics of the modeling language are only around 150 lines of Isabelle/HOL text, and I claim that they compare favorably in complexity even to a handwritten description as given in the UPPAAL manual, for instance.

For the last three items, a large body of existing and ongoing work within the verification community offers a potential remedy. Hupel and Nipkow [104] have started to tackle (e) by generating code from Isabelle/HOL to CakeML [177] in a provably correct way (in the sense of mechanically checked proof). In turn, CakeML is a dialect of ML that comes with a verified compiler and runtime system, addressing the potential soundness issues of (f). There are also other projects that provide a similar verified tool chain from a program specified in a proof assistant to a compiled binary. The first realistic fully verified compiler was CompCert [132, 24]. Verified code extraction to CakeML was first realized for the HOL system [143]. I already mentioned the improvements in refinement technology by Claudel et al. [162] and Lammich [125] that allow one to directly synthesize a verified program in a low level language, the Bedrock assembler language and the LLVM intermediate language, respectively. Such approaches can be extended with a verified runtime system (f) and a verified operating system (g) to complete the verified tool chain. Two major projects have addressed this goal: the work that evolved around the verified seL4 operating system kernel [112] and the DeepSpec project [9] (and in particular the CertiKOS project [81]). There is thus a reasonable justification to dream of connecting MUNTA to a fully verified tool chain that reduces the TCB of (e) to (g) to (nearly) zero.

6.3 Future Work and Perspectives

The discussion from the last sections directly sparks a number of interesting ideas for future work. First, it would be desirable to extend MUNTA with any of the solutions to lift its limitations that I discussed in the last section, of course. For the modeling language, an interesting extension of MUNTA would be the JANI format [44]. It aims to provide a simple standard format for TA and similar formalisms, and would thus be a fitting target for a reference model checker and certifier like MUNTA, in the hope that other model checkers will also adopt this format in the future. Related to that would be a verified parser for the format, which is based on JSON. As many verified parsers for JSON have already been developed (e.g. by Lasser et al. [129]), one will hopefully become available in Isabelle/HOL in the not too distant future as well.

An obvious extension of MUNTA in its role as a verified model checker would be support for better abstraction methods, as was already discussed in previous chapters (such as the

LU-extrapolation described in Section 3.3 and the implicit abstraction techniques discussed in Section 5.3). As a certifier, MUNTA’s value would probably be most increased by making it applicable to the whole process of LTL model checking. As discussed in Section 5.2, certifying emptiness of TBA is only a part of the automata-theoretic approach towards LTL model checking of TA. Fortunately, the relevant constructions have already been verified to a large extent in the CAVA project [168, 65, 42]. The main open question is how to best combine the verified conversion from an LTL formula to an NBA with an unverified model checker. There are two possible routes. One option is to also apply the verified construction on the unverified side before model checking continues. MUNTA could then check the given certificate for Büchi emptiness against the automaton it would construct from the LTL formula independently (with the same construction). Another option would be to extend the work of Seidl et al. [42] such that the unverified tool can provide yet another certificate for the correctness of the formula automaton.

There are a number of potential interesting applications for MUNTA as a validation tool. First, it could be employed in a model checking competition for TA. In other areas of formal methods, such as SAT solving or software verification, competitions have been an important driver for innovation and research. In both types of competitions, tools nowadays need to produce certificates to demonstrate correctness of their results. Such a competition does not exist yet for TA model checkers but in my opinion it would be desirable to initiate such an event, and MUNTA would complement it perfectly: on the one hand, correct functioning of the competing (semi-symbolic) model checkers could be ensured by its certification ability; on the other hand, it would put MUNTA to a real-world test, both as a model checker and a certifier. Second, MUNTA could be used to find bugs in other model checkers. There are multiple possible “vectors of attack” here. In one approach, models and formulas could be generated by some automated method. These could then be given to MUNTA and an unverified tool, and the outputs could be checked for discrepancies (or where possible, certificates could be checked for validity). A discrepancy would constitute a critical bug in the unverified tool (or at the least a bug in the certification pipeline). In another approach, the DBM library of other model checkers could be scrutinized, for instance by means of black-box or white-box testing.

Just as DBMs are not only used for TA model checking, MUNTA’s DBM library also has other potential applications. A particularly interesting instance would be in abstract interpretation. Abstract interpretation is a widely spread automated program analysis technique, usually computing an over-approximation of potentially reachable program states. To represent reachable states symbolically, a *domain* of abstract values needs to be chosen. The so-called octagonal domain [140, 141] is popular with tools that implement the technique and is based on DBMs as a data structure. Nipkow has already verified a toy abstract interpreter in Isabelle/HOL [147]. However, it lacks an interesting practical domain of abstract values, in particular it is not able to track differences between variables. It would thus be attractive (and quite feasible) to combine this with the verified DBM library to obtain a more realistic verified abstract interpreter.

Other visions are more ambitious. For one, I have only targeted semi-symbolic methods for TA model checking. While these are by far the most popular and the only ones that are practically employed, there is a large range of other techniques. The most interesting ones are fully symbolic. As mentioned in Section 2.6, they can be divided into two kinds, methods based on BDD-like data structures and methods based on SMT solvers. Both techniques would present an interesting target for certification. This seems particularly appealing for the methods based on SMT solvers since methods for replaying SMT proofs are readily available in Isabelle/HOL.

Beyond TA, there is of course a wealth of other modeling formalisms that could be targeted. This is particularly appealing for all well-structured transition systems [68] such as Petri nets because our certification methods are directly applicable to them (see Paper F). For more general hybrid systems, the certification approach again seems to be most apt as most model checking problems are undecidable there. Thus, for instance, certifying an unreachability invariant after model checking seems to be the most attractive option.

Finally, one can dream more broadly of how ITPs could be used to build realtime systems with the highest degree of trustworthiness in the future. Let me allude to the “chain of trust” metaphor from the introduction again. For the task of constructing a concrete realtime system, I have prolonged the chain on one end, where the ITP holds everything together instead of a model checker. I have already alluded to many options of proofing the thick links at this end of the chain even further. On the other end of the chain sits the model. The model is of course not yet the real system and thus there is ample opportunity to extend the chain of trust on that side. The most immediate target there would be to generate actual code from the model in a verified way. Such code needs to run on a dedicated realtime platform which provides guarantees on the worst-case execution time of instructions. Different groups have investigated code synthesis from TA [37, 200, 115, 176, 108]. However, to not cut the chain of trust in the middle, such an approach should be verified with an ITP. An independent research project is currently investigating exactly that⁴. Two other speculative and to me completely open questions are whether ITPs can help rule out modeling errors at the interface between user and specification, and whether a combination of interactive proof and verified model checking within an ITP could target models whose complexity currently lies beyond the capabilities of existing push-button technologies. As an example, I imagine that the ITP could be used to soundly and modularly combine properties of sub-components of a model, which in turn could either be established by further decomposition or model checking. Some light in that direction was already shed by previous work on compositionality and abstraction of TA [107, 155].

6.4 General Reflections

I believe that the research question I posed in the introduction has been addressed in a largely satisfactory way. With MUNTA, I have extended the chain of trust from model checkers to Isabelle/HOL in two ways: MUNTA can stand in for the model checker itself and thus link the chain to Isabelle/HOL; or in its role as a certifier it can serve as a link between an unverified model checker and Isabelle/HOL. If my only goal had been to extend this chain, then the latter functionality would seem quite clearly superior for the reasons given in the last chapter. However, having a fully verified model checker that can stand on its own also delivers certain benefits, for instance as a trustworthy tool for experimentation that can be run right in the browser like MUNTA. Moreover, one can argue that formalizing the correctness of the typical extrapolation operation used in semi-symbolic TA model checking is an important scientific undertaking by itself. This seems to be particularly true when considering all the uncertainty and difficulties that have been related to it historically and the complexity of the correctness argument in itself. In that light, developing a full verified model checker was also a natural step after I had finished the formalization of all the underlying concepts.

My work can also be seen as another piece of evidence in a big sea of large verification efforts that have been carried out with ITPs up to date (some of the most important were listed in the introduction). They all show that ITPs today are capable of verifying rather

⁴The project is titled “High-Confidence Formal Verification of Real Cyber-Physical Systems: from Models to Machine Code”. More information can be found online: <https://people.kth.se/~dbro/projects.html>.

complex pieces of software, and therefore one can hope for a further uptake of the technology in practice. I hope that the work presented in this thesis could serve as a stepping stone in that direction.

While most of my work could in principle also have been carried out with other ITPs, it highlights the particular maturity of Isabelle/HOL as a tool. Realistically speaking, there would have been only one significant contender to construct an efficient verified tool (particularly one that can implement DBMs imperatively) as opposed to the pure formalization work presented in Chapter 3: Coq. As discussed in Section 6.1, the necessary technology has only become available in Coq very recently. However, I claim that even if I were to carry out this work anew today, Isabelle/HOL would still be the most effective tool by some margin. I have relied on many of its distinctive features during my work: its good support for coinduction and coinductive datatypes [28], its ability to interface with automated provers and SMT solvers [29], its structured proof language [187], its mature code generation facilities [83], and Peter Lammich’s work on imperative refinement [126].

Having said that, the road to MUNTA still has been bumpy. While ITP technology and refinement technology are reaching ever better levels of maturity, it was still quite laborious to get from a formalized theory to a verified model checker—a step that might seem simple at first thought. I have already given reasons why the story is not so simple in Chapter 3 but some of it is certainly related to difficulties that are still posed by the aforementioned technologies. The refinement framework in particular can seem like a mysterious conundrum at first. Sometimes even the advanced apprentice sorcerer will be bamboozled by it again, and it will only submit after the headmaster has exacted on it a sermon of incantations. Similar beasts can also hide behind other trees in the magical forest that is Isabelle/HOL. Some of these I encountered when verifying product constructions; for instance when grappling with locally defined constants that were unfolded to their logical foundation for no apparent reason. This renders most of Isabelle’s automation useless and causes the current proof state to be unreadable. It took me years of stumbling through the forest until the right potion was given to me by seasoned masters (using congruence rules to prohibit this behavior in Isabelle’s simplifier).

Nevertheless, sometimes the forest can also reveal its magic to the great astonishment of the apprentice sorcerer. This happened when I extended the TA formalization to local clock ceilings with seemingly little effort (cf. Section 3.3). What is more, sometimes proof assistants can also aid in proof discovery as in the following example. The seminal paper that introduced *LU*-extrapolation gave a correctness argument on the basis of time-abstract simulations [14], and showed that the *LU*-simulation (defined in the same paper) is time-abstract. Before formalizing this result, Frédéric Herbreteau made me aware that Paul Gastin had suggested that *LU*-simulation might even be a regular simulation. It turned out that this result is quite trivial to prove in Isabelle/HOL, and proving that it is a proper simulation is even more natural than proving that it is time-abstract.

Thus, if I was to dream, I would hope that Isabelle/HOL and ITP technology in general will mature further and ease the education of future apprentice sorcerers. I believe that there would then be a strong case to involve ITPs in regular development and research of verification theory and tools. This is getting more and more widespread in the community, but like in my own work, only happens after fact: once theory and tools have been developed and scrutinized in a more informal setting, one tries to verify these. While this may often be reasonable, I conjecture that in the future, verification within a proof assistant could also interact beneficially with the research process itself.

Bibliography

- [1] Ben Algaze. *Software is Increasingly Complex. That Can Be Dangerous*. 2017. URL: <https://www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous> (visited on 05/28/2020).
- [2] Rajeev Alur and David L. Dill. “A theory of timed automata.” In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. ISSN: 03043975. DOI: 10.1016/0304-3975(94)90010-8.
- [3] Rajeev Alur and David L. Dill. “Automata For Modeling Real-Time Systems.” In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, 1990, Proceedings*. Ed. by Mike Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 322–335. DOI: 10.1007/BFb0032042.
- [4] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. “Parametric real-time reasoning.” In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*. Ed. by S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal. ACM, 1993, pp. 592–601. DOI: 10.1145/167088.167242.
- [5] Sidney Amani et al. “CoGENT: Verifying High-Assurance File System Implementations.” In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’16*. Ed. by Tom Conte and Yuanyuan Zhou. ACM, 2016, pp. 175–188. DOI: 10.1145/2872362.2872404.
- [6] Nada Amin and Tiark Rumpf. “LMS-Verify: abstraction without regret for verified systems programming.” In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 859–873. URL: <http://dl.acm.org/citation.cfm?id=3009867>.
- [7] Tobias Amnell et al. “Code Synthesis for Timed Automata.” In: *Nord. J. Comput.* 9.4 (2002), pp. 269–300.
- [8] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. “IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems.” In: *FM 2012: Formal Methods - 18th International Symposium, Proceedings*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. Lecture Notes in Computer Science. Springer, 2012, pp. 33–36. DOI: 10.1007/978-3-642-32759-9_6.
- [9] Andrew W. Appel et al. “Position paper: the science of deep specification.” In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* (2017). ISSN: 1364503X. DOI: 10.1098/rsta.2016.0331.
- [10] Myla Archer and Constance Heitmeyer. “TAME: A Specialized Specification and Verification System for Timed Automata.” In: “*Work In Progress session*” at the *17th IEEE Real-Time Systems Symposium*. 1996.
- [11] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

Bibliography

- [12] Aditi Barthwal and Michael Norrish. “Verified, Executable Parsing.” In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 160–174. DOI: 10.1007/978-3-642-00590-9_12.
- [13] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim Guldstrand Larsen. “Static Guard Analysis in Timed Automata Verification.” In: *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Proceedings*. Ed. by Hubert Garavel and John Hatcliff. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 254–277. DOI: 10.1007/3-540-36577-X_18.
- [14] Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. “Lower and upper bounds in zone-based abstractions of timed automata.” In: *International Journal on Software Tools for Technology Transfer* (2006). ISSN: 14332779. DOI: 10.1007/s10009-005-0190-0.
- [15] Gerd Behrmann, Patricia Bouyer, Kim Guldstrand Larsen, and Radek Pelánek. “Lower and Upper Bounds in Zone Based Abstractions of Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 312–326. DOI: 10.1007/978-3-540-24730-2_25.
- [16] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. “A Tutorial on Uppaal.” In: *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Revised Lectures*. Ed. by Marco Bernardo and Flavio Corradini. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 200–236. DOI: 10.1007/978-3-540-30080-9_7.
- [17] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. “Beyond Liveness: Efficient Parameter Synthesis for Time Bounded Liveness.” In: *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Proceedings*. Ed. by Paul Pettersson and Wang Yi. Vol. 3829. Lecture Notes in Computer Science. Springer, 2005, pp. 81–94. DOI: 10.1007/11603009_7.
- [18] Gerd Behrmann et al. “UPPAAL Implementation Secrets.” In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Proceedings*. Ed. by Werner Damm and Ernst-Rüdiger Olderog. Vol. 2469. Lecture Notes in Computer Science. Springer, 2002, pp. 3–22. DOI: 10.1007/3-540-45739-9_1.
- [19] Johan Bengtsson and Wang Yi. “Timed Automata: Semantics, Algorithms and Tools.” In: *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer, 2003, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.
- [20] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. “Characterization of the Expressive Power of Silent Transitions in Timed Automata.” In: *Fundamenta Informaticae* 36.2-3 (1998), pp. 145–182. DOI: 10.3233/FI-1998-36233.

- [21] Ulrich Berger, Helmut Schwichtenberg, and Monika Seisenberger. “The Warshall Algorithm and Dickson’s Lemma: Two Examples of Realistic Program Extraction.” In: *J. Autom. Reasoning* 26.2 (2001), pp. 205–221. DOI: 10.1023/A:1026748613865.
- [22] Stefan Berghofer. “Program Extraction in Simply-Typed Higher Order Logic.” In: *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Selected Papers*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer, 2002, pp. 21–38. DOI: 10.1007/3-540-39185-1_2.
- [23] V Bertin et al. “TAXYS=Esterel+Kronos. A tool for verifying real-time properties of embedded systems.” In: *Proceedings of the 40th IEEE Conference on Decision and Control*. Vol. 3. IEEE, 2001, pp. 2875–2880. ISBN: 0-7803-7061-9. DOI: 10.1109/.2001.980712.
- [24] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. “CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics.” In: *J. Autom. Reason.* 63.2 (2019), pp. 369–392. DOI: 10.1007/s10817-018-9496-y.
- [25] Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. “Certified Result Checking for Polyhedral Analysis of Bytecode Programs.” In: *Trustworthy Global Computing - 5th International Symposium, TGC 2010, Revised Selected Papers*. Ed. by Martin Wirsing, Martin Hofmann, and Axel Rauschmayer. Vol. 6084. Lecture Notes in Computer Science. Springer, 2010, pp. 253–267. DOI: 10.1007/978-3-642-15640-3_17.
- [26] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. “Correctness Witnesses: Exchanging Verification Results between Verifiers.” In: *FSE 2016*. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 326–337. ISBN: 9781450342186. DOI: 10.1145/2950290.2950351.
- [27] Dirk Beyer, Claus Lewerentz, and Andreas Noack. “Rabbit: A Tool for BDD-Based Verification of Real-Time Systems.” In: *Computer Aided Verification, 15th International Conference, CAV 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 122–125. DOI: 10.1007/978-3-540-45069-6_13.
- [28] Julian Biendarra et al. “Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic.” In: *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Proceedings*. Ed. by Clare Dixon and Marcelo Finger. Vol. 10483. Lecture Notes in Computer Science. Springer, 2017, pp. 3–21. DOI: 10.1007/978-3-319-66167-4_1.
- [29] Jasmin Christian Blanchette. “Automatic proofs and refutations for higher-order logic.” PhD thesis. Technical University Munich, 2012. URL: <https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20120628-1097834-1-6>.
- [30] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. “Extending Sledgehammer with SMT Solvers.” In: *J. Autom. Reason.* 51.1 (2013), pp. 109–128. DOI: 10.1007/s10817-013-9278-5.
- [31] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. “Automatic Proof and Disproof in Isabelle/HOL.” In: *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, 2011. Proceedings*. Ed. by Cesare Tinelli and Viorica Sofronie-Stokkermans. Vol. 6989. Lecture Notes in Computer Science. Springer, 2011, pp. 12–27. DOI: 10.1007/978-3-642-24364-6_2.

Bibliography

- [32] Clement Blaudeau and Natarajan Shankar. “A verified packrat parser interpreter for parsing expression grammars.” In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*. Ed. by Jasmin Blanchette and Catalin Hritcu. ACM, 2020, pp. 3–17. DOI: 10.1145/3372885.3373836.
- [33] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. “On-the-fly symbolic model checking for real-time systems.” In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*. IEEE Computer Society, 1997, pp. 25–34. DOI: 10.1109/REAL.1997.641266.
- [34] Patricia Bouyer. “Forward Analysis of Updatable Timed Automata.” In: *Formal Methods Syst. Des.* 24.3 (2004), pp. 281–320. DOI: 10.1023/B:FORM.0000026093.21513.31.
- [35] Patricia Bouyer. “Untameable Timed Automata!” In: *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*. Ed. by Helmut Alt and Michel Habib. Vol. 2607. Lecture Notes in Computer Science. Springer, 2003, pp. 620–631. DOI: 10.1007/3-540-36494-3_54.
- [36] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. “Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems.” In: *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Proceedings*. Ed. by Paul Pettersson and Wang Yi. Vol. 3829. Lecture Notes in Computer Science. Springer, 2005, pp. 112–126. DOI: 10.1007/11603009_10.
- [37] Patricia Bouyer, Nicolas Markey, and Ocan Sankur. “Robust Model-Checking of Timed Automata via Pumping in Channel Machines.” In: *Formal Modeling and Analysis of Timed Systems - 9th International Conference, FORMATS 2011, Proceedings*. Ed. by Uli Fahrenberg and Stavros Tripakis. Vol. 6919. Lecture Notes in Computer Science. Springer, 2011, pp. 97–112. DOI: 10.1007/978-3-642-24310-3_8.
- [38] Patricia Bouyer et al. “Timed Automata Can Always Be Made Implementable.” In: *CONCUR 2011 - Concurrency Theory - 22nd International Conference, Proceedings*. Ed. by Joost-Pieter Katoen and Barbara König. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 76–91. DOI: 10.1007/978-3-642-23217-6_6.
- [39] Alan W. Brown and John A. McDermid. “The Art and Science of Software Architecture.” In: *Software Architecture, First European Conference, ECSA 2007, Proceedings*. Ed. by Flávio Oquendo. Vol. 4758. Lecture Notes in Computer Science. Springer, 2007, pp. 237–256. DOI: 10.1007/978-3-540-75132-8_19.
- [40] Julian Brunner. “Büchi Complementmentation.” In: *Arch. Formal Proofs 2017* (2017). URL: https://www.isa-afp.org/entries/Buchi_Complementmentation.html.
- [41] Julian Brunner and Peter Lammich. “Formal Verification of an Executable LTL Model Checker with Partial Order Reduction.” In: *J. Autom. Reason.* 60.1 (2018), pp. 3–21. DOI: 10.1007/s10817-017-9418-4.
- [42] Julian Brunner, Benedikt Seidl, and Salomon Sickert. “A Verified and Compositional Translation of LTL to Deterministic Rabin Automata.” In: *10th International Conference on Interactive Theorem Proving, ITP 2019*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 11:1–11:19. DOI: 10.4230/LIPIcs.ITP.2019.11.
- [43] Véronique Bruyère, Emmanuel Dall’Olio, and Jean-François Raskin. “Durations and parametric model-checking in timed automata.” In: *ACM Trans. Comput. Log.* 9.2 (2008), 12:1–12:23. DOI: 10.1145/1342991.1342996.

- [44] Carlos E. Budde et al. “JANI: Quantitative Model and Tool Interaction.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Proceedings, Part II*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. 2017, pp. 151–168. DOI: 10.1007/978-3-662-54580-5_9.
- [45] Lukas Bulwahn et al. “Imperative Functional Programming with Isabelle/HOL.” In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 134–149. DOI: 10.1007/978-3-540-71067-7_14.
- [46] Franck Cassez, Thomas A. Henzinger, and Jean-François Raskin. “A Comparison of Control Problems for Timed and Hybrid Systems.” In: *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002, Proceedings*. Ed. by Claire Tomlin and Mark R. Greenstreet. Vol. 2289. Lecture Notes in Computer Science. Springer, 2002, pp. 134–148. DOI: 10.1007/3-540-45873-5_13.
- [47] P Castéran and D Rouillard. “Towards a Generic Tool for Reasoning about Labeled Transition Systems.” In: *TPHOLs 2001: Supplemental Proceedings*. 2001.
- [48] Robert N. Charette. *This Car Runs on Code*. 2009. URL: <https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code> (visited on 05/28/2020).
- [49] Arthur Charguéraud. “Characteristic formulae for the verification of imperative programs.” In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 418–430. DOI: 10.1145/2034773.2034828.
- [50] Arthur Charguéraud. “Higher-order representation predicates in separation logic.” In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. Ed. by Jeremy Avigad and Adam Chlipala. ACM, 2016, pp. 3–14. DOI: 10.1145/2854065.2854068.
- [51] Arthur Charguéraud. “Program verification through characteristic formulae.” In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 321–332. DOI: 10.1145/1863543.1863590.
- [52] Joonwon Choi et al. “Kami: a platform for high-level parametric hardware specification and its modular verification.” In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 24:1–24:30. DOI: 10.1145/3110268.
- [53] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*. Springer International Publishing, May 2018, pp. 1–1210. ISBN: 9783319105758. DOI: 10.1007/978-3-319-10575-8.
- [54] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. “Refinements for Free!” In: *Certified Programs and Proofs - Third International Conference, CPP 2013, Proceedings*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 147–162. DOI: 10.1007/978-3-319-03545-1_10.
- [55] Robert L. Constable et al. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. ISBN: 978-0-13-451832-9. URL: <http://dl.acm.org/citation.cfm?id=10510>.

Bibliography

- [56] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. “Efficient Certified Resolution Proof Checking.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 118–135. DOI: 10.1007/978-3-662-54577-5_7.
- [57] Luís Cruz-Filipe et al. “Efficient Certified RAT Verification.” In: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Proceedings*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. Springer, 2017, pp. 220–236. DOI: 10.1007/978-3-319-63046-5_14.
- [58] Lukasz Czajka and Cezary Kaliszyk. “Hammer for Coq: Automation for Dependent Type Theory.” In: *J. Autom. Reason.* 61.1-4 (2018), pp. 423–453. DOI: 10.1007/s10817-018-9458-4.
- [59] Conrado Daws and Stavros Tripakis. “Model Checking of Real-Time Reachability Properties Using Abstractions.” In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Proceedings*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 313–329. DOI: 10.1007/BFb0054180.
- [60] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. “Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant.” In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 689–700. DOI: 10.1145/2676726.2677006.
- [61] Edsger W. Dijkstra. “The Humble Programmer.” In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591.
- [62] David L. Dill. “Timing Assumptions and Verification of Finite-State Concurrent Systems.” In: *Automatic Verification Methods for Finite State Systems, International Workshop, Proceedings*. Ed. by Joseph Sifakis. Vol. 407. Lecture Notes in Computer Science. Springer, 1989, pp. 197–212. DOI: 10.1007/3-540-52148-8_17.
- [63] Manuel Eberl. “Verified Real Asymptotics in Isabelle/HOL.” In: *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019*. Ed. by James H. Davenport, Dongming Wang, Manuel Kauers, and Russell J. Bradford. ACM, 2019, pp. 147–154. DOI: 10.1145/3326229.3326240.
- [64] Romain Edelmann, Jad Hamza, and Viktor Kuncak. “Zippy LL(1) parsing with derivatives.” In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 1036–1051. DOI: 10.1145/3385412.3385992.
- [65] Javier Esparza et al. “A Fully Verified Executable LTL Model Checker.” In: *Computer Aided Verification - 25th International Conference, CAV 2013, Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 463–478. DOI: 10.1007/978-3-642-39799-8_31.
- [66] Rebeka Farkas, Tamás Tóth, Ákos Hajdu, and András Vörös. “Backward reachability analysis for timed automata with variables.” In: *Electronic Communications of the EASST* (2019). ISSN: 18632122. DOI: 10.14279/tuj.eceasst.76.1076.

- [67] Jean-Christophe Filliâtre and Martin Chlochard. *Warshall algorithm*. URL: http://toccata.lri.fr/gallery/warshall%7B_%7Dalgorithm.en.html (visited on 07/06/2020).
- [68] Alain Finkel and Philippe Schnoebelen. “Well-structured transition systems everywhere!” In: *Theor. Comput. Sci.* 256.1-2 (2001), pp. 63–92. DOI: 10.1016/S0304-3975(00)00102-X.
- [69] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer, 1990. ISBN: 978-1-4684-0359-6. DOI: 10.1007/978-1-4684-0357-2.
- [70] Robert W Floyd. “Algorithm 97: Shortest path.” In: *Commun. ACM* 5.6 (1962), p. 345. DOI: 10.1145/367766.368168.
- [71] Simson Garfinkel. *History’s Worst Software Bugs | WIRED*. 2005. URL: <https://www.wired.com/2005/11/historys-worst-software-bugs/> (visited on 08/27/2020).
- [72] Manuel Garnacho, Jean-Paul Bodeveix, and Mamoun Filali-Amine. “A Mechanized Semantic Framework for Real-Time Systems.” In: *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Proceedings*. Ed. by Víctor A. Braberman and Laurent Fribourg. Vol. 8053. Lecture Notes in Computer Science. Springer, 2013, pp. 106–120. DOI: 10.1007/978-3-642-40229-6_8.
- [73] Paul Gastin, Sayan Mukherjee, and B. Srivathsan. “Fast Algorithms for Handling Diagonal Constraints in Timed Automata.” In: *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 41–59. DOI: 10.1007/978-3-030-25540-4_3.
- [74] Rodolfo Gómez and Howard Bowman. “Efficient Detection of Zeno Runs in Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Proceedings*. Ed. by Jean-François Raskin and P. S. Thiagarajan. Vol. 4763. Lecture Notes in Computer Science. Springer, 2007, pp. 195–210. DOI: 10.1007/978-3-540-75454-1_15.
- [75] Georges Gonthier. “The Four Colour Theorem: Engineering of a Formal Proof.” In: *Computer Mathematics, 8th Asian Symposium, ASCM 2007. Revised and Invited Papers*. Ed. by Deepak Kapur. Vol. 5081. Lecture Notes in Computer Science. Springer, 2007, p. 333. DOI: 10.1007/978-3-540-87827-8_28.
- [76] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem.” In: *Interactive Theorem Proving - 4th International Conference, ITP 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.
- [77] Michael J. C. Gordon. “Introduction to the HOL System.” In: *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*. Ed. by Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley. IEEE Computer Society, 1991, pp. 2–3.
- [78] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Vol. 78. Lecture Notes in Computer Science. Springer, 1979. ISBN: 3-540-09724-4. DOI: 10.1007/3-540-09724-4.
- [79] Stijn de Gouw et al. “Verifying OpenJDK’s Sort Method for Generic Collections.” In: *J. Autom. Reason.* 62.1 (2019), pp. 93–126. DOI: 10.1007/s10817-017-9426-4.

Bibliography

- [80] Alberto Griggio, Marco Roveri, and Stefano Tonetta. “Certifying Proofs for LTL Model Checking.” In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018*. Ed. by Nikolaj Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8603022.
- [81] Ronghui Gu et al. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.” In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 653–669. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu>.
- [82] Leo Gugerty. “Newell and Simon’s Logic Theorist: Historical Background and Impact on Cognitive Modeling.” In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting 50.9* (Oct. 2006), pp. 880–884. ISSN: 1541-9312. DOI: 10.1177/154193120605000904.
- [83] Florian Haftmann. “Code Generation from Specifications in Higher Order Logic.” Dissertation. Technische Universität München, 2009. URL: <http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20091208-886023-1-1>.
- [84] Florian Haftmann and Tobias Nipkow. “Code Generation via Higher-Order Rewrite Systems.” In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer, 2010, pp. 103–117. DOI: 10.1007/978-3-642-12251-4_9.
- [85] Niusha Hakimipour, Paul A. Strooper, and Andy J. Wellings. “TART: Timed-Automata to Real-Time Java Tool.” In: *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010*. Ed. by José Luiz Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini. IEEE Computer Society, 2010, pp. 299–309. DOI: 10.1109/SEFM.2010.39.
- [86] Thomas C. Hales et al. “A formal proof of the Kepler conjecture.” In: *CoRR* abs/1501.02155 (2015). arXiv: 1501.02155.
- [87] Nick Harley. *11 of the most costly software errors in history [2019 update]* · *Raygun Blog*. 2019. URL: <https://raygun.com/blog/costly-software-errors-history/> (visited on 08/27/2020).
- [88] John Harrison. “HOL Light: An Overview.” In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Proceedings*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 60–66. DOI: 10.1007/978-3-642-03359-9_4.
- [89] John Harrison. “Towards Self-verification of HOL Light.” In: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*. Ed. by Ulrich Furbach and Natarajan Shankar. Vol. 4130. Lecture Notes in Computer Science. Springer, 2006, pp. 177–191. DOI: 10.1007/11814771_17.
- [90] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund. “Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL.” In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS ’97)*. IEEE Computer Society, 1997, pp. 2–13. DOI: 10.1109/REAL.1997.641264.

- [91] Frédéric Herbreteau, Dileep Kini, B. Srivathsan, and Igor Walukiewicz. “Using non-convex approximations for efficient analysis of timed automata.” In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011*. Ed. by Supratik Chakraborty and Amit Kumar. Vol. 13. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 78–89. DOI: 10.4230/LIPIcs.FSTTCS.2011.78.
- [92] Frédéric Herbreteau and Gérald Point. *TChecker*. 2019. URL: <https://github.com/fredher/tchecker>.
- [93] Frédéric Herbreteau and B. Srivathsan. “Coarse abstractions make Zeno behaviours difficult to detect.” In: *Log. Methods Comput. Sci.* 9.1 (2011). DOI: 10.2168/LMCS-9(1:6)2013.
- [94] Frédéric Herbreteau, B. Srivathsan, Thanh-Tung Tran, and Igor Walukiewicz. “Why Liveness for Timed Automata Is Hard, and What We Can Do About It.” In: *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016*. Ed. by Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen. Vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 48:1–48:14. DOI: 10.4230/LIPIcs.FSTTCS.2016.48.
- [95] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. “Better abstractions for timed automata.” In: *Inf. Comput.* 251 (2016), pp. 67–90. DOI: 10.1016/j.ic.2016.07.004.
- [96] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. “Efficient emptiness check for timed Büchi automata.” In: *Formal Methods in System Design* 40.2 (Apr. 2012), pp. 122–146. ISSN: 0925-9856. DOI: 10.1007/s10703-011-0133-1. arXiv: 1104.1540.
- [97] Frédéric Herbreteau and Thanh-Tung Tran. “Improving Search Order for Reachability Testing in Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Proceedings*. Ed. by Sriram Sankaranarayanan and Enrico Vicario. Vol. 9268. Lecture Notes in Computer Science. Springer, 2015, pp. 124–139. DOI: 10.1007/978-3-319-22975-1_9.
- [98] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. “Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer.” In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 228–245. DOI: 10.1007/978-3-319-40970-2_15.
- [99] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. “Efficient, Verified Checking of Propositional Proofs.” In: *Interactive Theorem Proving - 8th International Conference, ITP 2017, Proceedings*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Vol. 10499. Lecture Notes in Computer Science. Springer, 2017, pp. 269–284. DOI: 10.1007/978-3-319-66107-0_18.
- [100] C. A. R. Hoare. “Proof of Correctness of Data Representations.” In: *Acta Informatica* 1 (1972), pp. 271–281. DOI: 10.1007/BF00289507.
- [101] Martin Hofmann, Christian Neukirchen, and Harald Rueß. “Certification for μ -calculus with winning strategies.” In: *Lecture Notes in Computer Science*. Vol. 9641. Springer Verlag, 2016, pp. 111–128. ISBN: 9783319325811. DOI: 10.1007/978-3-319-32582-8_8.

Bibliography

- [102] Johannes Hölzl. “Markov Chains and Markov Decision Processes in Isabelle/HOL.” In: *J. Autom. Reason.* 59.3 (2017), pp. 345–387. DOI: 10.1007/s10817-016-9401-5.
- [103] Brian Huffman and Ondrej Kunčar. “Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL.” In: *Certified Programs and Proofs - Third International Conference, CPP 2013, Proceedings*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 131–146. DOI: 10.1007/978-3-319-03545-1_9.
- [104] Lars Hupel and Tobias Nipkow. “A Verified Compiler from Isabelle/HOL to CakeML.” In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 999–1026. ISBN: 978-3-319-89884-1. DOI: 10.1007/978-3-319-89884-1_35.
- [105] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based software testing and analysis with C#*. Vol. 9780521886. Cambridge University Press, Jan. 2007, pp. 1–349. ISBN: 9780511619540. DOI: 10.1017/CB09780511619540.
- [106] Marie-Christine Jakobs and Heike Wehrheim. “Certification for Configurable Program Analysis.” In: *SPIN 2014*. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 30–39. ISBN: 9781450324526. DOI: 10.1145/2632362.2632372.
- [107] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. “Scaling up Uppaal Automatic Verification of Real-Time Systems Using Compositionality and Abstraction.” In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000, Proceedings*. Ed. by Mathai Joseph. Vol. 1926. Lecture Notes in Computer Science. Springer, 2000, pp. 19–30. DOI: 10.1007/3-540-45352-0_4.
- [108] Zhihao Jiang et al. “Modeling and Verification of a Dual Chamber Implantable Pacemaker.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Proceedings*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 188–203. DOI: 10.1007/978-3-642-28756-5_14.
- [109] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. “Validating LR(1) Parsers.” In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 397–416. DOI: 10.1007/978-3-642-28869-2_20.
- [110] Cezary Kaliszyk and Josef Urban. “HOL(y)Hammer: Online ATP Service for HOL Light.” In: *Math. Comput. Sci.* 9.1 (2015), pp. 5–22. DOI: 10.1007/s11786-014-0182-0.
- [111] Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. “SMT-Based Induction Methods for Timed Systems.” In: *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, Proceedings*. Ed. by Marcin Jurdzinski and Dejan Nickovic. Vol. 7595. Lecture Notes in Computer Science. Springer, 2012, pp. 171–187. DOI: 10.1007/978-3-642-33365-1_13.
- [112] Gerwin Klein et al. “seL4: Formal Verification of an OS Kernel.” In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596.

- [113] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications - Second Edition*. 2011. ISBN: 0792398947. DOI: 10.1007/978-1-4419-8237-7.
- [114] Adam Koprowski and Henri Binsztok. “TRX: A Formally Verified Parser Interpreter.” In: *Log. Methods Comput. Sci.* 7.2 (2011). DOI: 10.2168/LMCS-7(2:18)2011.
- [115] Piotr Kordy, Rom Langerak, Sjouke Mauw, and Jan Willem Polderman. “A Symbolic Algorithm for the Analysis of Robust Timed Automata.” In: *FM 2014: Formal Methods - 19th International Symposium, Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 351–366. DOI: 10.1007/978-3-319-06410-9_25.
- [116] Tuomas Kuismin and Keijo Heljanko. “Increasing Confidence in Liveness Model Checking Results with Proofs.” In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013*. Ed. by Valeria Bertacco and Axel Legay. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 32–43. DOI: 10.1007/978-3-319-03077-7_3.
- [117] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. “Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation.” In: *J. Autom. Reason.* 56.3 (2016), pp. 221–259. DOI: 10.1007/s10817-015-9357-x.
- [118] Orna Kupferman. “Automata Theory and Model Checking.” In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 107–151. DOI: 10.1007/978-3-319-10575-8_4.
- [119] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems.” In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.
- [120] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. “Stochastic Games for Verification of Probabilistic Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Proceedings*. Ed. by Joël Ouaknine and Frits W. Vaandrager. Vol. 5813. Lecture Notes in Computer Science. Springer, 2009, pp. 212–227. DOI: 10.1007/978-3-642-04368-0_17.
- [121] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. “Automatic verification of real-time systems with discrete probability distributions.” In: *Theor. Comput. Sci.* 282.1 (2002), pp. 101–150. DOI: 10.1016/S0304-3975(01)00046-9.
- [122] Alfons Laarman et al. “Multi-core emptiness checking of timed Büchi automata using inclusion abstraction.” In: *Lecture Notes in Computer Science*. Vol. 8044 LNCS. 2013, pp. 968–983. ISBN: 9783642397981. DOI: 10.1007/978-3-642-39799-8_69.
- [123] Peter Lammich. “Automatic Data Refinement.” In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 84–99. DOI: 10.1007/978-3-642-39634-2_9.
- [124] Peter Lammich. “Efficient Verified (UN)SAT Certificate Checking.” In: *J. Autom. Reason.* 64.3 (2020), pp. 513–532. DOI: 10.1007/s10817-019-09525-z.

Bibliography

- [125] Peter Lammich. “Generating Verified LLVM from Isabelle/HOL.” In: *10th International Conference on Interactive Theorem Proving, ITP 2019*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 22:1–22:19. DOI: 10.4230/LIPIcs.ITP.2019.22.
- [126] Peter Lammich. “Refinement to Imperative/HOL.” In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 253–269. DOI: 10.1007/978-3-319-22102-1_17.
- [127] Peter Lammich and Andreas Lochbihler. “Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches.” In: *J. Autom. Reason.* 63.1 (2019), pp. 53–94. DOI: 10.1007/s10817-018-9461-9.
- [128] Peter Lammich and Thomas Tuerk. “Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm.” In: *Proc. of ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 166–182.
- [129] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. “A Verified LL(1) Parser Generator.” In: *10th International Conference on Interactive Theorem Proving, ITP 2019*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 24:1–24:18. DOI: 10.4230/LIPIcs.ITP.2019.24.
- [130] John Launchbury and Simon L. Peyton Jones. “Lazy Functional State Threads.” In: *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI)*. Ed. by Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa. ACM, 1994, pp. 24–35. DOI: 10.1145/178243.178246.
- [131] Edward Lee and Sanjit Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Jan. 2011. ISBN: 9780201101799.
- [132] Xavier Leroy. “Formal verification of a realistic compiler.” In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.
- [133] G Li. “Checking timed Büchi automata emptiness using LU-abstractions.” In: *FORMATS*. Vol. 5813. LNCS. 2009, pp. 228–242.
- [134] Wenda Li, Grant Olney Passmore, and Lawrence C. Paulson. “Deciding Univariate Polynomial Problems Using Untrusted Certificates in Isabelle/HOL.” In: *J. Autom. Reason.* 62.1 (2019), pp. 69–91. DOI: 10.1007/s10817-017-9424-6.
- [135] Liya Liu, Osman Hasan, Vincent Aravantinos, and Sofiène Tahar. “Formal Reasoning about Classified Markov Chains in HOL.” In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 295–310. DOI: 10.1007/978-3-642-39634-2_22.
- [136] Liya Liu, Osman Hasan, and Sofiène Tahar. “Formal Reasoning About Finite-State Discrete-Time Markov Chains in HOL.” In: *J. Comput. Sci. Technol.* 28.2 (2013), pp. 217–231. DOI: 10.1007/s11390-013-1324-6.
- [137] Andreas Lochbihler. “Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable.” In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 116–132. DOI: 10.1007/978-3-642-39634-2_11.

- [138] Andreas Lochbihler. “Making the java memory model safe.” In: *ACM Trans. Program. Lang. Syst.* 35.4 (2013), 12:1–12:65. DOI: 10.1145/2518191.
- [139] Pamela McCorduck. *Machines who think: a personal inquiry into the history and prospects of artificial intelligence*. 2nd ed. A K Peters/CRC Press, 2004, pp. 123–125. ISBN: 978-1-56881-205-2.
- [140] Antoine Miné. “A New Numerical Abstract Domain Based on Difference-Bound Matrices.” In: *Programs as Data Objects, Second Symposium, PADO 2001, Proceedings*. Ed. by Olivier Danvy and Andrzej Filinski. Vol. 2053. Lecture Notes in Computer Science. Springer, 2001, pp. 155–172. DOI: 10.1007/3-540-44978-7_10.
- [141] Antoine Miné. “The octagon abstract domain.” In: *Higher-Order and Symbolic Computation* (2006). ISSN: 13883690. DOI: 10.1007/s10990-006-8609-1.
- [142] Georges Morbé, Florian Pigorsch, and Christoph Scholl. “Fully Symbolic Model Checking for Timed Automata.” In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 616–632. DOI: 10.1007/978-3-642-22110-1_50.
- [143] Magnus O. Myreen and Scott Owens. “Proof-producing translation of higher-order logic into pure and stateful ML.” In: *J. Funct. Program.* 24.2-3 (2014), pp. 284–315. DOI: 10.1017/S0956796813000282.
- [144] Kedar S. Namjoshi. “Certifying Model Checkers.” In: *Computer Aided Verification, 13th International Conference, CAV 2001, Proceedings*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 2–13. DOI: 10.1007/3-540-44585-4_2.
- [145] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1994. ISBN: 9780080887180.
- [146] René Neumann. “Using Promela in a Fully Verified Executable LTL Model Checker.” In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Revised Selected Papers*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 105–114. DOI: 10.1007/978-3-319-12154-3_7.
- [147] Tobias Nipkow. “Abstract Interpretation of Annotated Commands.” In: *Interactive Theorem Proving - Third International Conference, ITP 2012, Proceedings*. Ed. by Lennart Beringer and Amy P. Felty. Vol. 7406. Lecture Notes in Computer Science. Springer, 2012, pp. 116–132. DOI: 10.1007/978-3-642-32347-8_9.
- [148] Tobias Nipkow. “Constructive Rewriting.” In: *Comput. J.* 34.1 (1991), pp. 34–41. DOI: 10.1093/comjnl/34.1.34.
- [149] Tobias Nipkow, Manuel Eberl, and Maximilian P. L. Haslbeck. “Verified Textbook Algorithms. A Biased Survey.” In: *ATVA 2020, Automated Technology for Verification and Analysis*. Ed. by Dang Van Hung and Oleg Sokolsky. Vol. ? LNCS. Invited paper. To appear. Springer, 2020, ?–?
- [150] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. 2014. DOI: 10.1007/978-3-319-10542-0.
- [151] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

Bibliography

- [152] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.
- [153] Gethin Norman, David Parker, and Jeremy Sproston. “Model checking for probabilistic timed automata.” In: *Formal Methods in System Design* (2013). ISSN: 09259856. DOI: 10.1007/s10703-012-0177-x.
- [154] Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. “Verification of Certifying Computations through AutoCorres and Simpl.” In: *NASA Formal Methods - 6th International Symposium, NFM 2014. Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 8430. Lecture Notes in Computer Science. Springer, 2014, pp. 46–61. DOI: 10.1007/978-3-319-06200-6_4.
- [155] Ernst-Rüdiger Olderog and Mani Swaminathan. “Layered Composition for Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Proceedings*. Ed. by Krishnendu Chatterjee and Thomas A. Henzinger. Vol. 6246. Lecture Notes in Computer Science. Springer, 2010, pp. 228–242. DOI: 10.1007/978-3-642-15297-9_18.
- [156] Sam Owre, John M. Rushby, and Natarajan Shankar. “PVS: A Prototype Verification System.” In: *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Proceedings*. Ed. by Deepak Kapur. Vol. 607. Lecture Notes in Computer Science. Springer, 1992, pp. 748–752. DOI: 10.1007/3-540-55602-8_217.
- [157] Miroslav Pajic et al. “Safety-critical medical device development using the UPP2SF model translation tool.” In: *ACM Trans. Embed. Comput. Syst.* 13.4s (2014), 127:1–127:26. DOI: 10.1145/2584651.
- [158] Christine Paulin-Mohring. “Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions).” PhD thesis. Paris Diderot University, France, 1989. URL: <https://tel.archives-ouvertes.fr/tel-00431825>.
- [159] Christine Paulin-Mohring. “Modélisation de Timed Automata in Coq.” In: *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Proceedings*. Ed. by Naoki Kobayashi and Benjamin C. Pierce. Vol. 2215. Lecture Notes in Computer Science. Springer, 2001, pp. 298–315. DOI: 10.1007/3-540-45500-0_15.
- [160] Lawrence Paulson. “Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers.” In: 2018. DOI: 10.29007/tnfd.
- [161] Benjamin C. Pierce et al. *Logical Foundations*. Software Foundations series, volume 1. Version 5.5. <http://www.cis.upenn.edu/~bcpierce/sf>. Electronic textbook, May 2018.
- [162] Clément Pit-Claudel et al. “Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs.” In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Proceedings, Part II*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 119–137. DOI: 10.1007/978-3-030-51054-1_7.
- [163] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018. ISBN: 978-3-319-63587-3. DOI: 10.1007/978-3-319-63588-0.
- [164] Amir Pnueli. “The temporal logic of programs.” In: *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*. Vol. 1977-October. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/sfcs.1977.32.

- [165] John C. Reynolds. “An Overview of Separation Logic.” In: *Verified Software: Theories, Tools, Experiments, VSTTE 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer, 2005, pp. 460–469. DOI: 10.1007/978-3-540-69149-5_49.
- [166] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures.” In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.
- [167] Talia Ringer et al. “QED at Large: A Survey of Engineering of Formally Verified Software.” In: *Found. Trends Program. Lang.* 5.2-3 (2019), pp. 102–281. DOI: 10.1561/25000000045.
- [168] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. “Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL.” In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 424–439. DOI: 10.1007/978-3-642-03359-9_29.
- [169] Mathijs Schuts, Feng Zhu, Faranak Heidarian, and Frits W. Vaandrager. “Modelling Clock Synchronization in the Chess gMAC WSN Protocol.” In: *Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications, QFM 2009*. Ed. by Suzana Andova et al. Vol. 13. EPTCS. 2009, pp. 41–54. DOI: 10.4204/EPTCS.13.4.
- [170] Stefan Schwoon and Javier Esparza. “A Note on On-the-Fly Verification Algorithms.” In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 174–190. DOI: 10.1007/978-3-540-31980-1_12.
- [171] Stephen F. Siegel. “What’s Wrong with On-the-Fly Partial Order Reduction.” In: *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 478–495. DOI: 10.1007/978-3-030-25543-5_27.
- [172] David P. L. Simons and Mariëlle Stoelinga. “Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k.” In: *Int. J. Softw. Tools Technol. Transf.* 3.4 (2001), pp. 469–485. DOI: 10.1007/s100090100059.
- [173] Carsten Sinz and Armin Biere. “Extended Resolution Proofs for Conjoining BDDs.” In: *Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, Proceedings*. Ed. by Dima Grigoriev, John Harrison, and Edward A. Hirsch. Vol. 3967. Lecture Notes in Computer Science. Springer, 2006, pp. 600–611. DOI: 10.1007/11753728_60.
- [174] Christoph Sprenger. “A verified model checker for the modal mu-calculus in Coq.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1384. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 167–183. ISBN: 978-3-540-64356-2, 978-3-540-69753-4. URL: <http://proxy.library.upenn.edu:2230/chapter/10.1007%7B%5C%7D2FBFb0054171>.
- [175] Christian Sternagel, René Thiemann, et al. *IsaFoR/CeTA: An Isabelle/HOL Formalization of Rewriting for Certified Tool Assertions*. URL: <http://cl-informatik.uibk.ac.at/isafor>.

Bibliography

- [176] Mani Swaminathan and Martin Fränzle. “A Symbolic Decision Procedure for Robust Safety of Timed Systems.” In: *14th International Symposium on Temporal Representation and Reasoning (TIME 2007)*. IEEE Computer Society, 2007, p. 192. DOI: 10.1109/TIME.2007.39.
- [177] Yong Kiam Tan et al. “The verified CakeML compiler backend.” In: *J. Funct. Program.* 29 (2019), e2. DOI: 10.1017/S0956796818000229.
- [178] The Coq Development Team. *The Coq Proof Assistant, version 8.10.0*. 2019. DOI: 10.5281/zenodo.3476303.
- [179] Stavros Tripakis. “L’analyse formelle des systèmes temporisés en pratique. (The Formal Analysis of Timed Systems in Practice).” PhD thesis. Joseph Fourier University, Grenoble, France, 1998. URL: <https://tel.archives-ouvertes.fr/tel-00004907>.
- [180] Stavros Tripakis. “Verifying Progress in Timed Systems.” In: *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS’99, Proceedings*. Ed. by Joost-Pieter Katoen. Vol. 1601. Lecture Notes in Computer Science. Springer, 1999, pp. 299–314. DOI: 10.1007/3-540-48778-6_18.
- [181] Stavros Tripakis and Thao Dang. “Modeling, Verification and Testing using Timed and Hybrid Automata.” In: *Test* (2008). URL: <http://www-verimag.imag.fr/%7B~%7Dttripakis/papers/crc08.pdf>.
- [182] Stavros Tripakis and Sergio Yovine. “Timing Analysis and Code Generation of Vehicle Control Software using Taxys.” In: *Electron. Notes Theor. Comput. Sci.* 55.2 (2001), pp. 277–286. DOI: 10.1016/S1571-0661(04)00257-9.
- [183] Moshe Y. Vardi and Pierre Wolper. “Reasoning About Infinite Computations.” In: *Inf. Comput.* 115.1 (1994), pp. 1–37. DOI: 10.1006/inco.1994.1092.
- [184] Farn Wang. “Efficient verification of timed automata with BDD-like data structures.” In: *Int. J. Softw. Tools Technol. Transf.* 6.1 (2004), pp. 77–97. DOI: 10.1007/s10009-003-0135-4.
- [185] Stephen Warshall. “A Theorem on Boolean Matrices.” In: *Journal of the ACM* 9.1 (1962), pp. 11–12. DOI: 10.1145/321105.321107.
- [186] Makarius Wenzel. “Transitive closure according to Roy-Floyd-Warshall.” In: *Archive of Formal Proofs* (May 2014). http://isa-afp.org/entries/Roy_Floyd_Warshall.html, Formal proof development. ISSN: 2150-914x.
- [187] Markus Wenzel. “Isabelle/Isar — a versatile environment for human-readable formal proof documents.” Dissertation. München: Technische Universität München, 2002. URL: <http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss2002020117092>.
- [188] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. “DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs.” In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Proceedings*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 422–429. DOI: 10.1007/978-3-319-09284-3_31.
- [189] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. “Mechanical Verification of SAT Refutations with Extended Resolution.” In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 229–244. DOI: 10.1007/978-3-642-39634-2_18.

- [190] Freek Wiedijk. *The Seventeen Provers of the World, Foreword by Dana S. Scott*. Vol. 3600. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-30704-4. DOI: 10.1007/11542384.
- [191] Simon Wimmer. “Formalized Timed Automata.” In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Proceedings*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 425–440. DOI: 10.1007/978-3-319-43144-4_26.
- [192] Simon Wimmer. “Munta: A Verified Model Checker for Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Proceedings*. Ed. by Étienne André and Mariëlle Stoelinga. Vol. 11750. Lecture Notes in Computer Science. Springer, 2019, pp. 236–243. DOI: 10.1007/978-3-030-29662-9_14.
- [193] Simon Wimmer, Frédéric Herbreteau, and Jaco van de Pol. “Certifying Emptiness of Timed Büchi Automata.” In: *Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020*. Vol. 12288 LNCS. Lecture Notes in Computer Science. Springer, Sept. 2020, pp. 58–75. ISBN: 9783030576271. DOI: 10.1007/978-3-030-57628-8_4.
- [194] Simon Wimmer, Frédéric Herbreteau, and Jaco van de Pol. “Certifying Emptiness of Timed Büchi Automata.” In: *CoRR abs/2007.04150* (2020). arXiv: 2007.04150. URL: <https://arxiv.org/abs/2007.04150>.
- [195] Simon Wimmer and Johannes Hölzl. “MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper).” In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 597–603. DOI: 10.1007/978-3-319-94821-8_35.
- [196] Simon Wimmer and Peter Lammich. *Munta: A Fully Verified Model Checker for Timed Automata*. URL: <https://wimmers.github.io/munta/>.
- [197] Simon Wimmer and Peter Lammich. “Verified Model Checking of Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 61–78. DOI: 10.1007/978-3-319-89960-2_4.
- [198] Simon Wimmer and Joshua von Mutius. “Verified Certification of Reachability Checking for Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Proceedings, Part I*. Ed. by Armin Biere and David Parker. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 425–443. DOI: 10.1007/978-3-030-45190-5_24.
- [199] Niklaus Wirth. “Program Development by Stepwise Refinement.” In: *Communications of the ACM* (1971). ISSN: 15577317. DOI: 10.1145/362575.362577.
- [200] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. “Robust safety of timed automata.” In: *Formal Methods Syst. Des.* 33.1-3 (2008), pp. 45–84. DOI: 10.1007/s10703-008-0056-7.
- [201] Qingguo Xu and Huaikou Miao. “Formal Verification Framework for Safety of Real-Time System based on Timed Automata Model in PVS.” In: *Proceedings of the IASTED International Conference on Software Engineering*. Ed. by Peter Kokol. IASTED/ACTA Press, 2006, pp. 107–112.

Bibliography

- [202] Qingguo Xu and Huaikou Miao. “Manipulating Clocks in Timed Automata Using PVS.” In: *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, SNPD 200*. Ed. by Haeng-Kon Kim and Roger Y. Lee. IEEE Computer Society, 2009, pp. 555–560. DOI: 10.1109/SNPD.2009.69.
- [203] Bo Yang, Cailian Chen, Wenbin Dai, and Yiyin Wang. “A Comprehensive Overview of Cyber-Physical Systems: From Perspective of Feedback System.” In: *IEEE/CAA Journal of Automatica Sinica* 3 (Jan. 2016), pp. 1–14. DOI: 10.1109/JAS.2016.7373757.
- [204] Wang Yi, Paul Pettersson, and Mats Daniels. “Automatic verification of real-time communicating systems by constraint-solving.” In: *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques*. Ed. by Dieter Hogrefe and Stefan Leue. Vol. 6. IFIP Conference Proceedings. Chapman & Hall, 1994, pp. 243–258.
- [205] Xia Yin, Qingguo Xu, and Kunliang Han. “Modeling Predicate Abstraction of Timed Automata in PVS.” In: *2011 IEEE International Conference on Internet of Things (iThings) & 4th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*. IEEE Computer Society, 2011, pp. 438–443. DOI: 10.1109/iThings/CPSCom.2011.21.
- [206] Sergio Yovine. “KRONOS: A Verification Tool for Real-Time Systems.” In: *Int. J. Softw. Tools Technol. Transf.* 1.1-2 (1997), pp. 123–133. DOI: 10.1007/s100090050009.
- [207] Zhengqi Yu, Armin Biere, and Keijo Heljanko. “Certifying Hardware Model Checking Results.” In: *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Proceedings*. Ed. by Yamine Aït Ameur and Shengchao Qin. Vol. 11852. Lecture Notes in Computer Science. Springer, 2019, pp. 498–502. DOI: 10.1007/978-3-030-32409-4_32.

A Formalized Timed Automata

Original publication. This chapter was originally published as a full paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer. “Formalized Timed Automata.” In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Proceedings*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 425–440. DOI: 10.1007/978-3-319-43144-4_26

Synopsis. This work formalizes timed automata and the essential theory of reachability checking for timed automata in Isabelle/HOL. I define the main concepts, formalize the classic decidability result for the language emptiness problem (in terms of reachability) using the region construction, and prove correctness of the basic forward analysis operations on zones and DBMs. Finally, I outline how both streams of work can be combined to show that forward analysis with the common extrapolation correctly decides emptiness of timed automata.

Contribution. I am the sole author of this paper. Therefore, all contributions are mine.

Copyright notice. On the following pages the full article is reprinted by permission from Springer Nature Customer Service Centre GmbH. The original publication can be found under the DOI cited above.

Formalized Timed Automata

Simon Wimmer^(✉)

Institut für Informatik, Technische Universität München, Munich, Germany
wimmers@in.tum.de

Abstract. Timed automata are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. However, mistakes have previously been made. This particularly regards an approximation operation, which is used by model-checking algorithms to obtain a finite search space. The use of this operation left a soundness problem in the tools employing it, which was only discovered years after the first model checkers were devised. This work aims to provide certainty to our knowledge of the basic theory via formalization in Isabelle/HOL: we define the main concepts, formalize the classic decidability result for the language emptiness problem, prove correctness of the basic forward analysis operations, and finally outline how both streams of work can be combined to show that forward analysis with the common approximation operation correctly decides emptiness for the class of diagonal-free timed automata.

1 Introduction

The foundations of the theory of timed automata are presented in the seminal work of Alur and Dill [1,2]. They introduced the formalism as a model for systems with real-time constraints and showed how to decide the language emptiness problem via the so-called *region* construction. Unfortunately, the number of regions explored by this algorithm is exponential in the size of the automaton under consideration. Moreover, Alur and Dill also showed that the language emptiness problem for timed automata is PSPACE-hard. Still, the formalism is employed in practical model checking [12,13,19] by means of algorithms based on *Difference Bound Matrices* (DBMs). These algorithms (with some more elaborate optimizations) can cope with many interesting real-life model checking problems. The search space examined by the DBM algorithms is potentially infinite. Therefore an approximation is used to obtain a finite search space. The basic idea is to represent every state (called *zone*) by the smallest set of regions which contains the state.

It took nearly a decade after this operation was initially devised, until Patricia Bouyer discovered [5] that the common algorithmic realization of this operation diverges from its intended result: the computed result is always a convex union of

Supported by DFG project NI 491/16-1.

© Springer International Publishing Switzerland 2016
J.C. Blanchette and S. Merz (Eds.): ITP 2016, LNCS 9807, pp. 425–440, 2016.
DOI: 10.1007/978-3-319-43144-4_26

regions, whereas the smallest set of regions containing a zone can be non-convex. This left a soundness problem, which fortunately vanishes for the restricted class of so-called diagonal-free timed automata [6] (Sect. 2.1 precisely characterizes this class). While not as expressive as the full formalism of timed automata, this class is sufficient for modeling most of the problems of practical interest, which explains why the problem was not discovered for many years.

This work aims to solidify the theoretical grounds on which real-time model checking with *diagonal-free* timed automata stands, by formalizing the basic theory and algorithms in Isabelle/HOL, and then going the full length to prove Bouyer’s correctness result. Section 2 will present the formalization of the basic notions for diagonal-free timed automata. Then Sect. 3 will show how we formalized DBMs and obtained soundness and completeness results for their basic algorithms. This includes a formalization of the Floyd-Warshall algorithm. Afterwards (Sect. 4) we define the notion of regions and prove that they are suitable for deciding the emptiness problem on timed automata. Finally, in Sect. 5, a refined version of these regions will be used to precisely formalize the approximation operation. To tie the ends of our formalization together, this characterization of approximation will be connected with its algorithmic version. This enables us to reuse the decidability result on the first region construction to prove that DBM-based algorithms together with approximation can decide the language emptiness problem for diagonal-free timed automata. For lack of space, many of our definitions and proofs are shortened or stated informally. We refer the reader to the entry in the Archive of Formal Proofs [15] for the full version (over 18500 lines).

1.1 History and Related Work

As mentioned, the basic theory was devised by Alur and Dill [1, 2]. The use of DBMs was also proposed by Dill [10] and brought to practical model checking by Yi et al. [18]. Bouyer’s developments of our main correctness results are spread over two papers. The first one presents a generalization of timed automata to *updatable* timed automata and revisits the basic decidability results for this class [7]. The second one [6] connects these results with DBMs to prove that the combination of DBM-based forward analysis operations and approximation decides the language emptiness problem.

We are aware of one previous proof-assistant formalization of timed automata using PVS [16, 17]. This work has the basic decidability result using regions and claims to make some attempt to extend the formalization towards DBMs. Another line of work [11, 14] aims at modeling the class of *p-automata* [3] (which is undecidable in the general case) in Coq and proving properties of concrete p-automata within Coq. A similar approach was pursued with the help of Isabelle/HOL in the *CClair* project [8]. In contrast, the most important contributions of our work are the full formalization of the relevant DBM algorithms, and particularly the rather intricate developments towards the correctness proof for the approximation operation – both of which pertain to practical real-time model checking.

Unless otherwise stated, our formalizations of the basic notions and DBMs are based on a popular tutorial by Bengtsson and Yi [4], while the developments for the region constructions and the final correctness result follow Bouyer's precise work.

2 Diagonal-Free Timed Automata in Isabelle/HOL

2.1 Syntactic Definition

Compared to standard finite automata, timed automata introduce a notion of clocks. We will fix a type $'c$ for the space of clocks, type $'t$ for time, and a type $'s$ for locations. While most of our formalizations only require $'t$ to belong to a custom type class for totally ordered dense abelian groups, we worked on the concrete type *real* for the region construction for simplicity. Figure 1 depicts an example of a diagonal-free timed automaton.

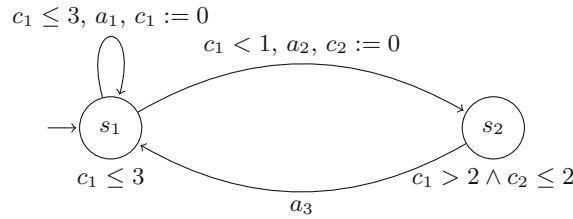


Fig. 1. Example of a diagonal-free timed automaton with two clocks.

Locations and transitions are guarded with *clock constraints*, which have to be fulfilled to stay in a location or to transition between them. The variants of these constraints are modeled by

```
datatype ('c, 't) cconstraint =
  AND (('c, 't) cconstraint) (('c, 't) cconstraint) |
  LT 'c 't | LE 'c 't | EQ 'c 't | GT 'c 't | GE 'c 't
```

where the atomic constraints in the second line represent the constraint $c \sim d$ for $\sim = <, \leq, =, >, \geq$, respectively. The sole difference to the full class of timed automata is that those would also allow constraints of the form $c_1 - c_2 \sim d$. We define a timed automaton A as a pair $(\mathcal{T}, \mathcal{I})$ where $\mathcal{I} :: 's \Rightarrow ('c, 't) cconstraint$ is an assignment of clock invariants to locations; \mathcal{T} is a set of transitions written as $A \vdash l \xrightarrow{g, a, r} l'$ where

- $l :: 's$ and $l' :: 's$ are start and successor location,
- $g :: ('c, 't) cconstraint$ is the guard of the transition,
- $a :: 'a$ is an action label,
- and $r :: 'c list$ is a list of clocks that will be reset to zero when the transition is taken.

Standard definitions of timed automata would include a fixed set of locations with a designated start location and a set of end locations. The language emptiness problem usually asks if any number of legal transitions can be taken to reach an end location from the start location. Thus we can confine ourselves to study reachability and implicitly assume the set of locations to be given by the transitions of the automaton. Note that although the definition of clock constraints allows constants from the whole time space, we will later crucially restrict them to the natural numbers in order to obtain decidability.

2.2 Operational Semantics

We want to define an operational semantics for timed automata via an inductive relation. States of timed automata are pairs of a location and a *clock valuation* of type $'c \Rightarrow 't$ assigning time values to clocks. Time lapse is modeled by shifting a clock valuation u by a constant value d : $u \oplus d = (\lambda x. u x + d)$. Finally, we connect clock valuations and constraints by writing, for instance, $u \vdash \text{AND} (LT\ c_1\ 1) (EQ\ c_2\ 2)$ if $u\ c_1 < 1$ and $u\ c_2 = 2$. The precise definition is standard.

Using these definitions, the operational semantics can be defined as a relation between pairs of locations and clock valuations. More specifically, we define *action steps*

$$\frac{A \vdash l \xrightarrow{g,a,r} l' \wedge u \vdash g \wedge u' \vdash \text{inv-of } A\ l' \wedge u' = [r \rightarrow 0]u}{A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle}$$

and *delay steps* via $\frac{u \vdash \text{inv-of } A\ l \wedge u \oplus d \vdash \text{inv-of } A\ l \wedge 0 \leq d}{A \vdash \langle l, u \rangle \rightarrow^d \langle l, u \oplus d \rangle}$. Here *inv-of*

$(\mathcal{T}, \mathcal{I}) = \mathcal{I}$ and the notation $[r \rightarrow 0]u$ means that we update the clocks in r to 0 in u . We write $A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ if either $A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$ or $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$.

2.3 Zone Semantics

The first conceptual step to get from this abstract operational semantics towards concrete algorithms on DBMs is to consider *zones*. Informally, the concept is simple; a zone is the set of clock valuations fulfilling a clock constraint: $('c, 't)\ \text{zone} \equiv ('c \Rightarrow 't)\ \text{set}$. This allows us to abstract from a concrete state $\langle l, u \rangle$ to a pair of location and zone $\langle l, Z \rangle$. We need the following operations on zones:

$$Z^\uparrow = \{u \oplus d \mid u \in Z \wedge 0 \leq d\} \text{ and } Z_{r \rightarrow 0} = \{[r \rightarrow 0]u \mid u \in Z\}.$$

Naturally, we define a zone-based semantics by means of another inductive relation:

$$\frac{A \vdash \langle l, Z \rangle \rightsquigarrow \langle l, (Z \cap \{u \mid u \vdash \text{inv-of } A \ l\})^\dagger \cap \{u \mid u \vdash \text{inv-of } A \ l\} \rangle}{A \vdash l \xrightarrow{g, a, r} l'}$$

$$\frac{}{A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', (Z \cap \{u \mid u \vdash g\})_r \rightarrow \emptyset \cap \{u \mid u \vdash \text{inv-of } A \ l'\} \rangle}$$

With the help of two easy inductive arguments one can show soundness and completeness of this semantics w.r.t. the original semantics (where $*$ is the *Kleene star* operator):

$$\begin{aligned} \text{(Sound)} \quad & A \vdash \langle l, Z \rangle \rightsquigarrow^* \langle l', Z' \rangle \wedge u' \in Z' \implies \exists u \in Z. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle \\ \text{(Complete)} \quad & A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle \wedge u \in Z \\ & \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow^* \langle l', Z' \rangle \wedge u' \in Z' \end{aligned}$$

This is an example of where proof assistants really shine. Not only are our Isabelle proofs shorter to write down than for example the proof given in [18] – we have also found that the less general version given there (i.e. where $Z = \{u\}$) yields an induction hypothesis that is not strong enough in the completeness proof. This slight lapse is hard to detect in a human-written proof.

3 Difference Bound Matrices

3.1 Fundamentals

Difference Bound Matrices constrain differences of clocks (or more precisely, the difference of values assigned to individual clocks by a valuation). The possible constraints are given by:

$$\text{datatype } 't \text{ DBMEntry} = \text{Le } 't \mid \text{Lt } 't \mid \infty$$

This yields a simple definition of DBMs: $'t \text{ DBM} \equiv \text{nat} \Rightarrow \text{nat} \Rightarrow 't \text{ DBMEntry}$. To relate clocks with rows and columns of a DBM, we use a numbering $v :: 'c \Rightarrow \text{nat}$ for clocks. DBMs will regularly be accompanied by a natural number n , which designates the number of clocks constrained by the matrix. Although this definition complicates our formalization at times, we hope that it allows us to easily obtain executable code for DBMs while retaining a flexible “interface” for applications. To be able to represent the full set of clock constraints with DBMs, we add an imaginary clock $\mathbf{0}$, which shall be assigned to 0 in every valuation. Zero column and row will always be reserved for $\mathbf{0}$ (i.e. $\forall c. v \ c > 0$). If necessary, we assume that v is an injection or surjection for indices less or equal to n . Informally, the zone $[M]_{v, n}$ represented by a DBM M is defined as

$$\begin{aligned} \{u \mid \forall c_1, c_2, d. v \ c_1, v \ c_2 \leq n \longrightarrow \\ (M \ (v \ c_1) \ (v \ c_2) = \text{Lt } d \longrightarrow u \ c_1 - u \ c_2 < d) \\ \wedge (M \ (v \ c_1) \ (v \ c_2) = \text{Le } d \longrightarrow u \ c_1 - u \ c_2 \leq d)\} \end{aligned}$$

assuming that $v \ \mathbf{0} = 0$.

Example 1.

$$\begin{array}{ccc}
\mathbf{0} & c_1 & c_2 \\
\mathbf{0} \left(\begin{array}{ccc} \infty & Lt(-3) & Le 0 \\ \infty & \infty & \infty \\ Le 4 & \infty & \infty \end{array} \right) & \mathbf{0} \left(\begin{array}{ccc} Le 0 & Lt(-3) & Le 0 \\ \infty & Le 0 & \infty \\ Le 4 & Lt 1 & Le 0 \end{array} \right) & \mathbf{0} \left(\begin{array}{ccc} \infty & Le 0 & Le 0 \\ \infty & \infty & Lt(-3) \\ \infty & Le 3 & Le 0 \end{array} \right)
\end{array}$$

The left two DBMs both represent the zone described by the constraint $c_1 > 3 \wedge c_2 \leq 4$, while the DBM on the right represents the empty zone.¹

To simplify the subsequent discussion, we will set $'c = nat, v = id$ and assume that the set of clocks of the automaton in question is $\{1..n\}$. We define an ordering relation \prec on $'t DBMEntry$ by means of

$$\frac{a < b}{Le a \prec Le b} \quad \frac{a < b}{Le a \prec Lt b} \quad \frac{a < b}{Lt a \prec Lt b} \quad \frac{a \leq b}{Lt a \prec Le b} \quad \frac{}{Lt - \prec \infty} \quad \frac{}{Le - \prec \infty}$$

and extend it to \preceq in the obvious way. Observe that \prec and \preceq are total orders. Additionally, we get the following important ordering property of DBMs (by nearly automatic proof):

Lemma 1. $\forall i j. i \leq n \longrightarrow j \leq n \longrightarrow M i j \preceq M' i j \implies [M]_{v,n} \subseteq [M']_{v,n}$

We can interpret DBMs as a graph with clocks as vertices and difference constraints as edges between them. To give a concrete meaning to this interpretation, we first define addition on DBM entries: $a \boxplus \infty = \infty; \infty \boxplus b = \infty$; and $(\sim_1 x) \boxplus (\sim_2 y) = \sim' (x + y)$ where $\sim' = Le$ if $\sim_1 = \sim_2 = Le$ and $\sim' = Lt$ if otherwise. Now the length of a path (of DBM indices representing clocks) defined by²

$$len M s t [] = M s t \text{ and } len M s t (w \cdot ws) = M s w \boxplus len M w t ws$$

gives the key to reasoning about this interpretation: for any $u \in [M]_{v,n}$ and i, j, xs with $set(i \cdot j \cdot xs) \subseteq \{0..n\}$,³ we get $Lt(u i - u j) \prec len M i j xs$ via induction on xs . Setting $i = j$, we can immediately conclude that DBMs with negative cycles are always empty. In the following we will make use of a predicate expressing that a DBM does not contain any negative cycles which only consist of vertices less or equal to k for some k :

$$\begin{array}{l}
\text{cycle-free-up-to } M k n \equiv \\
\forall i xs. i \leq n \wedge set xs \subseteq \{0..k\} \longrightarrow Le 0 \preceq len M i i xs
\end{array}$$

We write *cycle-free* $M n$ if *cycle-free-up-to* $M n n$.

¹ We assume a default clock numbering, mapping c_i to index i , for our examples.

² $[]$ denotes the empty list and $x \cdot xs$ is a list constructed from head x and tail xs .

³ $set xs$ is the set of elements contained in xs .

3.2 Operations

We define the necessary operations on DBMs to obtain a basic forward analysis algorithm for reachability.

Floyd-Warshall algorithm. From Example 1 we can see that to be able to tell if two DBMs represent the same zone, we first need to put them into some *canonical* form. Formally, this canonical form is characterized by the following property:

$$\text{canonical } M \ n \equiv \forall i \ j \ k. i \leq n \wedge j \leq n \wedge k \leq n \longrightarrow M \ i \ k \preceq M \ i \ j \boxplus M \ j \ k$$

The key property of non-empty canonical DBMs is that we can find a valuation $u \in [M]_{v,n}$ with $u \ i - u \ j = d$ for any d between $-M \ j \ i$ and $M \ i \ j$, or equivalently:

Lemma 2. *Assume $Le \ d \preceq M \ i \ j$, $Le \ (-d) \preceq M \ j \ i$ for M with cycle-free $M \ n$, canonical $M \ n$, and $i, j \leq n$ with $i \neq j$. We define M' by setting $M' \ i \ j = Le \ d$ and $M' \ j \ i = Le \ (-d)$ and $M' \ i' \ j' = M \ i' \ j'$ for all (i', j') where $(i', j') \neq (i, j), (j, i)$. Then $[M']_{v,n} \subseteq [M]_{v,n}$ and cycle-free $M' \ n$.*

Proof. From Lemma 1, we get $[M']_{v,n} \subseteq [M]_{v,n}$. It remains to show that M' does not contain a negative cycle. Suppose there is one. Then we can also find a *smallest* negative cycle, which, without loss of generality, is of the form $len \ M' \ i \ i \ (j \cdot xs) \prec Le \ 0$ for some xs where $i, j \notin set \ xs$. This proof step is rather intricate in Isabelle. We use a function that explicitly computes smallest negative cycles. An inductive argument yields a result that allows us to rotate cycles. Now, we get $Le \ d \boxplus len \ M' \ j \ i \ xs \prec Le \ 0$. We have $xs \neq []$ as this would directly give us the contradiction $Le \ d \boxplus Le \ (-d) \prec Le \ 0$. This means that $Le \ d \boxplus len \ M \ j \ i \ xs \prec Le \ 0$ (by induction on xs), and because M is canonical, $M \ j \ i \prec Le \ (-d)$, which is a contradiction to our assumption. \square

An important consequence is that any canonical DBM without a negative diagonal has at least one valuation, which we can construct by repeatedly applying the theorem. Observe that this also implies that a DBM in canonical form is empty iff there is a negative entry on its diagonal.

The canonical form can be computed by the Floyd-Warshall algorithm for the all-pairs shortest paths problem. A simple HOL formulation of the algorithm is

$$fw\text{-upd } M \ k \ i \ j \equiv M \ (i := (M \ i) \ (j := \min (M \ i \ j) (M \ i \ k \boxplus M \ k \ j)))$$

$$\begin{aligned} fw \ M \ n \ 0 \ 0 \ 0 &= fw\text{-upd } M \ 0 \ 0 \ 0 \\ fw \ M \ n \ (Suc \ k) \ 0 \ 0 &= fw\text{-upd } (fw \ M \ n \ k \ n \ n) \ (Suc \ k) \ 0 \ 0 \\ fw \ M \ n \ k \ (Suc \ i) \ 0 &= fw\text{-upd } (fw \ M \ n \ k \ i \ n) \ k \ (Suc \ i) \ 0 \\ fw \ M \ n \ k \ i \ (Suc \ j) &= fw\text{-upd } (fw \ M \ n \ k \ i \ j) \ k \ i \ (Suc \ j) \end{aligned}$$

where $f(a := b) \equiv \lambda x. \text{ if } x = a \text{ then } b \text{ else } f \ x$. We abbreviate $fw \ M \ n \ n \ n \ n$ as $FW \ M \ n$. To prove that this algorithm computes the tightest difference constraint for all pairs of clocks, we claim:

Theorem 1

cycle-free-up-to $M k n \wedge i' \leq i \wedge j' \leq j \wedge i \leq n \wedge j \leq n \wedge k \leq n \implies$
 $\text{Min } \{\text{len } M i' j' \text{ xs} \mid \text{set xs} \subseteq \{0..k\} \wedge i' \notin \text{set xs} \wedge j' \notin \text{set xs} \wedge \text{distinct xs}\}$
 $= \text{fw } M n k i j i' j'$

The proof is a nested induction, which follows the program structure and uses a standard argument. The theorem implies that *FW* computes a canonical form:

Corollary 1. *cycle-free* $M n \implies \text{canonical } (FW M n) n$

The Floyd-Warshall algorithm also *detects* negative cycles by computing a negative diagonal entry. The key observation is that a matrix of this kind either has a negative diagonal entry to start with, or there is a maximal $k < n$ with *cycle-free-up-to* $M k n$. The latter means that the algorithm computes a negative diagonal entry in iteration $k + 1$. In either case the negative diagonal entry will be preserved by monotonicity of the algorithm. This yields an emptiness check for DBMs.

Intersection. The intersection of two DBMs is trivial to compute. It is simply the point-wise minimum: $\text{And } A B \equiv \lambda i j. \min (A i j) (B i j)$. The operation is correct in the following sense: $[A]_{v,n} \cap [B]_{v,n} = [\text{And } A B]_{v,n}$. The \subseteq -direction can directly be proved by Isabelle's simplifier, while \supseteq requires a rather lengthy proof by cases.

Reset. We need an operator *reset* such that $u c = d$ for all $u \in [\text{reset } M n c d]_{v,n}$. Thus we define $(\text{reset } M n c d) c 0 = \text{Le } d$ and $(\text{reset } M n c d) 0 c = \text{Le } (-d)$. By doing so, all difference constraints involving c are invalidated. Therefore we set the corresponding DBM entries to ∞ . However, this alone does not yield a correct operation. Consider clocks c_1, c_2 and c_3 and a DBM represented by the clock constraint $c_1 \geq c_2 + 1 \wedge c_1 \leq c_3$. By setting c_1 to 0, we will lose all constraints on c_2 and c_3 . This means that the resulting zone will contain a valuation u with $u c_1 = u c_2 = u c_3 = 0$. There is clearly no way to set c_1 back to a different value such the resulting valuation would satisfy the original constraint. The way to resolve this issue is to encode the information we had about c_2 and c_3 in the original constraint (or DBM) also in the new DBM. This is, we derive $c_2 - c_3 \leq -1$. Concretely, we calculate $(\text{reset } M n i d) j k = \min (M j i + M i k) (M j k)$ for all $j, k \leq n$. Note that this computation does nothing if M is already in canonical form, allowing a simpler implementation.

For a list of clocks cs and a list of time stamps $ts(|cs| = |ts|)$, *set-clocks* $cs ts u$ is the valuation for which $(\text{set-clocks } cs ts u) cs_i = ts_i$ and the value of $u c$ is unchanged for all other clocks $c \notin \text{set } cs$. We lift *reset* to *reset* many clocks at once by simply folding it over the list of clocks. We proved correctness of the lifted operation (*reset'*):

(Sound) $(\forall c \in \text{set } cs. 0 < c \wedge c \leq n) \wedge u \in [\text{reset}' M n cs v d]_{v,n}$
 $\implies \exists ts. \text{set-clocks } cs ts u \in [M]_{v,n}$
(Complete) $(\forall c \in \text{set } cs. 0 < c \wedge c \leq n) \wedge u \in [M]_{v,n}$
 $\implies [cs \rightarrow d]u \in [\text{reset}' M n cs v d]_{v,n}$

The proofs for these results are among the most complex ones in the whole formalization. The reason is that manual case analyses have to be combined with (linear) arithmetic reasoning, which is hard to automate in Isabelle.

Delay. We need an operation to compute time lapse, i.e. $([M]_{v,n})^\dagger$. For canonical DBMs, this simply amounts to setting $M\ i\ 0 = \infty$ for all $i \leq n$. In the general case, intuitively we can lose information about the difference of two clocks that was recorded between the upper bound of one of them and the lower bound of the other. Accounting for this, we arrive at the following general operation:

$$\begin{aligned} up\ M &\equiv \\ \lambda i\ j. &\text{ if } 0 < i \text{ then if } j = 0 \text{ then } \infty \text{ else } \min (M\ i\ 0 \boxplus M\ 0\ j) (M\ i\ j) \text{ else } M\ i\ j \end{aligned}$$

Correctness can be obtained similarly to the reset operation.

Abstraction. It is easy to turn an atomic clock constraint into a DBM that represents the same zone. For instance, the zone $\{u \mid u \vdash EQ\ c\ d\}$ is represented by a DBM M where $M\ c\ 0 = Le\ d$ and $M\ 0\ c = Le\ (-d)$, and all other entries are unbounded. Using the already defined intersection operation for constructor *AND*, a function *abstr*, which records entries in this manner while working recursively through a constraint, turns constraints into a DBM-equivalent. Again, we proved correctness (where *collect-clks cc* is the set of all clocks appearing in constraint *cc*):

$$\forall c \in \text{collect-clks } cc. 0 < c \wedge c \leq n \implies [\text{abstr } cc\ (\lambda i\ j. \infty)\ v]_{v,n} = \{u \mid u \vdash cc\}$$

3.3 DBM Operational Semantics

In the last section we have elaborated the adequacy of our DBM-equivalents for all zone operations, allowing us to compute the zone semantics with the help of DBMs. Indeed we can define a new operational semantics based on DBMs:

$$\frac{M_i = \text{abstr } (\text{inv-of } A\ l)\ (\lambda i\ j. \infty)\ v}{A \vdash \langle l, M \rangle \rightsquigarrow_{v,n} \langle l, \text{And } (up\ (And\ M\ M_i))\ M_i \rangle}$$

$$\frac{A \vdash l \xrightarrow{g,a,r} l' \wedge M_i = \text{abstr } (\text{inv-of } A\ l')\ (\lambda i\ j. \infty)\ v}{A \vdash \langle l, M \rangle \rightsquigarrow_{v,n} \langle l', \text{And } (\text{reset}'\ (And\ M\ (\text{abstr } g\ (\lambda i\ j. \infty)\ v))\ n\ r\ v\ 0)\ M_i \rangle}$$

Using the correctness results for the DBM operations, it is straightforward to show that this semantics is equivalent to the zone semantics:

$$\begin{aligned} &A \vdash \langle l, [M]_{v,n} \rangle \rightsquigarrow^* \langle l', Z \rangle \\ \iff &\exists M'. A \vdash \langle l, M \rangle \rightsquigarrow_{v,n}^* \langle l', M' \rangle \wedge Z = [M']_{v,n} \end{aligned}$$

However, we are not done yet: while we can practically compute the semantics of timed automata, the search space could still be infinite. The rest of the paper is concerned with overcoming this problem.

4 From Classic Decidability to a Correct Approximation

4.1 Regions

In their seminal paper, Alur and Dill showed decidability of the emptiness problem for timed automata by giving an adequate finite partitioning of the set of valuations into what they call *regions*. In this section, we will present our formalization of this result and then show how to apply it to obtain a *finite* operational semantics of zones. We use Bouyer’s definition of regions as, for one it is more formal and thus easier to formalize, and secondly we will have to use a modified version of it later on.

From now on we will work in a parametric theory (called *locale* in Isabelle), which fixes X as the set of clocks of the automaton. Moreover, a *clock ceiling* k will define an upper bound $k\ c$ for the “relevant” range of any clock $c \in X$ – this ought to correspond to the *maximal* constant appearing for c in any constraint of the timed automaton, e.g., $k\ c_1 = 3$ and $k\ c_2 = 2$ for the automaton of Fig. 1. This is, if $\sim\ c\ m$ is a constraint of the automaton, we postulate that $m \leq k\ c$, $c \in X$, and that m is a natural number.

A single clock value will always fall into one of three types of intervals from

datatype *intv* = *Const nat* | *Intv nat* | *Greater nat*

where the set of values they contain is given by the following rules:

$$\frac{u\ x = d}{\text{intv-elem } x\ u\ (\text{Const } d)} \quad \frac{d < u\ x \wedge u\ x < d + 1}{\text{intv-elem } x\ u\ (\text{Intv } d)}$$

$$\frac{d < u\ x}{\text{intv-elem } x\ u\ (\text{Greater } d)}$$

Let $I :: 'c \Rightarrow \text{intv}$ be assigning intervals to clocks and r be a finite total pre-order over $X_0 \equiv \{x \in X \mid \exists d. I\ x = \text{Intv } d\}$. Then we define the corresponding region *region* $X\ I\ r$ as the set for which⁴

$$u \in \text{region } X\ I\ r \text{ iff } \forall x \in X. 0 \leq u\ x \wedge \text{intv-elem } x\ u\ (I\ x)$$

$$\text{and } \forall x \in X_0. \forall y \in X_0. (x, y) \in r \iff \text{frac } (u\ x) \leq \text{frac } (u\ y)$$

We will fix a set of regions $\mathcal{R}_\alpha \equiv \{\text{region } X\ I\ r \mid \text{valid-region } X\ k\ I\ r\}$ where *valid-region* $X\ k\ I\ r$ holds if X is finite, r is a total preorder on X_0 , and $d \leq k\ x$ if $I\ x = \text{Const } d$, $d < k\ x$ if $I\ x = \text{Intv } d$, and $k\ x = d$ if $I\ x = \text{Greater } d$ for all $x \in X$. Observe that this definition remedies the potential overlap of intervals that the definition of *intv-elem* would admit.

It is clear from Fig. 1, and relatively straightforward to prove in Isabelle/HOL, that \mathcal{R}_α is a finite partitioning of

$$V \equiv \{u \mid \forall x \in X. 0 \leq u\ x\},$$

⁴ *frac* r denotes the fractional part of any real number r .

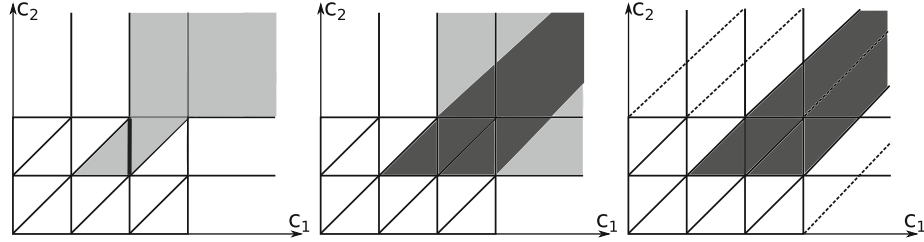


Fig. 2. (1) A region and its time successors in \mathcal{R}_α , (2) the α -closure of a zone, and (3) the β -approximation of a zone for $X = \{c_1, c_2\}$ with $k_{c_1} = 3$ and $k_{c_2} = 2$.

the set of all positive valuations. What is not so obvious (and not mentioned by Bouyer) but a useful property to work with, is that any valid region is also non-empty. The crux of this proof is to observe that X_0 can be ordered in equivalence classes according to r such that a valuation u can be chosen for which $\text{frac}(u x) \leq \text{frac}(u y)$ iff $(x, y) \in r$. This ordering property of finite total preorders is non-trivial to formalize and makes this step rather technical.

4.2 Decidability with Regions

How are regions and timed automata connected? We will present three key properties that connect regions to time lapse, clock resets, and clock constraints, respectively, allowing us to implement timed automata with the help of regions. Let $[u]_{\mathcal{R}_\alpha} \in \mathcal{R}_\alpha$ be the unique region containing u . We call $[u \oplus t]_{\mathcal{R}_\alpha}$ a *time successor* of $[u]_{\mathcal{R}_\alpha}$ for $t \geq 0$ and denote by $\text{Succ } \mathcal{R}_\alpha R$ the set of all such time successors of all $u \in R$ (cf. Fig. 2.1). Now the three key properties are in order of decreasing difficulty:

(Set of regions) $R \in \mathcal{R}_\alpha \wedge u \in R \wedge R' \in \text{Succ } \mathcal{R}_\alpha R$
 $\implies \exists t \geq 0. [u \oplus t]_{\mathcal{R}_\alpha} = R'$

(Compatibility with resets) $R \in \mathcal{R}_\alpha \wedge u \in R \wedge 0 \leq d \wedge d \leq k x \wedge x \in X$
 $\implies [u(x := d)]_{\mathcal{R}_\alpha} = \{u(x := d) \mid u \in R\}$

(Compatibility with constraints)

$R \in \mathcal{R}_\alpha \wedge \forall (x, m) \in \text{collect-clock-pairs } cc. m \leq k x \wedge x \in X \wedge m \in \mathbb{N}$
 $\implies R \subseteq \{u \mid u \vdash cc\} \vee \{u \mid u \vdash cc\} \cap R = \emptyset$

Proof. We concentrate on the set of regions property as it has the most interesting formalization. Our proof combines elements of the “classic” result as presented e.g., in [9], and Bouyer’s approach. Let $R = \text{region } X I r \in \mathcal{R}_\alpha$ for some I, r , let $R' = [v \oplus t]_{\mathcal{R}_\alpha}$, and assume $u, v \in R$ and $t \geq 0$. If $I x = \text{Greater}(k x)$ for all $x \in X$ (“upper-right region”), we have $\text{Succ } \mathcal{R}_\alpha R = \{R\} = \{R'\}$ and the proposition is obvious.

Otherwise observe that there exists a single *closest* successor R_{succ} of R (depicted as the thick, dark gray line in Fig. 2.1). We refer to Bouyer for a

formal construction of this successor. We can show the characteristic property of this closest successor:

$$\forall u \in R. \forall t \geq 0. (u \oplus t) \notin R \longrightarrow (\exists t' \leq t. (u \oplus t') \in R_{succ} \wedge t' \geq 0)$$

At this point Bouyer states that the proposition follows by “immediate induction”. However, regarding formalization, this induction is not quite immediate. For instance, we attempted induction on the set of successors. This necessitates a proof that this set is monotone, which we did not find ourselves able to prove without asserting the very property we were about to prove. Instead, we split the argument in two: one for the case where $t < 1$ and the other for the case where t is an integer. For the first case, consider the “critical” set $C = \{x \in X \mid \exists d. I x = Intv\ d \wedge d + 1 \leq u x + t\}$, the set of clocks for which $u \oplus t$ is shifted beyond R 's interval boundaries. Observe that for the closest successor, the critical set is either the same (if $\{x \in X \mid \exists d. I x = Const\ d\} \neq \emptyset$) or a strict subset (if otherwise). Thus the proposition follows by induction on the cardinality of C . The case where t is an integer follows by direct proof over the structure of regions. Shifting u first by $frac\ t$ and then by $\lfloor t \rfloor$, we arrive at the proposition. \square

This allows us to define a region-based operational semantics for timed automata:

$$\frac{R \in \mathcal{R}_\alpha \wedge R' \in Succ\ \mathcal{R}_\alpha\ R \wedge R \cup R' \subseteq \{u \mid u \vdash inv\text{-of}\ A\ l\}}{A, \mathcal{R}_\alpha \vdash \langle l, R \rangle \rightsquigarrow \langle l, R' \rangle}$$

$$\frac{A \vdash l \xrightarrow{g, a, r} l' \wedge R \in \mathcal{R}_\alpha}{A, \mathcal{R}_\alpha \vdash \langle l, R \rangle \rightsquigarrow \langle l', \{[r \rightarrow 0]u \mid u \in R \wedge u \vdash g\} \cap \{u \mid u \vdash inv\text{-of}\ A\ l'\} \rangle}$$

From the aforementioned properties, we proved its adequacy w.r.t. to reachability:

$$A, \mathcal{R}_\alpha \vdash \langle l, [u]_{\mathcal{R}_\alpha} \rangle \rightsquigarrow^* \langle l', R' \rangle \wedge R' \neq \emptyset$$

$$\longleftrightarrow \exists u'. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle \wedge [u']_{\mathcal{R}_\alpha} = R'$$

Note that it is quite natural that this property is weaker compared to previous ones: (sets of) regions only approximate zones and thus can contain valuations that were never reachable in the concrete semantics.

4.3 Approximating Zone Semantics with Regions

From the pure decidability result on regions, we now move back towards zones by *approximating* zones with the smallest set of regions that covers them. Formally we define the α -closure of a zone Z : $Closure_\alpha\ Z = \bigcup \{R \in \mathcal{R} \mid R \cap Z \neq \emptyset\}$. Observe that this set need not be convex (cf. Fig. 2.2). We use the α -closure to define an operational semantics on zones that approximates a zone with its α -closure at the end of each step:

$$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \implies A \vdash \langle l, Z \rangle \rightsquigarrow_\alpha \langle l', Closure_\alpha\ Z' \rangle$$

Bouyer would now go and prove from the region properties that the α -closure can be “pushed through” each step:

$$\begin{aligned} Z \subseteq V \wedge A \vdash \langle l, \text{Closure}_\alpha Z \rangle &\rightsquigarrow \langle l', Z' \rangle \\ \implies \exists Z''. A \vdash \langle l, Z \rangle &\rightsquigarrow_\alpha \langle l', Z'' \rangle \wedge Z' \subseteq Z'' \end{aligned}$$

However, we did not find this property strong enough to prove soundness of $\rightsquigarrow_\alpha^*$:

$$\begin{aligned} A \vdash \langle l, Z \rangle &\rightsquigarrow_\alpha^* \langle l', Z' \rangle \wedge Z \subseteq V \\ \implies \exists Z''. A \vdash \langle l, Z \rangle &\rightsquigarrow^* \langle l', Z'' \rangle \wedge \text{Closure}_\alpha Z' \subseteq \text{Closure}_\alpha Z'' \wedge Z'' \subseteq Z' \end{aligned}$$

Note that this property is really what one wants to have since $\text{Closure}_\alpha Z = \emptyset$ iff $Z = \emptyset$ (assuming that $Z \subseteq V$). We conceived that instead it is sufficient to prove monotonicity of the α -closure w.r.t. to steps in the zone semantics:

$$\begin{aligned} A \vdash \langle l, Z \rangle &\rightsquigarrow \langle l', Z' \rangle \wedge \text{Closure}_\alpha Z = \text{Closure}_\alpha W \wedge W \subseteq Z \wedge Z \subseteq V \\ \implies \exists W'. A \vdash \langle l, W \rangle &\rightsquigarrow \langle l', W' \rangle \wedge \text{Closure}_\alpha Z' = \text{Closure}_\alpha W' \wedge W' \subseteq Z' \end{aligned}$$

Combining this with the fact that α -closure is an involution, we proved soundness by induction over $\rightsquigarrow_\alpha^*$. Completeness follows easily from monotonicity of \rightsquigarrow^* :

$$\begin{aligned} A \vdash \langle l, Z \rangle &\rightsquigarrow^* \langle l', Z' \rangle \wedge Z \subseteq V \wedge Z' \neq \emptyset \\ \implies \exists Z''. A \vdash \langle l, Z \rangle &\rightsquigarrow_\alpha^* \langle l', Z'' \rangle \wedge Z' \subseteq Z'' \end{aligned}$$

While these results are nice from a theoretical standpoint, it is not easier to compute the α -closure than to directly implement timed automata with the region construction presented in the last section. Therefore, the next section will present Bouyer’s main insight – that these results can be used to show the correctness of an easily computable approximation operation.

5 Normalization

Consider Fig. 2.3. In addition to \mathcal{R}_α (solid lines), the figure shows a refinement to what we will call \mathcal{R}_β (dashed lines). Observe that the smallest set of regions covering the zone painted in dark gray (i.e. its β -closure) is *convex*, whereas its α -closure is not (cf. Fig. 2.2). The idea is to use this β -closure to obtain an effectively computable convex approximation for zones represented by DBMs – DBMs always represent a convex zone and are always covered by a convex β -closure – while inheriting the correctness result from the α -closure as we only refine things.

5.1 β – approximation

Due to a lack of space, we do not present our construction of \mathcal{R}_β and only say that it can be adopted from \mathcal{R}_α with some modifications. Note that we do not need to transfer the (rather intricate) properties connecting \mathcal{R}_α with

transitions of timed automata since we will infer correctness directly from the original construction.

We now want to formalize the notion of a convex approximation of zones with regions from \mathcal{R}_β . We capture the notion of convexity directly with DBMs. From Example 1, we can see that the types of regions in \mathcal{R}_β also induce a specific format for our DBMs: for a DBM entry $M\ i\ j$, we do not need constants outside of $[-k\ i; k\ j]$ because this is precisely the range to which our regions bound the corresponding values (analogously for constraints involving $\mathbf{0}$). Thus we use the following notion of *normalized* DBMs:

$$\begin{aligned} \text{normalized } M &\equiv \\ (\forall i\ j. 0 < i \wedge i \leq n \wedge 0 < j \wedge j \leq n \wedge M\ i\ j \neq \infty \longrightarrow \\ &\quad Lt(-k\ j) \preceq M\ i\ j \wedge M\ i\ j \preceq Le(k\ i)) \wedge \\ (\forall i \leq n. 0 < i \longrightarrow (M\ i\ 0 \preceq Le(k\ i) \vee M\ i\ 0 = \infty) \wedge Lt(-k\ i) \preceq M\ 0\ i) \end{aligned}$$

Furthermore, all constraints only need to use integer constants, which we denote by *dbm-int* M . Building from these ideas, we define for any zone Z :

$$\begin{aligned} \text{Approx}_\beta Z &\equiv \bigcap \{[M]_{v,n} \mid \\ \exists U \subseteq \mathcal{R}_\beta. [M]_{v,n} = \bigcup U \wedge Z \subseteq [M]_{v,n} \wedge \text{dbm-int } M\ n \wedge \text{normalized } M\} \end{aligned}$$

5.2 Connecting Approx_β and Closure_α

We already argued that it is possible to inherit correctness from Closure_α because we only refine regions. Precisely, Bouyer proposed that for any convex zone Z (i.e. $Z = [M]_{v,n}$ for some DBM M), we have $\text{Approx}_\beta Z \subseteq \text{Closure}_\alpha Z$, or equivalently:

Theorem 2. $R \in \mathcal{R}_\alpha \wedge Z \subseteq V \wedge R \cap Z = \emptyset \wedge Z = [M]_{v,n} \wedge \text{dbm-int } M\ n \implies R \cap \text{Approx}_\beta Z = \emptyset$

The formalization of Bouyer's proof for this proposition is one of the most complicated parts of our development. As the prose proof is already sufficiently complicated, we abstain from presenting our formalization of this result.

Analogously to \rightsquigarrow_α , we define an approximating semantics \rightsquigarrow_β using Approx_β . The main fact we can derive from the Theorem 2 is that \rightsquigarrow_α is an approximation of \rightsquigarrow_β :

Lemma 3

$$\begin{aligned} A \vdash \langle l, [M]_{v,n} \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge \text{dbm-int } M\ n \wedge [M]_{v,n} \subseteq W \wedge W \subseteq V \\ \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_\alpha \langle l', W' \rangle \wedge Z' \subseteq W' \end{aligned}$$

Using this result and some additional work, we could infer soundness and completeness of \rightsquigarrow_β^* from the corresponding results for $\rightsquigarrow_\alpha^*$.

5.3 Computing $Approx_\beta$

So far, we have shown how to obtain a correct approximation operation from \mathcal{R}_β , which only produces convex sets. The huge gain from that is that this approximation can also be easily computed by *normalizing* DBMs:

$$\begin{aligned} norm\ M\ k\ n &\equiv \\ \lambda i\ j. \text{ let } ub &= \text{if } 0 < i \text{ then } k\ i \text{ else } 0; lb = \text{if } 0 < j \text{ then } -\ k\ j \text{ else } 0 \\ &\text{ in if } i \leq n \wedge j \leq n \text{ then } norm\text{-lower } (norm\text{-upper } (M\ i\ j)\ ub)\ lb \\ &\text{ else } M\ i\ j \\ norm\text{-upper } e\ t &= (\text{if } Le\ t < e \text{ then } \infty \text{ else } e) \\ norm\text{-lower } e\ t &= (\text{if } e < Lt\ t \text{ then } Lt\ t \text{ else } e) \end{aligned}$$

Lemma 4. $canonical\ M\ n \wedge [M]_{v,n} \subseteq V \wedge dbm\text{-int } M\ n \implies$
 $Approx_\beta ([M]_{v,n}) = [norm\ M\ k\ n]_{v,n}$

Again, we abstain from providing a full presentation of our formalization and only mention that the main ideas are: (1) to observe that normalized integral DBMs can always be represented by an equivalent subset of \mathcal{R}_β , and (2) that $norm\ M\ k\ n$ computes a minimal normalized DBM.

5.4 A Final Semantics

We have assembled all the ingredients to define a semantics for timed automata which captures the essence of what DBM-based model checkers compute:

$$A \vdash \langle l, D \rangle \rightsquigarrow_{v,n} \langle l', D' \rangle \implies A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}} \langle l', norm\ (FW\ D'\ n)\ k\ n \rangle$$

Combining the fact that β -approximation is computable and the correctness properties of \rightsquigarrow_β^* and \rightsquigarrow^* , we have achieved our main result: a timed automaton can reach a certain location l' iff we can compute a valid run (using the DBM operations and normalization) that ends in l' .

Theorem 3. $Z = [M]_{v,n} \wedge Z \subseteq V \wedge dbm\text{-int } M\ n \implies$
 $(\exists u \in Z. \exists u'. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle)$
 $\iff (\exists M'. A \vdash \langle l, M \rangle \rightsquigarrow_{\mathcal{N}}^* \langle l', M' \rangle \wedge [M']_{v,n} \neq \emptyset)$

6 Conclusion

We have presented a formalization that, beginning with basic definitions and classic results, closes the loop to show correctness of the basic DBM-based algorithms that are used in forward analysis of timed automata. However, we have not yet harvested potential practical fruits of this development. A self-evident goal is to obtain an executable version for the algorithms above. By combination with a verified version of e.g., depth-first search, this could already yield a verified tool for deciding language emptiness of timed automata, which could in turn be extended to a fully verified model checker. In another direction of development, the author has already started to reuse the presented formalization to formalize first results about decidability of probabilistic timed automata.

Acknowledgement. I would like to thank Tobias Nipkow and the anonymous reviewers for their helpful comments on earlier versions of this paper.

References

1. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**, 183–235 (1994)
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pp. 592–601 (1993)
4. Bengtsson, J.E., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets*. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
5. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
6. Bouyer, P.: Forward analysis of updatable timed automata. *Form. Methods Syst. Des.* **24**(3), 281–320 (2004)
7. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Are timed automata updatable? In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 464–479. Springer, Heidelberg (2000)
8. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLS 2001: Supplemental Proceedings (2001). <http://www.informatics.ed.ac.uk/publications/report/0046.html>
9. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (2001)
10. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
11. Garnacho, M., Bodeveix, J.P., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 106–120. Springer, Heidelberg (2013)
12. Henzinger, T.A., Ho, P.-H., Wong-toi, H.: Hytech: a model checker for hybrid systems. *Softw. Tools Technol. Transf.* **1**(1), 460–463 (1997)
13. Larsen, G.K., Pettersson, P., Yi, W.: Uppaal in a nutshell. *Softw. Tools Technol. Transf.* **1**(1), 134–152 (1997)
14. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Kobayashi, N., Babu, C.S. (eds.) TACS 2001. LNCS, vol. 2215, pp. 298–315. Springer, Heidelberg (2001)
15. Wimmer, S.: Timed automata. Archive of Formal Proofs, March 2016. http://isa-afp.org/entries/Timed_Automata.shtml, Formal proof development
16. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: Proceedings of the IASTED International Conference on Software Engineering, pp. 107–112 (2006)
17. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: Proceedings of SNPD 2009, pp. 555–560 (2009)
18. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Proceedings of Formal Description Techniques VII, pp. 243–258 (1994)
19. Yovine, S.: KRONOS: a verification tool for real-time systems. *Softw. Tools Technol. Transf.* **1**(1), 123–133 (1997)

B MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper)

Original publication. This chapter was originally published as a short paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer and Johannes Hölzl. “MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper).” In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 597–603. DOI: 10.1007/978-3-319-94821-8_35

Synopsis. The paper presents a formalization of probabilistic timed automata (PTA) in Isabelle/HOL. The formalization tries to follow the formula “MDP + TA = PTA” as far as possible: the work starts from existing formalizations of Markov decision processes (MDP) and timed automata (TA) and combines them modularly. The fundamental result for probabilistic timed automata is formalized: the region construction that is known from timed automata carries over to the probabilistic setting. In particular, it is proved that minimum and maximum reachability probabilities can be computed via a reduction to MDP model checking. The case where unrealizable Zeno behaviors shall be excluded from the calculation of minimum and maximum reachability properties is also studied.

Contribution. I have contributed most of the formalization on the fundamentals of PTA and the region construction on PTA, and the treatment of Zeno behaviors. We have contributed the ideas for the bisimulation argument to equal parts. Johannes Hölzl has contributed most proofs about measurability and formalized the central bisimulation theorem in Isabelle/HOL. I have contributed most of the text in the paper aside from the introductions to Markov chains and MDPs (first half of Section 2).

Copyright notice. On the following pages the full article is reprinted by permission from Springer Nature Customer Service Centre GmbH. The original publication can be found under the DOI cited above.



MDP + TA = PTA: Probabilistic Timed Automata, Formalized (Short Paper)

Simon Wimmer¹ and Johannes Hölzl²

¹ TU München, Munich, Germany

wimmers@in.tum.de

² VU Amsterdam, Amsterdam, Netherlands

jh1890@vu.nl

Abstract. We present a formalization of probabilistic timed automata (PTA) in which we try to follow the formula “MDP + TA = PTA” as far as possible: our work starts from existing formalizations of Markov decision processes (MDP) and timed automata (TA) and combines them modularly. We prove the fundamental result for probabilistic timed automata: the region construction that is known from timed automata carries over to the probabilistic setting. In particular, this allows us to prove that minimum and maximum reachability probabilities can be computed via a reduction to MDP model checking, including the case where one wants to disregard unrealizable behavior.

1 Introduction

Timed automata (TA) [1] are a widely used formalism for modeling nondeterministic real-time systems. Markov decision processes (MDPs) with discrete time are popular for modeling probabilistic systems with nondeterminism. Probabilistic timed automata (PTA) fuse the concepts of TA and MDPs and allow probabilistic modeling of nondeterministic real-time systems. PRISM [3] implements model checking functionality for MDPs and PTA and has successfully been applied to a number of case studies [6].

We have previously formalized MDPs [2] and TA [8] in Isabelle/HOL. This paper presents an Isabelle/HOL formalization of PTA, which follows the formula “MDP + TA = PTA” as far as possible by combining our existing formalizations modularly. We prove the fundamental result for PTA: the region construction that is known from TA carries over to the probabilistic setting. In particular, we prove that minimum and maximum reachability probabilities (with respect to possible resolutions of nondeterminism) can be computed via a reduction to MDP model checking, including the case where one wants to disregard unrealizable behavior. This work is a necessary first step towards our long-term goal of certifying the computation results of PRISM’s backward reachability algorithm [4] for reducing PTA to MDP model checking. The formalization can be found in the Archive of Formal Proofs [9].

2 Preliminaries

Markov Chains. A probability mass function (PMF, or discrete distribution) $\mu :: \sigma \text{ pmf}$ is a function $\sigma \Rightarrow \mathbb{R}_{\geq 0}$ with countable support $\{x \mid \mu x \neq 0\}$ whose range sums to 1. Any PMF forms a monad, thus we have $(\text{map}_{\text{pmf}} f \mu) y = \mu \{x \mid f x = y\}$ and $(\text{ret}_{\text{pmf}} x) x = 1$. A *Markov chain (MC)* is represented by the transition system $K :: \sigma \Rightarrow \sigma \text{ pmf}$ (its kernel, which is commonly represented by a transition matrix $\mathbb{R}^{|\sigma| \times |\sigma|}$), mapping each state to a distribution of next states. The trace space $T_K s$ is the probability measure with the property $T_K s (x_0 \cdots x_n) = K s x_0 * \cdots * K x_{n-1} x_n$ (where $(x_0 \cdots x_n)$ is the set of state traces starting with x_0, \dots, x_n). A *probabilistic coupling* with respect to a relation R exists between two PMFs μ and μ' (written $\text{rel}_{\text{pmf}} R \mu \mu'$) if there exists a distribution ν on the product type, such that the support of ν is a subset of R and the marginal distributions of ν are $\mu = \text{map}_{\text{pmf}} \pi_1 \nu$ and $\mu' = \text{map}_{\text{pmf}} \pi_2 \nu$. Probabilistic couplings allow us to relate two Markov chains.

Markov Decision Processes. MDPs are automata allowing probabilistic and non-deterministic choice. An MDP is represented by the transition system $K :: \sigma \Rightarrow \sigma \text{ pmf set}$, where σ is the type of states, and the probability distributions over the next states of type $\sigma \text{ pmf}$ are called *actions*. Each MDP gives rise to a set of MCs, each showing one possible behaviour. We introduce, coinductively, *configurations* $\sigma \text{ cfg}$, where each $c :: \sigma \text{ cfg}$ consists of a state σ , an action $\sigma \text{ pmf}$, and a continuation $\sigma \Rightarrow \sigma \text{ cfg}$. The configurations give rise to a Markov chain $K_c :: \sigma \text{ cfg} \Rightarrow \sigma \text{ cfg pmf}$, by mapping the continuations over the actions. Each $c :: \sigma \text{ cfg}$ whose actions are closed under K and which is in state s induces a MC showing a possible behavior of the MDP K starting in s . To simplify the theory, we assume that $K x \neq \emptyset$. See [2] for details.

Timed Automata. Compared to standard finite automata, TA introduce a notion of clocks. Clocks are indexed by natural numbers and do not have any structure. A *clock valuation* u is a function of type $\mathbb{N} \Rightarrow \mathbb{R}$. Locations and transitions are guarded by *clock constraints*, which have to be fulfilled to stay in a location or to take a transition. Clock constraints are conjunctions of constraints of the form $c \sim d$ for a clock c , an integer d , and $\sim \in \{<, \leq, =, \geq, >\}$. We write $u \vdash cc$ if the clock constraint cc holds for the clock valuation u . We define a timed automaton A as a pair $(\mathcal{T}, \mathcal{I})$ where \mathcal{I} is an assignment of clock constraints to locations (also named invariants) and \mathcal{T} is a set of transitions written as $A \vdash l \xrightarrow{g, r} l'$ where l and l' are the start and successor location, g is the guard of the transition, and r is a set of clocks that will be reset to zero when the transition is taken. States of TA are pairs of a location and a clock valuation. The operational semantics define two kinds of steps:

Delay: $(l, u) \rightarrow (l, u \oplus d)$ if $d \geq 0$ and $u \oplus d \vdash \mathcal{I} l$;

Action: $(l, u) \rightarrow (l', [r \rightarrow 0]u)$ if $A \vdash l \xrightarrow{g, r} l'$, $u \vdash g$, and $[r \rightarrow 0]u \vdash \mathcal{I} l'$;

where $(u \oplus d) c = u c + d$ and $([r \rightarrow 0]u) c = \text{if } c \in r \text{ then } 0 \text{ else } u c$.

Regions. The initial decidability result [1] partitioned the set of clock valuations into a quotient of sets of clock valuations, the so-called regions, and showed that these yield a sound and complete abstraction¹. Our formalization [8] proves this fundamental result and decidability of reachability properties for ordinary TA.

3 Probabilistic Timed Automata

PTA fuse the concepts of TA and MDPs: discrete transitions are replaced by probability distributions over pairs of a set of clocks to be reset and a successor location. An example of a PTA is depicted in the left part of Fig. 1.

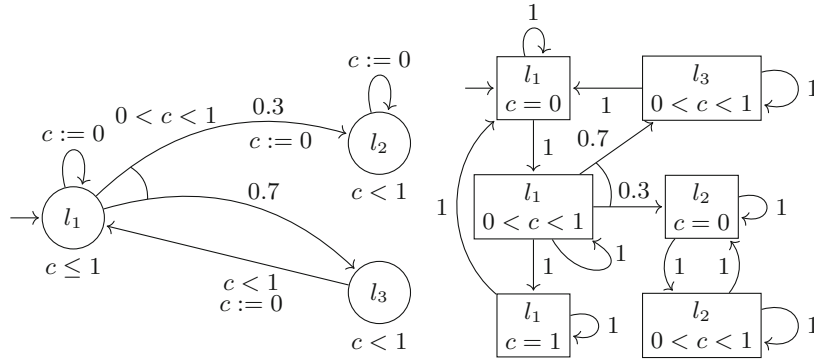


Fig. 1. Example of a PTA with one clock and its region graph

Definition. Consequently, the syntactic definition of PTA is very similar to TA. The only difference is that now transitions are of the form $A \vdash l \xrightarrow{g} \mu$ for μ of type $(\mathbb{N} \text{ set} \times \sigma) \text{ pmf}$ (the clocks to reset and σ for the type of locations).

Typical presentations define the semantics of PTA based on the notion of so-called *probabilistic timed structures*, which are just a special type of MDP. We omit this detour and directly formalize PTA in terms of MDPs. Consequently, to formalize the semantics of a PTA, we define its kernel K as the smallest set that is compatible with

$$\frac{(l, u) \in S \quad t \geq 0 \quad u \oplus t \vdash \mathcal{I} l}{ret_{pmf}(l, u \oplus t) \in K(l, u)} \text{ DELAY}$$

$$\frac{(l, u) \in S \quad A \vdash l \xrightarrow{g} \mu \quad u \vdash g}{map_{pmf}(\lambda(r, l). (l, [r \rightarrow 0]u)) \mu \in K(l, u)} \text{ ACTION}$$

where S is the set of valid states. A state (l, u) is valid if l belongs to A and if $u \vdash \mathcal{I} l$. For technical reasons there is a third rule to add self loops for non-valid

¹ We use the same notions as in [8]. Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.

states. These do not change the semantics as they are not reachable from valid states. The MDP K is uncountably infinite as S is generally infinite.

Region Graph. We want to reduce the computation of reachability probabilities for A to a computation on a finite MDP. Analogous to TA, this reduction can be obtained through the region quotient. More precisely, we will partition S into a finite set of states \mathcal{S} of the form (l, R) , for l a location of A , and R a region of A such that $\forall u \in R. u \vdash \mathcal{I} l$. With this notion, the finite MDP, coined *region graph* in [5], is defined through its kernel \mathcal{K} :

$$\frac{(l, R) \in \mathcal{S} \quad R' \in \text{Succ } R \quad \forall u \in R'. u \vdash \mathcal{I} l}{\text{ret}_{pmf}(l, R') \in \mathcal{K}(l, R)} \text{DELAY}_R$$

$$\frac{(l, R) \in \mathcal{S} \quad A \vdash l \xrightarrow{g} \mu \quad \forall u \in R. u \vdash g}{\text{map}_{pmf}(\lambda(r, l). (l, \{[r \rightarrow 0]u \mid u \in R\}) \mu) \in \mathcal{K}(l, R)} \text{ACTION}_R$$

Here $\text{Succ } R$ denotes the set of regions that can be reached from R by delaying for an arbitrary amount of time. Again, for technical reasons there is a third rule to add self loops for non-valid states. The *maximum* probability (under all valid initial configurations) to reach the state in the upper right of the region graph depicted in Fig. 1 is 0.7, while the *minimum* probability is 0.

4 Bisimulation

We relate the infinite MDP that defines the PTA with the finite region graph in a way that directly allows us to prove correctness of the reduction for maximum and minimum reachability probabilities in one go. Concretely, our agenda is to first define abstraction and representation functions that map between configurations of the infinite MDPs and the finite region graph, and vice versa. We then prove a more general bisimulation theorem on MDPs which states that the path measure assigned to related paths is the same for related configurations.

Representation and Abstraction. We will use the overloaded notations α and rep to denote the abstraction and representation functions for states, actions, and configurations. The main difficulty of our formalization effort was to define these such that one obtains the desired properties. What are these properties? Chiefly, for a valid configuration c , the probability distributions of the successors of c and αc should expose a *probabilistic coupling* w.r.t. the relation $\lambda c a. \alpha c = a$. Moreover, the abstraction of a representative should yield the original object: $\alpha(rep\ x) = x$. Finally, validity should be preserved, i.e. $\alpha(l, u) \in \mathcal{S} \leftrightarrow (l, u) \in S$.

The elementary abstraction functions are easy to define: $\alpha(l, u) = (l, [u]_{\mathcal{R}})$ for $[u]_{\mathcal{R}}$ the unique region with $u \in [u]_{\mathcal{R}}$, and $\alpha t = \text{map}_{pmf} \alpha t$ for an action t . For a configuration c , αc is defined co-recursively in terms of c : the concrete configuration c is maintained as the internal state of αc and states and actions are simply mapped with α ; the internal successor configuration however is determined by the continuation of c for the *unique* successor state s of c such that αs is the successor state of αc . The definition of rep is more involved and omitted for brevity.

Bisimulation Theorem. At the core of our argument lies the following bisimulation theorem on Markov chains:

$$T_K x A = T_L y B \text{ if } R x y \text{ and } \forall \omega \omega'. \text{rel}_{stream} R \omega \omega' \longrightarrow (\omega \in A \leftrightarrow \omega' \in B) \\ \text{and } \forall x y. R x y \longrightarrow \text{rel}_{pmf} R (K x) (L y)$$

where $T_{\{K,L\}}$ denotes the trace space induced by Markov chains K and L , respectively; x and y are states of K and L ; $\text{rel}_{stream} R$ compares two traces pointwise by R ; and A and B are sets of infinite traces of K and L . Finally, R has to be of the form $R s t = (s \in S \wedge f s = t)$ for some S and f .

For a configuration c with state s , we can instantiate this theorem by taking $K = \mathcal{K}_c$, $L = \mathcal{K}_c$, $x = s$, $y = \alpha s$, $f = \alpha$, and S as the set of valid configurations of the PTA. The coupling property of K and L follows because

$$\mathcal{K}_c (\alpha c) = \text{map}_{pmf} \alpha (\mathcal{K}_c c).$$

For this instantiation, $\text{rel}_{stream} R x y$ essentially means that y is the pointwise abstraction of x . We consider reachability properties on state traces of the form $\varphi U \psi$ (where φ and ψ can be a mixture of predicates on location and clock), so

$$A = \{\omega \mid \varphi U \psi (\text{smap } \omega)\} \text{ and } B = \{\omega \mid (\varphi \circ \text{rep}) U (\psi \circ \text{rep}) (\text{smap } \omega)\}$$

where $\text{smap } \omega$ maps traces of configurations to traces of MDP states. Consequently, the premise on A and B is easily satisfied if

$$\forall s t. \alpha s = \alpha t \longrightarrow \varphi s \leftrightarrow \varphi t \wedge \psi s \leftrightarrow \psi t,$$

which matches exactly the property that is delivered by the region construction.

5 Taking Zenoness into Account

So far, we have considered bisimulation properties between trace spaces of pairs of related configurations. Minimum and maximum reachability probabilities, however, are considered in relation to a set of configurations C . To compute these probabilities, one can consider the set of configurations C_α on the finite MDP such that $\forall c \in C. \alpha c \in C_\alpha$ and $\forall c \in C_\alpha. \text{rep } c \in C$. If C is the set of valid configurations, for instance, then C_α is easily proved to be the set of valid configurations of the region graph.

Often one wants to restrict C such that unrealizable behaviors are excluded: a configuration should not be able to keep time from passing beyond a fixed deadline. A configuration is *zeno* if it admits such behaviours. In the example, a zeno configuration could continuously take the loop transition on l_1 without letting any time pass. In [5] a *computable* description of C_α is given for the case that C is restricted to configurations that only yield non-zeno behaviors with probability 1.

The critical component of our proof for the correctness of C_α (in the sense outlined above) is that rep chooses the successor states always such that at least half of the amount of time that could possibly elapse does elapse.

Interestingly, the proof of $\forall c \in C_\alpha. \text{repc} \in C$ was much harder to formalize than the other direction, although roles seem to be flipped in the argument of [5]. For the harder direction, we illustrate the structure of our proof on the part that is concerned with the single region R_∞ in which all clocks have elapsed beyond the *maximal clock constant of the automaton* (the region $c > 1$ in the example): any run on the region graph that stays in R_∞ forever is classified as non-zeno.

Our argument establishes that for a transition $(l, R_\infty) \rightarrow (l', R_\infty)$ of the region graph, the representing transition $(l, u) \rightarrow (l', u')$ will incur a time delay of 0.5 if $u \neq u'$. An informal argument can get away by claiming that the transition can always be chosen such that the latter condition is satisfied. Unfortunately, this is not immediately true for the semantics given above as the abstract transition could always be a reset transition and thus time would never be allowed to elapse. A possible remedy is to fuse delay and action transitions into a single step. We rather want to keep them separate and instead employ a probabilistic argument: assuming that a transition with $l = l'$ occurs infinitely often, with probability 1 a step with $u \neq u'$ has to occur infinitely often.

6 Conclusion

Discussion. Our bisimulation argument neatly separates discrete, TA-related reasoning from probabilistic, MDP-related reasoning. In fact, most of the proofs take place on the discrete side, because none of the arguments to satisfy the bisimulation theorem are predominantly of probabilistic nature. As seen above, only the reasoning on zeno-ness needs to break with this style.

We found it crucial to carry out each argument on the right level of abstraction. There are three main levels to consider here (from low to high): Markov chains, MDPs and configuration traces, and states and state traces of the PTA. The theorem is usually stated on the highest level possible, and often we can move easily from a higher to a lower level by applying a number of rewrite rules. For the divergence argument, we even introduce another level of abstraction: since we are only concerned with time, the location part can be dropped, and thus we consider traces of clock valuations. The probabilistic argument described in the last section manifests a rare case where one needs to put in some upfront work on a lower level to hold the argument on the higher level together.

It is not yet clear to us whether it is necessary or advantageous to work with *rep*. In the current formalization it still plays an important role by providing the diverging concrete witness for a diverging configuration of the region graph. The bisimulation argument in Sect. 4, however, can be made relying only on α .

Lastly, our simple definition of the PTA semantics and of the region graph shows that derived concepts can be surprisingly easy to define—even compared to an informal definition—if the necessary foundations have already been laid.

Related Work. We are not aware of any previous proof-assistant formalizations of PTA. There is, however, another formalization of TA and the region construction using PVS [11]. A formalization in Coq [7] is aimed at modeling a subclass of TA and proving properties of concrete automata.

Future Work. We conjecture that many further results for PTA can be formalized by following the formula “MDP + TA = PTA” in the style that we outlined above. In particular, most of the more practical *zone* based (as opposed to region based) exploration methods for the reduction to a finite MDP should lie within the scope of this technique. The backward reachability algorithm of PRISM [4] is an instance. This also means that verified or certified model checkers for PTA can be devised from a modular combination of verified tools for MDPs and TA. Work in this direction already exists for the latter [10] but not the former formalism.

Acknowledgments. We want to thank David Parker and Gethin Norman for clarifying our understanding of PTA model checking w.r.t. divergence. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 713999 - Matryoshka).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Th. Comp. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. Hölzl, J.: Markov chains and Markov decision processes in Isabelle/HOL. *J. Autom. Reasoning* **59**(3), 345–387 (2017). <https://doi.org/10.1007/s10817-016-9401-5>
3. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
4. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. *Inf. Comput.* **205**(7), 1027–1077 (2007)
5. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. *Th. Comp. Sci.* **282**(1)
6. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. *Formal Methods Syst. Des.* **43**(2), 164–190 (2013)
7. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 298–315. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45500-0_15
8. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_26
9. Wimmer, S., Hölzl, J.: Probabilistic timed automata. *Archive of Formal Proofs* (2018). Formal proof development. http://isa-afp.org/entries/Probabilistic_Timed_Automata.html
10. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4
11. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: *Proceedings of IASTED 2006*, pp. 107–112 (2006)

C Verified Model Checking of Timed Automata

Original publication. This chapter was originally published as a full paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer and Peter Lammich. “Verified Model Checking of Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Proceedings, Part I*. ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 61–78. DOI: [10.1007/978-3-319-89960-2_4](https://doi.org/10.1007/978-3-319-89960-2_4)



Synopsis. This paper reports on the construction of a mechanically verified prototype implementation of a model checker for timed automata. The goal of this work was two-fold: first, as a reference implementation the prototype should be fast enough to check other model checkers against it on reasonably-sized benchmarks; second, maximal feature compatibility with the state-of-the-art tool UPPAAL should be achieved. The starting point of this work was an existing highly abstract formalization of reachability checking for timed automata (as described in Paper A). Model checking of UPPAAL-style models was reduced to the problem of model checking a single automaton in this abstract formalization, while retaining the ability to perform on-the-fly model checking. Using the Isabelle RefinementFramework, the abstract specification of the model checker was refined, via multiple intermediate steps, to an actual imperative implementation in Standard ML. The resulting tool was evaluated on a set of standard benchmarks to demonstrate its practical usability.

Contribution. We have contributed the imperative refinement of DBMs, the verification and refinement of search algorithms, and the formalization of an imperative version of the Floyd–Warshall algorithm to equal parts. I have mostly contributed the work on the remaining parts, in particular: the functional refinement steps for DBMs, the product construction, the formalization of the bytecode semantics and a corresponding program analysis, the treatment of local clock ceilings, the connection between model checking properties and search algorithms, the extraction of an executable tool, and the experimental evaluation. I have written most of the text of the paper.

Copyright notice. This article is an open-access publication, which was published under the Creative Commons Attribution 4.0 International License. It is reproduced on the following pages in its original form. The original publication can be found under the DOI cited above.



Verified Model Checking of Timed Automata

Simon Wimmer^(✉) and Peter Lammich^(✉)

Fakultät für Informatik,
Technische Universität München,
Munich, Germany
{wimmers,lammich}@in.tum.de



Abstract. We have constructed a mechanically verified prototype implementation of a model checker for timed automata, a popular formalism for modeling real-time systems. Our goal is two-fold: first, we want to provide a reference implementation that is fast enough to check other model checkers against it on reasonably sized benchmarks; second, we strive for maximal feature compatibility with the state-of-the-art tool UPPAAL. The starting point of our work is an existing highly abstract formalization of reachability checking of timed automata. We reduce checking of UPPAAL-style models to the problem of model checking a single automaton in this abstract formalization, while retaining the ability to perform on the fly model-checking. Using the Isabelle Refinement Framework, the abstract specification of the model checker is refined, via multiple intermediate steps, to an actual imperative implementation in Standard ML. The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practical usability.

1 Introduction

Timed automata [1] are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL [2]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. However, mistakes have previously been made. This particularly concerns an approximation operation that is used by model-checking algorithms to obtain a finite search space. The use of this operation induced a soundness problem in the tools employing it [3], which was only discovered years after the first model checkers were devised.

Our ongoing work¹ addresses this issue by constructing a fully verified model checker for timed automata, using Isabelle/HOL [4]. Our tool is not intended to replace existing model checkers, but to serve as a reference implementation against which other implementations can be validated. Thus, it must provide sufficient performance to check real world examples. To this end, we use the

¹ <https://github.com/wimmers/munta>.

Isabelle Refinement Framework (IRF) [5, 6] to obtain efficient imperative implementations of the algorithms required for model checking.

Our work starts from an existing abstract formalization of *reachability* checking of timed automata [7]. To close the gap to a practical model checker, we need to address two types of issues: efficient implementation of abstract model checking algorithms, and expressiveness of the offered modeling formalism. Two kinds of algorithms deserve special attention here. The first are operations to manipulate Difference Bound Matrices (DBMs) [2], which represent abstract states. With the help of the IRF, we obtain efficient implementations of DBMs represented as arrays. The second are search algorithms that govern the search for reachable states. These algorithms are interesting in their own right, since they make use of *subsumption*: during the search process an abstract state can be ignored if a larger abstract state was already explored. We provide a generalized framework for different variants of search algorithms, including a version which resembles UPPAAL’s unified passed and waiting list [2].

We aim to offer a modeling formalism that is comparable in its expressiveness to the one of UPPAAL. To accomplish this goal while keeping the formalization effort manageable, we opt to accept UPPAAL *bytecode* as input. At the current state of the project we have formalized the semantics of a subset of the bytecode produced by UPPAAL. We support the essential modeling features: networks of automata with synchronization, and bounded integer state variables. We apply a product construction to reduce models of this formalism to a single timed automaton. As in real model checkers, the whole construction is computed *on the fly*. However, not every bytecode input designates a valid automaton. To this end, we employ a simple *program analysis* to accept a sufficiently large subset of the valid inputs.

We conducted experiments on a small number of established benchmark models. The throughput of our model checker — the number of explored states per time unit — is within an order of magnitude of a version of UPPAAL running a comparable algorithm.

1.1 Isabelle/HOL

Isabelle/HOL [4] is an interactive theorem prover based on Higher-Order Logic (HOL). You can think of HOL as a combination of a functional programming language with logic. Although Isabelle/HOL largely follows ordinary mathematical notation, there are some operators and conventions that should be explained. Like in functional programming, functions are mostly curried, i.e. of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$ instead of $\tau_1 \times \tau_2 \Rightarrow \tau$. This means that function application is usually written $f a b$ instead of $f(a, b)$. Lambda terms are written in the standard syntax $\lambda x. t$ (the function that maps x to t) but can also have multiple arguments $\lambda x y. t$, paired arguments $\lambda(x, y). t$, or dummy arguments $\lambda_. t$. Type variables are written $'a, 'b$, etc. Compound types are written in postfix syntax: $\tau \text{ set}$ is the type of sets of elements of type τ . In some places in the paper we have simplified formulas or code marginally to avoid distraction by syntactic or technical details, but in general we have stayed faithful to the sources.

1.2 Related Work

The basis of the work presented in this paper is our existing formalization of timed automata [7]. We are aware of one previous proof-assistant formalization of timed automata using PVS [8,9]. This work has the basic decidability result using regions and claims to make some attempt to extend the formalization towards DBMs. Another line of work [10,11] aims at modeling the class of p-automata [12] in Coq and proving properties of concrete p-automata within Coq. A similar approach was pursued with the help of Isabelle/HOL in the CClair project [13]. In contrast, our formalization [7] focuses on the foundations of timed automata model checking. In particular, it encompasses a formalization of the relevant DBM algorithms and the rather intricate developments towards the correctness proof for the approximation operation.

We are not aware of any previous formalizations or verified implementations of timed automata model checking. The first verification of a model checker we are aware of is by Sprenger for the modal μ -calculus in Coq [14]. Our important forerunner, however, is the CAVA project [15–17] by Esparza et al. It sets out for similar goals as we do but for *finite state LTL* model checking. A significant part of the refinement technology that we make use of was developed for this project, and it was the first project to demonstrate that verification of model checking can yield practical implementations. Compared to CAVA, our work offers several novelties: we target model checking of timed automata, which have an infinite state space; we use imperative data structures, which is crucial for efficient DBMs; finally, we implemented complex search algorithms with subsumption. Additionally, we operate on automata annotated with UPPAAL bytecode, which has interesting ramifications: for the product construction, and because we need to ensure that the input actually defines a timed automaton.

2 Timed Automata and Model Checking

2.1 Transition Systems

We take a very simple view of transition systems: they are simply a relation \rightarrow of type $'a \Rightarrow 'a \Rightarrow \text{bool}$. We model (*finite*) runs as *inductive* lists, and *infinite runs* as *coinductive* streams. We write $a \rightarrow xs \rightarrow b$ to denote the \rightarrow -run from a to b using the intermediate states in the list xs , and $a \rightarrow^{ys}$ to denote the infinite \rightarrow -run starting in a and then continuing with states from the stream ys . Additionally, we define:

$$a \rightarrow^+ b = (\exists xs. a \rightarrow xs \rightarrow b) \text{ and } a \rightarrow^* b = (a \rightarrow^+ b \vee a = b).$$

We define the five *CTL* properties that are supported by UPPAAL, $\mathbf{A}\Diamond$, $\mathbf{A}\Box$, $\mathbf{E}\Diamond$, $\mathbf{E}\Box$, and $\dashv\rightarrow$, as properties of infinite runs² starting from a state. For instance,

$$\mathbf{A}\Diamond \phi x = (\forall xs. x \rightarrow^{xs} \implies ev(\text{holds } \phi)(x \cdot xs)),$$

and

$$\phi \dashv\rightarrow \psi = \mathbf{A}\Box (\lambda x. \phi x \implies \mathbf{A}\Diamond \psi x),$$

where *ev* specifies that a property on a stream eventually holds, and *holds* constrains *ev* to the current state instead of the remainder stream. It then is trivial to prove identities such as $\mathbf{E}\Box \phi x = (\neg \mathbf{A}\Diamond (\text{Not} \circ \phi) x)$.

2.2 Timed Automata

Compared to standard finite automata, timed automata introduce a notion of clocks. Figure 1 depicts an example of a timed automaton. We will assume that clocks are of type *nat*. A *clock valuation* *u* is a function of type $\text{nat} \Rightarrow \text{real}$.

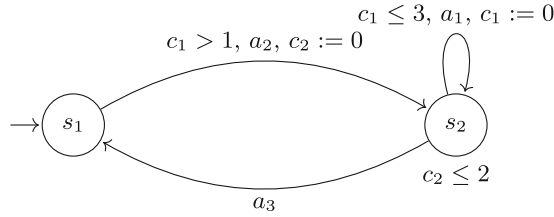


Fig. 1. Example of a timed automaton with two clocks.

Locations and transitions are guarded by *clock constraints*, which have to be fulfilled to stay in a location or to take a transition. Clock constraints are conjunctions of constraints of the form $c \sim d$ for a clock *c*, an integer *d*, and $\sim \in \{<, \leq, =, \geq, >\}$. We write $u \vdash cc$ if the clock constraint *cc* holds for the clock valuation *u*. We define a timed automaton *A* as a pair $(\mathcal{T}, \mathcal{I})$ where \mathcal{I} is an assignment of clock constraints to locations (also named invariants); and \mathcal{T} is a set of transitions written as $A \vdash l \xrightarrow{g,a,r} l'$ where *l* and *l'* are start and successor location, *g* is the guard of the transition, *a* is an action label, and *r* is a list of clocks that will be reset to zero when the transition is taken. States of timed automata are pairs of a location and a clock valuation. The operational semantics define two kinds of steps:

- Delay: $(l, u) \xrightarrow{d} (l, u \oplus d)$ if $d \geq 0$ and $u \oplus d \vdash \mathcal{I} l$;
- Action: $(l, u) \xrightarrow{a} (l', [r \rightarrow 0]u)$
if $A \vdash l \xrightarrow{g,a,r} l'$, $u \vdash g$, and $[r \rightarrow 0]u \vdash \mathcal{I} l'$;

² This is fairly standard in the literature [2,3,12,18] but differs slightly from the implementation in UPPAAL.

where $u \oplus d = (\lambda c. u \ c + d)$ and $[r \rightarrow 0]u = (\lambda c. \text{if } c \in r \text{ then } 0 \text{ else } u \ c)$. For any (timed) automaton A , we consider the transition system

$$(l, u) \rightarrow_A (l', u') = (\exists d \geq 0. \exists a \ u''. (l, u) \xrightarrow{d} (l, u'') \wedge (l, u'') \xrightarrow{a} (l', u')).$$

That is, each transition consists of a delay step that advances all clocks by some amount of time, followed by an action step that takes a transition and resets the clocks annotated to the transition. We write $A, s_0 \models \phi$ if ϕ holds in state s_0 w.r.t. \rightarrow_A . Note that it is crucial to combine the two types of steps in order to reason about liveness. Consider the automaton from Fig. 1 and assume the two kinds of steps could be taken independently. Then the automaton has a run on which some predicate P holds everywhere if and only if $P \ s_1$ holds.

2.3 Model Checking

Due to the use of clock valuations, the state space of timed automata is inherently infinite. Thus, model checking algorithms for timed automata are based on the idea of abstracting from concrete valuations to *sets* of clock valuations of type $(nat \Rightarrow real)$ *set*, often called *zones*. The initial decidability result [1] partitioned the state space into a quotient of zones, the so-called regions, and showed that these yield a sound and complete abstraction³. However, practical model checking algorithms rather explore the state space in an *on-the-fly* manner, computing successors directly on zones, which are typically represented symbolically as Difference Bound Matrices (DBMs). DBMs are simply a matrix-form representation of clock constraints, which contain exactly one conjunct for each pair of clocks. To represent constraints on single clocks, an artificial **0**-clock is added, which is assumed to be assigned 0 in any valuation.

The delicate part of this method is that the number of reachable zones could still be infinite. Therefore, an over-approximation is applied to zones to obtain a finite search space. We call the transition system of zones the *zone graph*, and the version where over-approximations are applied the *abstract zone graph* [18]. The soundness argument for this method (due to over-approximation completeness is trivial), starts from the region construction and then introduces the notion of the *closure* of a zone, which is defined to be the union of all regions intersecting with a zone. It can be shown from the correctness of the region construction that closures yield a sound over-approximation of zones. Finally, one shows that the result of applying the over-approximation operator to zones is always contained in the closure, thus inheriting soundness from the soundness of closures. We have formalized this argument and all of the material summarized in this section in previous work [7]. It only covers the case of reachability, but we will demonstrate how to extend the soundness argument to liveness below.

³ We use the same notions as in [7]. Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.

3 A First Glance at the Model Checker

This section provides a first overview of our model checker, its construction, and the correctness theorem we proved. The input to our checker consists of a model, i.e. a network of Timed Automata, and a formula to be checked against the model. To achieve high compatibility with UPPAAL, guards and updates can be formulated in UPPAAL bytecode⁴. This intermediate representation is computed by UPPAAL from the original C-style input before the actual model checking process is started. Given such an input, our tool will first determine whether the input is valid and lies in the supported fragment. This is achieved by a simple program analysis. As input formulae, our model checker accepts the same (T)CTL fragment that is supported by UPPAAL, but restricts formulae to not contain clocks. While this is not a principal limitation of our work, it reduced the complexity of our first prototype. If the input is invalid, our tool answers with “invalid input”, else it determines whether

$$\text{conv } N, (\text{init}, s_0, u_0) \models_{\text{max_steps}} \phi$$

holds for the all-zero valuation u_0 under the assumption that the automaton is deadlock-free⁵, and answers with true/false. Here, N is the input automaton, conv converts all integer constants to reals (as the semantics are specified on reals), and ϕ is the input formula. The relation $\models_{\text{max_steps}}$ is a variant of \models lifted to networks of timed automata with shared state and UPPAAL bytecode annotations. It is indexed with the maximum number of steps that any execution of a piece of UPPAAL bytecode can use (i.e. max_steps is the *fuel* available to executions). The vector of start locations init , and the shared state s_0 (part of the input) describe the initial configuration.

The actual model checking proceeds in two steps. First, a product construction converts the network to a *single* timed automaton, expressed by HOL functions for the transition relation and the invariant assignment. Second, according to the formula, a model checking algorithm is run on the single automaton. We need three algorithms: a reachability checker for $\mathbf{E}\diamond$ and $\mathbf{A}\square$, a loop detection algorithm for $\mathbf{E}\square$ and $\mathbf{A}\diamond$, and a combination of both to check $\text{--}\rightarrow$ -properties. Note that the aforementioned HOL functions are simply *functional programs* that construct the product automaton’s state and invariant assignments *on-the-fly*. The final correctness theorem we proved can be stated as follows:

$$\begin{aligned} & \{\text{emp}\} \\ & \text{precond_mc } p \ m \ k \ \text{max_steps } I \ T \ \text{prog } \text{formula } \text{bounds } P \ s_0 \\ & \{\lambda \text{Some } r \Rightarrow \text{valid_input } p \ m \ \text{max_steps } I \ T \ \text{prog } \text{bounds } P \ s_0 \ \text{na } k \wedge \\ & \quad (\neg \text{deadlock } (\text{conv } N) \ (\text{init}, s_0, u_0) \Longrightarrow \\ & \quad \quad r = \text{conv } N, (\text{init}, s_0, u_0) \models_{\text{max_steps}} \text{formula}) \\ & \quad | \ \text{None} \Rightarrow \neg \text{valid_input } p \ m \ \text{max_steps } I \ T \ \text{prog } \text{bounds } P \ s_0 \ \text{na } k\} \end{aligned}$$

⁴ For the time being, the bytecode needs to be pre-processed slightly, mainly to rename textual identifiers to integers.

⁵ Adding a check for deadlocked states to our algorithms would be conceptually simple but is left for future work.

This Hoare triple states that the model checker terminates and produces the result *None* if the input is invalid. If the input is valid and deadlock free, it produces the result *Some r*, where *r* is the answer to the model checking problem.

4 Single Automaton Model Checking

In this section, we describe the route from the abstract semantics of timed automata to the implementation of an actual model checker. The next section will describe the construction of a single timed automaton from the UPPAAL-model.

4.1 Implementation Semantics

Although we have established that the DBM-based semantics from Sect. 2 can only explore finitely many zones, it is still “too infinite”: the automaton and DBMs are described by real constants, and operations on DBMs are performed on infinitely many dimensions (i.e. clocks). Thus, we introduce an *implementation semantics*, in which automata are given by integer constants, and where the number of clocks is fixed. We prove equivalence of the semantics in two steps: first, we show that DBM operations need only be performed on the clocks that actually occur in the automaton; second, we show that all computations can be performed on integers, provided the initial state only contains integers.

For the former step, we simplify the operations under the assumptions that they maintain *canonicity* of DBMs. A DBM is canonical if it stores the tightest derivable constraint for each pair of clocks, i.e.

$$\text{canonical } M \ n = (\forall i \ j \ k. i \leq n \wedge j \leq n \wedge k \leq n \rightarrow M \ i \ k \leq M \ i \ j + M \ j \ k).$$

During model checking, the Floyd-Warshall algorithm is used to turn a DBM into its canonical counterpart.

For the latter step, we use Isabelle’s integrated parametricity prover [19] to semi-automatically transfer the operations from reals to integers.

As an example, Fig. 2 displays the refinement steps of the *up* operation, which computes the time successor of a zone *Z*, i.e. the set $\{u \oplus d \mid u \in Z \wedge d \geq 0\}$.

$$\begin{array}{ll} \text{up } M = (\lambda i \ j. & \text{up}_1 \ M = (\lambda i \ j. \\ \text{if } i > 0 \text{ then if } j = 0 \text{ then } \infty & \text{if } i > 0 \wedge j = 0 \\ \text{else } \min(M \ i \ 0 + M \ 0 \ j)(M \ i \ j) & \text{then } \infty \\ \text{else } M \ i \ j) & \text{else } M \ i \ j) \\ \text{(a)} & \text{(b)} \end{array}$$

$$\begin{array}{ll} \text{up}_2 \ M \ n = \text{fold} & \text{up}_3 \ M \ n = \text{imp_for}' \ 1 \ (n + 1) \\ (\lambda i \ M. M((i, 0) := \infty)) & (\lambda i \ M. \text{mtx_set} \ (n + 1) \ M \ (i, 0) \ \infty) \\ [1 ..< n + 1] \ M & M \\ \text{(c)} & \text{(d)} \end{array}$$

Fig. 2. Refinement stages of the *up* operation for computing time successors.

In the step from up to up_1 , the assumption that the input DBM is canonical is introduced. In up_2 , which is the version used in the implementation semantics, the operation is constrained to clocks 1 to n . Finally, in up_3 , the matrices are implemented as arrays and the fold is implemented as a foreach loop.

At this point, a naive exploration of the transitive closure of the implementation semantics would already yield a simple but inefficient model checker. The rest of this section outlines the derivation of a more elaborate implementation that is close to what can be found in UPPAAL.

4.2 Semantic Refinement of Successor Computation

We further refine the implementation semantics to add two optimizations to the computation of successor DBMs: to canonicalize DBMs it is sometimes sufficient to only “repair” single rows or columns instead of running the full Floyd-Warshall algorithm; moreover, we can terminate the computation early whenever we discover a DBM that represents an empty zone (as it will remain empty). Both arguments are again carried out on the semantic level.

4.3 Abstraction of Transition Systems with Closures

Recall that the correctness of the reachability analysis on the abstract zone graph in Sect. 2 was obtained arguing that the region closure of zones forms a sound over-approximation of zones, which in turn is larger than the abstract zone graph. We want to reuse the same kind of argument to also argue that there exists a cycle in the abstract zone graph if and only if there is a cycle in the automaton’s transition system. This proof is carried out in a general abstract framework for transition systems and their abstractions.

We consider a concrete step relation \rightarrow_C over type $'a$ and what is supposed to be its simulation, a step relation \rightarrow_{A_1} over type $'a$ set. We say that \rightarrow_{A_1} is *post-stable* [20] if $S \rightarrow_{A_1} T$ implies

$$\forall s' \in T. \exists s \in S. s \rightarrow_C s',$$

and that \rightarrow_{A_1} is *pre-stable* [20] if $S \rightarrow_{A_1} T$ implies

$$\forall s \in S. \exists s' \in T. s \rightarrow_C s'.$$

In the timed automata setting, for instance, the simulation graph is post-stable and the region graph is pre-stable.

Lemma 1. *If \rightarrow_{A_1} is post-stable and we have $a \rightarrow_{A_1} as \rightarrow_{A_1} a$ with a finite and non-empty, then there exist xs and $x \in a$ such that $x \rightarrow_C xs \rightarrow_C x$.*

Proof. Let $x \rightarrow y = (\exists xs. x \rightarrow_C xs \rightarrow_C y)$. As \rightarrow_{A_1} is post-stable, every a has an ingoing \rightarrow -edge. Because a is finite we can thus find an \rightarrow -cycle in a , and obtain the claim.

Lemma 2. *If \rightarrow_{A_1} is pre-stable and we have $a \rightarrow_{A_1} as \rightarrow_{A_1} a$ and $x \in a$, then there exist xs such that $x \rightarrow_C^{xs}$ and xs passes through a infinitely often.*

Proof. By coinduction. From pre-stability we can find $x_1 \in a$ such that $x \rightarrow_C^+ x_1$, from x_1 we find $x_2 \in a$ such that $x_1 \rightarrow_C^+ x_2$, and so forth.

We can now consider doubly-layered abstractions as in the case for regions and zones. That is, we add a second simulation \rightarrow_{A_2} and two predicates P_1 and P_2 that designate valid states of the first and second simulation layer, respectively. Then we define the *closure* \mathcal{C} of a state of the second layer as

$$\mathcal{C} a = \{x \mid P_1 x \wedge a \cap x \neq \emptyset\} \text{ and } a \rightarrow_C b = (\exists x y. a = \mathcal{C} x \wedge b = \mathcal{C} y \wedge x \rightarrow_{A_2} y).$$

We assume that \rightarrow_{A_1} is pre-stable w.r.t. \rightarrow_C and that \rightarrow_C is post-stable w.r.t. \rightarrow_{A_1} . Along with some side conditions on P_1 and P_2 ⁶ we can prove:

Theorem 1. *If $a_0 \rightarrow_{A_2} as \rightarrow_{A_2} a \rightarrow_{A_2} bs \rightarrow_{A_2} a$ and $P_2 a$, then there exist $x \in \bigcup(\mathcal{C} a_0)$ and xs such that $x \rightarrow_C^{xs}$ and xs passes through $\bigcup(\mathcal{C} a)$ infinitely often.*

Proof. We first apply \mathcal{C} to the second layer states and get a path of the form: $\mathcal{C} a_0 \rightarrow_C as' \rightarrow_C \mathcal{C} a \rightarrow_C bs' \rightarrow_C \mathcal{C} a$ for some as' and bs' . From Lemma 1 and post-stability, we obtain a path of the form $a_{0_1} \rightarrow_{A_1} as_1 \rightarrow_{A_1} a_1 \rightarrow_{A_1} bs_1 \rightarrow_{A_1} a_1$ with $a_{0_1} \in \mathcal{C} a_0$ and $a_1 \in \mathcal{C} a$. By applying Lemma 2 and pre-stability, we obtain the desired result.

This is the main theorem that allows us to run cycle detection on the abstract zone graph during model checking: the other direction is trivial, and the theorem can be directly instantiated for regions and (abstracted) zones. There is a slight subtlety here since we only guarantee $x \in \bigcup(\mathcal{C} a_0)$. However we typically have $\mathcal{C} a_0 = a_0$, as all clocks are initially set to zero.

4.4 Implementation of Search Algorithms

We first implement the three main model checking algorithms abstractly in the nondeterminism monad provided by the IRF. On this abstraction level, we can use such abstract notions as sets and specify the algorithm for an arbitrary (finite) transition system \rightarrow . We only showcase the implementation of our cyclicity checker (used for $\mathbf{A}\diamond$ and $\mathbf{E}\square$). The techniques used for the other algorithms are similar. The code for our cyclicity checker is displayed in Listing 1.1.

⁶ P_1 states are distinct and there are only finitely many of them. For every P_2 state, there is an overlapping P_1 state.

```

dfs P = do {
  (P, ST, r) ← rec⊥ (λdfs (P, ST, v).
    do {
      if ∃v' ∈ set ST. v' ≼ v then return (P, ST, True)
    else do {
      if ∃v' ∈ P. v ≼ v' then return (P, ST, False)
    else do {
      let ST' = v · ST;
      (P, ST', r) ←
        foreach {v' | v → v'} (λ(-, -, b). ¬b)
          (λv' (P, ST, -). dfs (P, ST, v'))
          (P, ST, False);
      assert (ST' = ST);
      return (insert v P, tl ST', r)
    }
  }
  } (P, [], a0);
return (r, P)}

```

Listing 1.1. Cyclicity Checker

We claim that this closely resembles the pseudo-code found, e.g., in [21]. The algorithm takes a passed set, and produces a new passed set in addition to the answer. This can be used in the algorithm for checking $\dashv\rightarrow$ -properties. The crux of the algorithm is the use of the subsumption operator \preceq , to check whether smaller states are already subsumed by larger states that we may have discovered earlier (for timed automata, this would correspond to set inclusion on zones). We assume that \preceq is a pre-order and monotone w.r.t. \rightarrow . Then, using the IRF's verification condition generator, we prove:

$$\begin{aligned}
& \text{dfs } P \leq \text{SPEC } (\lambda(r, P'). (r \implies (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x)) \\
& \wedge (\neg r \implies \neg (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x) \wedge \text{liveness_compatible } P')) \\
& \text{if } \text{liveness_compatible } P.
\end{aligned}$$

The invariant we maintain for the passed set P is encoded in the predicate *liveness_compatible* P . We say that a state x is covered by P if there exists $x' \in P$ such that $x \preceq x'$. Then, informally, *liveness_compatible* P states that the successors of every node that is covered by P are also covered, and that there is no cycle through nodes that are covered by P . After specifying the correct loop invariant (using *liveness_compatible* as the main insight) and the termination relation, together with some key lemmas about the invariant, the verification conditions can be discharged nearly automatically.

In subsequent steps, we gradually refine this implementation to use more efficient data structures. The final version uses a function to compute a list of successors, and is able to *index* the passed set and the stack according to a key function on states (this corresponds to the location part of states in the abstract zone graph). The refinement theorem can be stated as:

$$\text{dfs_map } P \leq \Downarrow (\text{Id} \times_r \text{map_set_rel}) (\text{dfs } P') \text{ if } (P, P') \in \text{map_set_rel}.$$

That is, dfs_map is a refinement of dfs , where the passed set is data-refined w.r.t. the relation map_set_rel . This relation describes the implementation of passed sets indexed by keys.

These refinement steps are conducted inside the nondeterminism monad of the IRF. The final step leads into the heap-monad of Imperative HOL [22], which supports imperative data structures. Here, the *Sepref* tool [6] replaces functional by imperative data structures and generates a refinement theorem automatically.

Maps are implemented via hash tables, which poses a challenge for the implementation as the maps contain objects stored on the heap. This was not supported by the existing implementation in the Imperative Collections Framework, due to sharing issues: when retrieving a value from the map, we cannot obtain ownership of the value while the map also retains ownership. This is even true if the value is read-only. One way to solve this problem would be to extend the separation logic that underlies the IRF to fractional permissions or read-only permissions. Our solution, however, is more ad-hoc: we simply restrict the operations that we perform on the hash map to insertions and an *extract* operation, which deletes a key-value pair from the map and returns the value (i.e. it combines lookup and delete). To define the map implementation, we use a trick similar to Chargueraud’s ideas from [23]: we use a *heap assertion* that first connects an abstract map m with an intermediate map m_h of pointers to the elements, and then implements the map of pointers by a regular hash map m_i . Formally, the assertion is defined as:

$$hms_assn\ A\ m\ m_i = (\exists_{A m_h}. is_map\ m_h\ m_i * map_assn\ A\ m\ m_h).$$

Here is_map is the assertion for an existing map implementation from the Imperative Collections Framework (which cannot store heap objects, but supports pointers), and $map_assn\ A\ m\ m_h$ connects a map of abstract values with a map of pointers, where the relation between abstract values and pointed-to objects is defined by A .

Then, the final implementation is produced by proving that all map-related operations in dfs_map can be replaced by insert and extract operations, and letting *Sepref* synthesize the imperative variant, making use of our new hash map implementation. The key theorem on the final implementation is the following Hoare triple:

$$\begin{aligned} &\{emp\} \\ &\quad dfs_map_impl' \ succsi\ a_0i\ Lei\ keyi\ copyi \\ &\{ \lambda r. r = (\exists x. a_0 \rightarrow^* x \wedge x \rightarrow^+ x) \} \end{aligned}$$

It is expressed in a locale (Isabelle’s module system) that assumes that a_0i , $succsi$, etc., are the correct imperative implementations of a_0 , the successor function, and so forth. Versions of the search algorithm for concrete transition systems are obtained by simply instantiating the locale with the operations of the transition system and their implementations.

4.5 Imperative Implementations of Model Checking Operations

Recall the refinement of the up operation (Fig. 2). It is crucial that up_2 is expressed as a fold-operation with explicit updates, as only then the IRF can extract an efficient imperative version with destructive updates and a foreach loop. The imperative implementation up_3 is, again, synthesized by the Sepref tool. As can be witnessed for up_3 , the pattern $fold f [1 ..< n + 1]$ is turned into a foreach loop. Technically, this is achieved by a set of rewrite rules that are applied automatically by the Sepref tool at the end of the synthesis process. The only hurdle for this kind of synthesis is that the dimension of DBMs needs to become a parameter of the refinement relations. For n clocks, we define

$$mtx_assn = asmtx_assn (n + 1) id_assn.$$

This specifies that our DBMs are implemented by square-arrays of dimension $n + 1$, and their elements are refined by the identity relation.

The refinement theorem for up_3 is proved automatically by the Sepref tool:

$$(up_3, up_2) \in [\lambda(-, i). i \leq n] mtx_assn^d * nat_assn^k \rightarrow mtx_assn.$$

This theorem states that, if the specified dimension is in bounds, up_3 refines up_2 . The \cdot^d annotation indicates that the operation is allowed to overwrite (destroy) the input matrix on the heap. Symmetrically, the \cdot^k annotation means that the second parameter is not overwritten (kept).

4.6 Code Extraction

Finally, Isabelle/HOL's code generator [24] is used to extract imperative Standard ML code from the Isabelle specifications generated by Sepref. Code generation involves some optimizations and rewritings that are carried out as refinement steps and proved correct, followed by pretty printing from the functional fragment of HOL and the heap monad to Standard ML.

5 From UPPAAL-Style Semantics to a Single Automaton

5.1 UPPAAL-Style Semantics

Due to the lack of documentation on the UPPAAL intermediate format, we define an approximation of this assembler-like language by reverse engineering. This is sufficient to check typical benchmarks, and gives a clearly defined semantics to the fragment that we cover. The language is defined as a simple data type $instr$. A $step$ function of type $instr \Rightarrow state \Rightarrow state option$ computes the successor state after executing an instruction, or fails. A state consists of an instruction pointer, the stack, the state of the shared integer variables, the state of the comparison flag, and a list of clocks that have been marked for reset. Using a fuel parameter, we execute programs by repeatedly applying the $step$ function

until we either reach a halt instruction, fail, or run out of fuel, which we also regard as a failed execution.

A special instruction *CEXP* is used to check whether an atomic clock constraint holds for a given valuation u . However, this instruction cannot simply be executed during model checking as it would need to work on zones instead of valuations. Unconstrained use of the *CEXP* instruction would allow for disjunctions of clock constraints on edges, which is not part of the standard timed automata formalism. Thus, in the same way as UPPAAL, we restrict the valid input programs to those that only yield conjunctions of clock constraints on edges. We then replace every *CEXP* instruction by a special meta instruction that sets the comparison flag to true. This amounts to enforcing a program execution where the clock constraint, which is expressed by a piece of bytecode, holds for a valuation. Edges are annotated with the conjunction of the atomic clock constraints encountered during execution. In the current version of our tool, we separate concerns for locations by using a state predicate, which is not allowed to use *CEXP* instructions, and a separate clock constraint. The two could be merged by using the same approach as for edges.

5.2 Program Analysis

As stated in the last section, we need to ensure that successful program executions can only induce conjunctive clock constraints. That is, we need to ensure that program executions can only be successful when all *CEXP* instructions that are encountered during execution evaluate to true. To this end, we use a naive analysis, which recognizes a subclass of these programs that is sufficiently large to cover common timed automata benchmarks. This analysis tries to identify what we call *conjunction blocks*. A conjunction block reaching from addresses pc_s to pc_t ends with a *halt* instruction at pc_t , starts with a *CEXP* instruction at pc_s and then is extended to pc_t by repeatedly using one of the following two patterns:

- a *copy* instruction to push the flag on the stack, followed by *CEXP* and an *and* instruction;
- a *copy* instruction, followed by a *jump-on-zero* instruction with pc_t as the destination, followed by *CEXP* and an *and* instruction.

We simultaneously show the two key properties of conjunction blocks via induction: if there is a conjunction block from pc_s to pc_t , then any successful execution starting from pc_s ends in pc_t , and every *CEXP* instruction that is encountered has to evaluate to true. Given a start address pc_s , the whole analysis works by computing an approximation of the set of possible addresses that can be reached from pc_s , say S , and then checking whether

$$\text{Min } \{pc \mid pc \in S \wedge (\exists ac. P_{pc} = \text{CEXP } ac)\} \text{ to Max } S$$

is a conjunction block, where P_{pc} is the program instruction at address pc . A major limitation of this analysis is that it cannot approximate the reachable

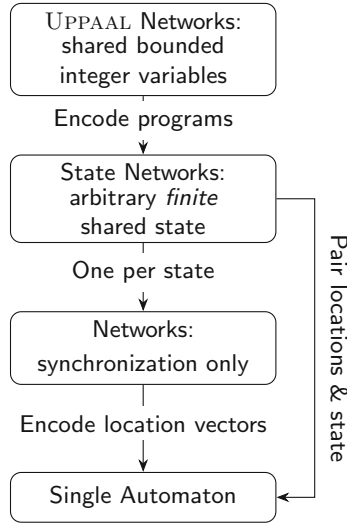


Fig. 3. Outline of the product construction.

set for *call* and *return* instructions, so we are not able to handle inputs that are compiled from UPPAAL programs with sub-routines. However, as the main objective of our work is not program analysis, we consider the current naive analysis sufficient to demonstrate the general viability of our approach.

5.3 Product Construction

The general shape of our product construction is outlined in Fig. 3. The first stage of the construction encodes the bytecode annotations as guards and updates on the shared state. The subsequent stage constructs a network of automata for *each shared state* by essentially filtering the transitions that are valid for a given state. For a simple network, the product can be constructed in the obvious way. However, this is only used in the correctness proof of the final step, which directly constructs a single automaton by pairing the location vector and the state.

The result of this construction is a highly contrived description of the single automaton. To obtain an efficiently executable version of this description, we specify an alternative functional implementation and prove the equivalence of the two.

6 Experimental Evaluation

We conducted experiments on some standard benchmark models for timed automata: a variant of Fischer’s mutual-exclusion protocol, the FDDI token ring protocol, and the CSMA/CD protocol used in Ethernet networks. We tested one reachability and one liveness property for each model: $\mathbf{E}\diamond(c > 1)$ and $P_1.b \dashrightarrow$

$P_1.c$ for Fischer’s protocol; $\mathbf{E}\diamond(\neg P_1.idle \wedge \neg P_2.idle)$ and $true \dashrightarrow z_async_1$ for FDDI; and $\mathbf{E}\diamond(P_1.abort \wedge P_2.send)$, and $collision \dashrightarrow active$ for CSMA/CD. We compare (c.f. Table 1) our tool against UPPAAL configured with two different approximation operators: difference (UPPAAL₁) and location-based (UPPAAL₂) extrapolation. We give the computation time in seconds and the number of explored states, as reported by our tool and UPPAAL⁷. Since the number of explored states differs significantly, we also calculated *throughput*, i.e. the number of explored states per second. The ratio of UPPAAL’s throughput and our tool’s throughput is given in the column *TR*. We specify the problem size as the number of automata in the network.

Table 1. Experimental results on a set of standard benchmarks.

Model	Prop	SAT	Size	Our tool		UPPAAL ₁			UPPAAL ₂		
				Time	#States	Time	#States	TR	Time	#States	TR
Fischer	R	N	5	6,61	38578	0,31	12363	6,83	0,04	3739	16,02
	L	Y	5	7,52	42439	0,31	20340	11,8	0,04	8149	40,1
		Y	6	485,9	697612	42,85	249295	4,1	1,53	67325	30,7
FDDI	R	N	8	16,04	6720	0,34	5416	37,6	0,31	5416	42,0
		N	10	142,8	29759	6,63	24210	17,5	6,44	24120	18,0
	L	Y	6	2,58	2083	0,05	2439	61,7	0,04	2439	68,7
		Y	7	6,50	3737	0,15	4944	57,0	0,14	4944	62,3
CSMA/CD	R	N	5	4,48	9959	0,03	2704	45,3	0,03	2769	40,6
		N	6	71,70	81463	1,70	17613	9,2	1,79	17939	8,8
	L	Y	5	4,93	11526	0,04	3802	42,4	0,04	3867	42,4
		Y	6	76,83	96207	1,78	23128	10,4	1,86	12603	10,1

The results indicate that our tool’s throughput is around one order of magnitude lower than UPPAAL’s. Encouragingly, the gap seems to decrease for larger models. However, for larger problem sizes of some models, we also start to run out of memory because our tool is not tuned towards space consumption. We do not have a convincing explanation for the difference in states explored by our tool and UPPAAL — particularly, because our tool already implements location-based extrapolation. Nevertheless, we conclude that the performance offered by our tool is reasonable for a reference implementation against which other tools can be validated: we can check medium sized instances of common benchmark models, which should be sufficient to scrutinize the functionality of a model checker.

⁷ UPPAAL comes with a note suggesting that these numbers might be wrong for liveness properties.

7 Conclusion

We have derived an efficiently executable and formally verified model checker for timed automata. Starting from an abstract formalization of timed automata, we first reduced the problem to model checking of a single automaton, and then used stepwise refinement techniques to gradually replace abstract mathematical notions by efficient algorithms and data structures. Some of the verified algorithms and data structures, e.g. search with subsumption and Difference Bound Matrices, are interesting in their own right. Our experiments demonstrate that our tool’s performance is suitable for validating other model checkers against it on medium sized instances of classic benchmark models. Using a simple program analysis, we can cover a subset of the UPPAAL bytecode that is sufficient to accept common models as an input.

Following the construction we expounded above, our checker can be improved on two different axes: advanced modeling feature such as broadcast channels or committed locations can be enabled by elaborating the product construction; using the refinement techniques that we demonstrated above, further improvements of the model checking algorithms can achieve better performance.

An alternative approach to tackle performance problems is to resort to certification of model checking results. For the simple CTL properties that are supported by our tool and UPPAAL, passed sets could be used as the certificates and the model checking algorithms could be reused for certificate checking. As the model checking algorithms for timed automata make use of subsumption, passed sets can contain significantly less states than the total number of states explored during model checking. We plan on exploring this avenue in the future.

Data Availability Statement. The datasets generated during and analyzed during the current study are available in the figshare repository [25]: <https://doi.org/10.6084/m9.figshare.5917363.v1>.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994)
2. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *ACPN 2003*. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
3. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) *STACS 2003*. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36494-3_54
4. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): *Isabelle/HOL*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
5. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft’s algorithm. In: Beringer, L., Felty, A. (eds.) *ITP 2012*. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12

6. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_17
7. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_26
8. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: Proceedings of the IASTED International Conference on Software Engineering, pp. 107–112 (2006)
9. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: Proceedings of SNPD 2009, pp. 555–560 (2009)
10. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 298–315. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45500-0_15
11. Garnacho, M., Bodeveix, J.-P., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 106–120. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40229-6_8
12. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pp. 592–601 (1993)
13. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLs 2001: Supplemental Proceedings (2001). <http://www.informatics.ed.ac.uk/publications/report/0046.html>
14. Sprenger, C.: A verified model checker for the modal μ -calculus in Coq. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054171>
15. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
16. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 105–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_7
17. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reasoning* **60**(1), 3–21 (2018)
18. Herbreteau, F., Srivathsan, B., Tran, T.T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) FSTTCS 2016, vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 48:1–48:14 (2016)
19. Huffman, B., Kunčar, O.: Lifting and transfer: a modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_9
20. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-fly symbolic model checking for real-time systems. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997), 3–5 December 1997, San Francisco, CA, USA, pp. 25–34 (1997)
21. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Beyond liveness: efficient parameter synthesis for time bounded liveness. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 81–94. Springer, Heidelberg (2005). https://doi.org/10.1007/11603009_7

22. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
23. Charguéraud, A.: Higher-order representation predicates in separation logic. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016, St. Petersburg, FL, USA, pp. 2–14. ACM, New York (2016). <https://doi.org/10.1145/2854065.2854068>
24. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
25. Wimmer, S., Lammich, P.: Verified model checking of timed automata - artifact (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



D Munta: A Verified Model Checker for Timed Automata

Original publication. This chapter was originally published as a short paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer. “Munta: A Verified Model Checker for Timed Automata.” In: *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Proceedings*. Ed. by Étienne André and Mariëlle Stoelinga. Vol. 11750. Lecture Notes in Computer Science. Springer, 2019, pp. 236–243. DOI: 10.1007/978-3-030-29662-9_14

Synopsis. This paper presents MUNTA, a verified model checker for timed automata. It is intended to achieve two goals: first, as a reference implementation it should be fast enough to test other model checkers against it on reasonably-sized benchmarks; second, the tool should be practical enough so that it can easily be used for experimentation. Munta can be compiled to Standard ML or OCaml and additionally features a web-based GUI. Its modeling language has a simple semantics but provides the most commonly used timed automata modeling features.

Contribution. I am the sole author of this paper. Therefore, all contributions are mine.

Copyright notice. On the following pages the full article is reprinted by permission from Springer Nature Customer Service Centre GmbH. The original publication can be found under the DOI cited above.



Munta: A Verified Model Checker for Timed Automata

Simon Wimmer^(✉) 

Fakultät für Informatik, Technische Universität München, Munich, Germany
wimmers@in.tum.de

Abstract. Munta is a mechanically verified model checker for timed automata, a popular formalism for modeling real-time systems. Our goal is two-fold: first, we want to provide a reference implementation that is fast enough to test other model checkers against it on reasonably sized benchmarks; second, the tool should be practical enough so that it can easily be used for experimentation. Munta can be compiled to Standard ML or OCaml and additionally features a web-based GUI. Its modeling language has a simple semantics but provides the most commonly used timed automata modeling features.

1 Objective and Overview

Timed automata [1] are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL [3]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. Consequently, we would like to ensure two things: first, the theory behind the tools should be sound and well-understood. Second, the implementations of the theory in real model checkers should be correct.

To address these concerns, we present Munta¹, a model checker for timed automata with a full correctness proof in the interactive theorem prover Isabelle/HOL [13]. Everything is in one place, in a formal and highly reliable format: the theory, from the basic formalism, over abstract formalizations of fundamental concepts such as regions and zones, down to concrete algorithms on Difference Bound Matrices (DBMs), is formalized and checked in Isabelle/HOL. Moreover, we can generate executable code from this formalization to obtain a trustworthy model checker.

Having a formally verified tool at hand gives rise to the possibility of testing other model checkers against it, in order to find errors in the other tools' implementations. Therefore, the verified checker needs to be fast enough to run it on reasonably sized benchmarks. To this end, we use refinement with the Imperative Refinement Framework (IRF) [11] to obtain efficient imperative implementations of the DBM algorithms that lie at the heart of state-of-the-art timed automata model checking.

¹ <https://wimmers.github.io/munta/>.

Moreover, a formally verified tool can serve as a valuable basis for experimentation. First, it allows one to devise extensions and modifications of the theory, to prove them correct, and to experiment with them in a real tool—all in one place. Second, the tool can be used to gain definite insights into the formalism and the model checking process by evaluating results on small models or by examining the state space that was explored by the verified tool. To support these roles, Munta provides a clear-cut modeling language with a standard semantics. Additionally, typical useful features of real model checkers such as reporting the set of explored states and deadlock checking are supported.

This paper gives an overview of Munta’s functionality and architecture from a user’s perspective. A theoretical account of the main ideas for the construction of the verified checker can be found in previous work [16, 17].

2 Functionality

2.1 Modeling Language

Munta’s modeling language supports a typical set of features: networks of timed automata that can synchronize over channels and share a discrete finite state, which is characterized by a set of integer variables. Guards and updates on the discrete state can be expressed with a simple language of Boolean and arithmetic expressions. Additionally, Munta supports the popular features of broadcast channels, and urgent and committed locations. Currently, there still exist some restrictions compared to other commonly used modeling languages: automata need to be diagonal free (i.e. clock constraints cannot involve differences of clocks) and updates can only reset clocks to zero. These limitations could be removed, however, by elaborating the current formalization. Moreover, they are the most commonly found restrictions of the formalism in tools and literature.

The formalized semantics of this language is compact, i.e., only around 150 lines of Isabelle formalization (compare this to the informal description found in the UPPAAL reference manual, for instance). This is the main basis of trust: if one accepts that this semantics is sensible and one trusts the correctness of Isabelle/HOL, one can assert full trust in Munta.

2.2 Correctness Theorem

This section briefly describes the correctness theorem for Munta informally. For a formal account see our previous work [17] or the Isabelle/HOL formalization. The correctness theorem is formulated in a separation logic [11] for Imperative HOL [5], which extends HOL with imperative programming features.

The theorem shows that the model checker will terminate and either return a result (*sat* or *unsat*), or report an error. It is proved that, when a result is returned, that it correctly indicates whether the model satisfies the formula. Two kind of errors can be reported: they either signify that the input model is

malformed, or that the correctness check for a certified part of Munta failed (c.f. Sect. 3.2). It is also proved that errors are only reported if they really arise.

Note that correctness is ensured only with respect to the semantics of the modeling language and the semantics of Imperative HOL. None of the individual proof steps nor our formalization of model checking algorithms need to be trusted as everything is checked by Isabelle/HOL’s logical kernel.

2.3 Input Format

Input models to the checker are provided in a simple JSON format. On the one hand, this means that input files are rather easy to read and understand for a human. On the other hand, it facilitates data exchange with other tools as parsers and printers for this format are readily available in many programming languages. We refrain from using a templating mechanism (like, e.g. UPPAAL) to provide templates of models that can be instantiated to obtain a concrete network. This way, translating to and from our input format remains simple. Finally, arbitrary fields can be added to objects anywhere in the JSON files. This allows one to transport formally irrelevant meta-information, such as the coordinates of locations in a visual representation.

2.4 Modeling Checking Capabilities

Munta can check formulas from the subset of CTL that corresponds to the subset of TCTL that is supported by UPPAAL. Moreover, Munta provides a deadlock checker. This is essential for practical use of the tool, as usually one wants to ensure that models are deadlock free before verifying more complex properties. Additionally, Munta can compute the complete set of reachable states and provide this information to the user. This is vital for understanding and debugging models.

2.5 Graphical User Interface

We provide a web-based GUI for Munta, programmed in the OCaml derivative Reason² using the ReasonReact framework³. The GUI can interface with the model checker in two ways: first, we provide a server mode, where queries can be sent to a verification server running locally; second, we compile the OCaml version of our checker to JavaScript, running it directly in the browser. We use a *parse-print-parse* loop to ensure that the user’s input is understood correctly by the model checker: the GUI can display a normalized version of the user’s input that is guaranteed to parse to the same internal representation as the user’s original input. Munta itself again uses another parse-print-parse loop to produce a JSON description of the model that is guaranteed to parse to the same object as the JSON description that was extracted from the internal representation in the GUI.

² <https://reasonml.github.io/>.

³ <https://reasonml.github.io/reason-react/>.

In addition, the user can directly inspect this final JSON representation of the model, to ensure that no errors were introduced during the translation from the visual representation.

3 Architecture

Figure 1 gives an overview of the system architecture, which is described in more detail in this section.

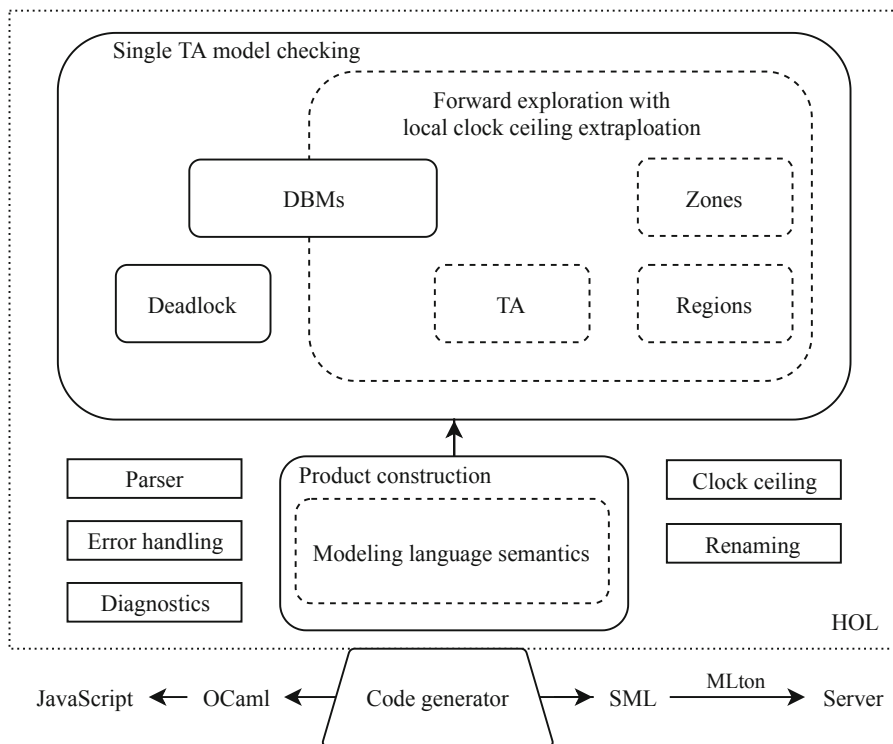


Fig. 1. Overview of the system architecture. Round boxes represent HOL formalizations. Boxes with sharp corners are pieces of unverified HOL code. Solid lines represent formalizations that include a refinement step to efficient (imperative) implementations.

3.1 Isabelle/HOL Formalization

The formalization consists of three main parts. The first part is an abstract formalization of the basic timed automata formalism (for a single automaton). This includes a formalization of the region construction, zones, and the local

clock ceiling extrapolation operation for zones [2]. In the end, we can prove that forward analysis of timed automata with zones and extrapolation is correct [16].

The second part is an abstract formalization of DBMs and elementary model checking algorithms for CTL, which are refined to concrete, executable, and partly imperative implementations. Together with the first part, we obtain an executable model checker for single timed automata [17].

The third part is a formalization of the modeling language and of an on-the-fly product construction to obtain a single automaton. This construction yields descriptions of the single automaton’s invariants and transitions as functional programs that can be plugged into the implementation obtained from the second part. To implement the product construction efficiently, a pre-processing step renames the labels in the model from human-readable strings to consecutive natural numbers.

3.2 Code Extraction and Glue Code

From the HOL specification of the model checker, Isabelle/HOL can generate code in Standard ML (SML) and OCaml [6]. However, this alone would not yield a usable tool. We need additional code for parsing the input, error handling, and retrieving diagnostic information. These functionalities are directly implemented as functional programs in HOL to keep the code portable, possibly to also export it to Isabelle/HOL’s other target languages, Haskell and Scala.

To implement the parser, we use a parser combinator library for Isabelle/HOL [12]. A correctness proof for the parser is replaced by ensuring consistency of the parse-print-parse loop for each concrete input, as explained above.

Diagnostic information is obtained in a minimally intrusive way. This usually involves defining, e.g., a constant *PRINT* of the HOL type $string \Rightarrow unit$ as $PRINT\ s = ()$. This constant can easily be stripped away by Isabelle’s proof automation (by essentially unfolding its definition). To actually obtain output, we instruct the code generator to translate this specific constant in a way that some side-effect is performed, e.g., the string *s* is output to the console or logged in some background data structure. A similar technique can be used to obtain time measurements or to trace information about explored states by using other constants of some type $\alpha \Rightarrow unit$.

Two parts of the model checker, the pre-processing step to relabel the model, and the code to compute the local clock ceilings, are not verified but only certified: their computed results are checked for soundness by a verified part of the model checker.

The ability to target SML as well as OCaml holds some advantages. With SML, faster executables can be obtained by using the highly optimizing compiler MLton. In contrast, OCaml code compiles to less efficient executables but is very conveniently compatible with the implementation of our frontend: Reason hinges on the same backend as the one that we use to compile OCaml to JavaScript⁴, making it easy to run our verified model checker directly in the browser.

⁴ <https://bucklescript.github.io/>.

4 Discussion

4.1 Comparison to Other Tools

We have previously reported on an experimental evaluation of Munta, comparing it to the state-of-the-art timed automata model checker UPPAAL [17]. Generally, Munta’s throughput (the number of explored states per time unit) is within an order of magnitude of UPPAAL’s throughput. Munta is also fast enough to check medium-sized benchmarks within reasonable time.

Compared to UPPAAL, Munta is not only much slower, but also only provides a less sophisticated modeling language. UPPAAL supports such sophisticated features as a C-like language to describe guards and invariants on edges, channels with priorities, or a templating mechanism. However, our modeling language does not differ as significantly in its expressiveness from tools such as Prism (and its implementation of probabilistic timed automata) [9], TChecker [7], Rabbit [4] and RED [15]. Moreover, one can argue that for a tool which is mainly intended as a platform for experimentation, it is not crucial to provide an exhaustive array of modeling features. Instead, a simple modeling language with a clear semantics may even be advantageous.

4.2 Trusted Code Base

To trust the results of Munta, one needs to trust the following components:

- (a) our formalization of the modeling language semantics as described above,
- (b) the formalization of Imperative HOL and its corresponding separation logic,
- (c) Isabelle/HOL’s logical kernel,
- (d) Isabelle/HOL’s code generator,
- (e) and the target language’s compiler and runtime system.

Trust in (a) and (b) can only be obtained by manual inspection. Regarding (c), it is widely accepted within the community that Isabelle/HOL only admits valid theorems (at least on the user level). The trustworthiness of components (d) and (e) is more debatable, however. Recent (ongoing) work by Hupel and Nipkow [8] opens the prospect to improve on this situation in the future. It perfects (d) by generating code from Isabelle/HOL to CakeML [14] in a provably correct way (in the sense of mechanically checked proof). In turn, CakeML is a dialect of ML that comes with a verified compiler and runtime system, addressing the potential soundness issues of (e).

5 Conclusion and Future Work

We have presented Munta, a mechanically verified model checker for timed automata. As indicated in our discussion above, further efforts are conceivable to reduce the trusted code base. There are also several ways in which performance of the tool could be improved. One would be to verify the model checking

algorithms with respect to a fully imperative target language such as LLVM [10] or C. As another approach, we are studying certification of reachability checking for timed automata in ongoing work.

Finally, the capabilities of Munta could be improved by either enriching the modeling formalism as discussed above, or by providing a more expressive specification language for model checking properties, such as full (T)CTL or LTL. To this end, we plan to extend our work on certification towards LTL model checking in the future.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_25
3. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
4. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: a tool for BDD-based verification of real-time systems. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_13
5. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
6. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
7. Herbreteau, F., Point, G.: TChecker (2019). <https://github.com/fredher/tchecker>
8. Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 999–1026. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_35
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
10. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Proceedings of ITP 2019 (2019, to appear)
11. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_17
12. Lammich, P.: Parser combinator library for Isabelle/HOL (2018). https://bitbucket.org/MohammadAbdulaziz/planning/src/master/isabelle/Parser_Combinator.thy

13. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
14. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: International Conference on Functional Programming (ICFP), pp. 60–73. ACM Press, September 2016. <https://doi.org/10.1145/2951913.2951924>, invited to special issue of Journal of Functional Programming
15. Wang, F.: Efficient verification of timed automata with BDD-like datastructures. *Int. J. Softw. Tools Technol. Transf.* **6**(1), 77–97 (2004). <https://doi.org/10.1007/s10009-003-0135-4>
16. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_26
17. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4

E Verified Certification of Reachability Checking for Timed Automata

Original publication. This chapter was originally published as a full paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer and Joshua von Mutius. “Verified Certification of Reachability Checking for Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Proceedings, Part I*. ed. by Armin Biere and David Parker. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 425–443. DOI: 10.1007/978-3-030-45190-5_24

Synopsis. This paper presents a method for certifying unreachability in timed automata. Existing model checkers can be extended to emit a certificate proving unreachability. The certificate is then checked for validity by an independent tool, the certifier. Building upon the fully verified model checker MUNTA (Papers **C** and **D**), the certifier is also verified in Isabelle/HOL. The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practicality, using a new unverified model checker implementation in Standard ML to construct the certificates. Various techniques to compress certificates (by reducing the number of symbolic states they consist of) are proposed and experimentally evaluated.

Contribution. I have developed the theory for certification. I have constructed and verified the certifier in Isabelle/HOL. I have studied and experimented with the different compression techniques. Joshua von Mutius has implemented the unverified model checker MLUNTA. We have contributed to the evaluation to equal parts. I have written most of the paper.

Copyright notice. This article is an open-access publication, which was published under the Creative Commons Attribution 4.0 International License. It is reproduced on the following pages in its original form. The original publication can be found under the DOI cited above.



Verified Certification of Reachability Checking for Timed Automata

Simon Wimmer^{id} and Joshua von Mutius^{id}

Fakultät für Informatik,
Technische Universität München,
Munich, Germany

wimmers@in.tum.de joshua.von-mutius@tum.de



Abstract. Prior research has shown how to construct a mechanically verified model checker for timed automata, a popular formalism for modeling real-time systems.

In this paper, we shift the focus from verified model checking to certifying unreachability. This allows us to benefit from better approximation operations for symbolic states, and reduces execution time by exploring fewer states and by exploiting parallelism. Moreover, this gives us the ability to audit results of unverified model checkers that implement a range of further optimizations, including certificate compression.

The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practicality, using a new unverified model checker implementation in Standard ML to construct the certificates.

Keywords: Timed automata · Certification · Model Checking · Interactive Theorem Proving · Isabelle/HOL

Timed automata [1] are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL [4]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. As a consequence, one wants to ensure as rigorously as possible that the computation results of timed automata model checkers are correct.

Previous work [31] has addressed this problem by constructing a model checker for timed automata that is fully verified using Isabelle/HOL [25]. This tool is intended to be a reference implementation that can be used to scrutinize the correctness of other model checkers. As such, it is mainly able to check small and medium-sized benchmark examples, but the performance gap w.r.t. more practical model checkers prevents it from checking realistic benchmark models within reasonable time and space bounds.

We address this issue by shifting the focus from full verified model checking to only certifying that the result produced by an unverified model checker is correct. We only study reachability: it is the most important property that is checked with timed automata model checkers, and some model checkers only support reachability. It is crucial to ensure that a bad state is certainly not reachable if

the model checker claims so, thus we want to certify *unreachability*. Certifying that a state is indeed reachable would amount to extracting a timed trace and certifying that the trace is compatible with the model. While implementing this in a verified manner would be comparatively easy, we consider it less important because it corresponds to the bug finding functionality of model checkers, which carries less trust.

The recipe for certifying unreachability is simple: the model checker explores a number of states until it determines that there are no more states to be found. If none of the states fulfill the final state predicate (i.e. violates the safety property), then the model checker will answer “unreachable”. We use the set of explored states as the unreachability certificate. In essence, we only need to check that the initial state is contained in this set, that there are no outgoing edges from this set, and that none of the states in the set fulfill the final state predicate.

The switch to certification holds many advantages. Timed automata model checking uses *over-approximations* of symbolic states to ensure termination. A large variety of these approximation operators has been studied [2,3,14]. Our previous work [29] has shown that, while formally proving the correctness of these approximation operations is feasible in principle with an interactive theorem prover, the effort is rather high. Instead, to *certify* unreachability, it is sufficient to only know that the approximation operator indeed yields a state that is at least as big as the precise symbolic state. Certifying this property is cheap.

Moreover, certification eases *parallelization*. Checking that a state is not final and that all its successors are covered by the state set are local properties. We show how to exploit this in a verified implementation, while only mildly increasing the verification effort and the size of the trusted code base.

Finally, the number of states explored by a model checker can vary immensely, depending on a range of factors such as the chosen approximation operator or the search order. Thus, an efficient unverified tool can exploit different heuristics and strategies to compute a state space that is as small as possible, and thereby speedup the certification effort. In this context, we also study a number of *compression techniques* to reduce the number of states in the certificate after the model checker has concluded its search.

We use a new unverified model checker called Mlunta, which is implemented in Standard ML (SML), to generate certificates for a set of standard benchmarks, and to evaluate our verified certifier’s performance on these benchmarks ¹.

Related Work This work is based on an existing Isabelle/HOL formalization of timed automata model checking [29,31]. Other proof-assistant formalizations of timed automata focus on proving elementary properties about the basic formalism [33,34], or proving properties about concrete automata [26,10,8], but none of them are concerned with model checking.

Earlier work formalizes a model checker for the modal μ -calculus [28], and constructs a verified finite state LTL model checker [9,24,6].

¹ Both tools are available online: <https://doi.org/10.5281/zenodo.3679245>.

The idea of extracting certificates from the model checking process has previously been studied in the context of the μ -calculus [23] and finite state LTL model checking [27]. However, these works are not accompanied by a verified certificate checker and do not attempt to scale the approach to practical examples. Only the recent work of Griggio et al. [11] provides a practical extraction mechanism and a certificate checker for LTL model checking, but the checker is not verified. To the best of our knowledge, we are the first to examine certification in the context of timed automata model checking.

Finally, in the context of software verification, the idea of producing certificates for the correctness of a program has been broadly studied [16,5].

Isabelle/HOL Isabelle/HOL [25] is an interactive theorem prover based on Higher-Order Logic (HOL). HOL can be thought of as a combination of a functional programming language and mathematical logic. Isabelle/HOL mostly resembles standard mathematical notation. Some conventions that are borrowed from functional programming need to be explained, however. Functions are mostly curried, i.e. of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$ instead of $\tau_1 \times \tau_2 \Rightarrow \tau$. As a consequence, function application is usually denoted as $f a b$ instead of $f(a, b)$. Function abstraction with lambda terms uses the standard syntax $\lambda x. t$ (the function that maps x to t) and can also have paired arguments $\lambda(x, y). t$. Type variables are written $'a, 'b$, etc. Compound types are written in postfix syntax: $\tau \text{ set}$ is the type of sets of elements of type τ . We use the Isabelle/HOL convention that free variables are implicitly all-quantified throughout the paper. In parts of the paper, formulas or syntax have been simplified for readability, but we have stayed largely faithful to the Isabelle/HOL formalization.

Contributions In short, these are the main contributions of our work:

- To the best of our knowledge, we are the first to study certification of the model checking results of reachability checking for timed automata, including techniques to compress certificates.
- We construct a verified implementation of such a certificate checker, including a number of optimization techniques to make it practically usable.

Outline The remainder of the paper is organized as follows. The first section briefly recalls the theory of timed automata, and sketches the state-of-the-art model checking process. The second section details our approach to certification and explains how, starting from an abstract theory, a concrete verified implementation of the certificate checker can be obtained. Section three illustrates a number of techniques to improve the certificate checker’s performance, while only mildly increasing the formalization effort. Section four discusses two methods for certificate compression. The paper is concluded by an experimental evaluation and remarks on potential future work.

1 Timed Automata and Model Checking

Transition Systems We take a very simple view of transition systems: they are simply a relation \rightarrow of type $'a \Rightarrow 'a \Rightarrow \text{bool}$ for a type of states $'a$. We write $a \rightarrow^* b$ to denote that b can be reached from a via a sequence of \rightarrow -transitions.

Timed Automata To make the paper self-contained, this paragraph briefly describes timed automata and is mostly reproduced from Wimmer and Lammich [29]. For a thorough introduction see the tutorial paper of Bengtsson and Yi [4].

Compared to standard finite automata, timed automata introduce a notion of clocks. Figure 1 depicts an example of a timed automaton. We will assume that clocks are of type *nat*. A *clock valuation* u is a function of type $\text{nat} \Rightarrow \text{real}$. Locations and transitions are guarded by *clock constraints*, which have to be

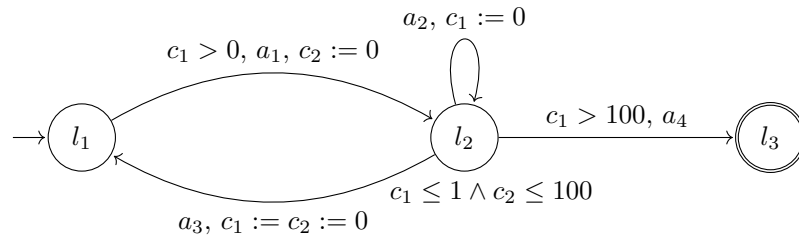


Fig. 1: Example of a timed automaton with two clocks.

fulfilled to stay in a location or to take a transition. Clock constraints are conjunctions of constraints of the form $c \sim d$ for a clock c , an integer d , and $\sim \in \{<, \leq, =, \geq, >\}$. We write $u \models cc$ if the clock constraint cc holds for the clock valuation u . We define a timed automaton A as a pair $(\mathcal{T}, \mathcal{I})$ where \mathcal{I} is a mapping from locations to clock constraints (also named invariants); and \mathcal{T} is a set of transitions written as $A \vdash l \xrightarrow{g,a,r} l'$ where l and l' are start and successor location, g is the guard of the transition, a is an action label, and r is a list of clocks that will be reset to zero when the transition is taken. States of timed automata are pairs of a location and a clock valuation. The operational semantics defines two kinds of steps (given as their HOL descriptions):

- Delay: $(l, u) \rightarrow^d (l, u \oplus d)$ if $d \geq 0$ and $u \oplus d \models \mathcal{I} l$;
- Action: $(l, u) \rightarrow_a (l', [r := 0]u)$
if $A \vdash l \xrightarrow{g,a,r} l'$, $u \models g$, and $[r := 0]u \models \mathcal{I} l'$;

where $u \oplus d = (\lambda c. u c + d)$ offsets all clocks by d in the valuation u , and $[r := 0]u = (\lambda c. \text{if } c \in r \text{ then } 0 \text{ else } u c)$ resets all clocks in r to 0 in valuation u . For any (timed) automaton A , we consider the transition system

$$(l, u) \rightarrow_A (l', u') = (\exists d \geq 0. \exists a u''. (l, u) \rightarrow^d (l, u'') \wedge (l, u'') \rightarrow_a (l', u')).$$

That is, each transition consists of a delay step that advances all clocks by some amount of time, followed by an action step that takes a transition and resets the clocks annotated to the transition. Given a final state predicate F and an initial state (l_0, u_0) , we are interested in whether $(l_0, u_0) \rightarrow_A^* (l, u)$ for any l, u with $F l$. In Figure 1, the final state is l_3 (i.e. $F l \iff l = l_3$). As the guard for action a_4 is never enabled, l_3 is unreachable.

Model Checking Due to the use of clock valuations, the state space of timed automata is inherently infinite. Thus, model checking algorithms for timed automata are based on the idea of abstracting from concrete valuations to *sets* of clock valuations of type $(nat \Rightarrow real)$ *set*, often called *zones*. The resulting transition system of reachable states from an initial zone is called the *zone graph*. It is explored in an *on-the-fly* manner, computing successors on zones, which are typically represented symbolically as *Difference Bound Matrices* (DBMs). Knowledge of this data structure is not necessary to understand the rest of the paper. Thus we refer the interested reader to Bengtsson and Yi [4] and to Wimmer and Lammich [29,31] for a verification of this data structure. In the remainder we will only use the term “zones” instead of referring to their implementation as DBMs.

The delicate part of this method is that the number of reachable zones could still be infinite. Therefore, over-approximations (or *abstractions*) of zones are computed to obtain a finite search space. For our purpose, it is sufficient to assume an abstraction operator α indeed computes an over-approximation, i.e. $Z \subseteq \alpha(Z)$ for any zone Z . We call the version of the zone graph where abstractions are applied the *abstract zone graph* [13]. For a number of such abstraction operators, it can be shown that the abstract zone graph is sound and complete². The proofs are rather intricate, however. Thus formalizing them would be a big effort. By focusing on certification of unreachability, this problem vanishes, as we only need to ensure that any state (l, Z) that we deem reachable in the zone graph is *subsumed* by some state (l, Z') with $Z \subseteq Z'$ that is part of the certificate and that was computed by the abstraction (i.e. $Z' = \alpha(Z_1)$ for some Z_1).

Certificates by Example Figure 2 depicts the zone graph of the automaton in Figure 1. Each zone Z is given as a clock constraint cc such that $Z = \{u \mid u \models cc\}$. A model checker like Munta would have to explore the full zone graph before being able to decide that l_3 is unreachable. Any model checker that uses the same abstraction technique as Munta [2] would not be able to benefit from abstractions for this example and thus the abstract zone graph is the same as the zone graph. However, such a model checker could apply subsumptions while exploring the zone graph. That is, when a symbolic state of the form $(l_2, \{u \mid u \models c_1 = 0 \wedge c_2 < k + 1\})$ is explored, the state $(l_2, \{u \mid u \models c_1 = 0 \wedge c_2 < k\})$ can safely be discarded.

This means that at the end of the model checking process, only the three states in Figure 3a will be stored. The solid edges are part of the zone graph,

² Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.

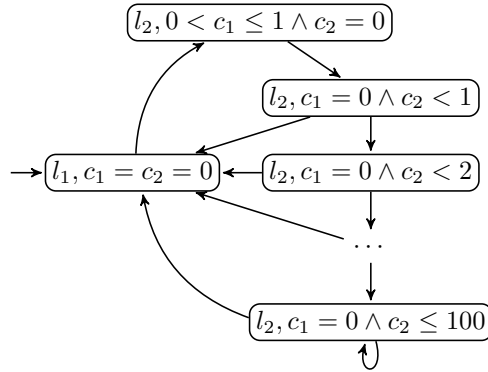


Fig. 2: The zone graph of the automaton depicted in Figure 1.

while the dashed edge indicates that the zone at its tail has a successor in the zone graph ($(l_2, \{u \mid u \models c_1 = 0 \wedge c_2 < 1\})$) that is subsumed by the tip of the edge. The set of these three states can act as a *certificate* of unreachability. They essentially form an inductive invariant of the zone graph: for each state in the certificate, all its successors in the zone graph are either contained in the certificate themselves or subsumed by another state in the certificate. Thus we know that any symbolic state that is reachable from the initial state is subsumed by some state in the certificate, and as the final state is not contained in the certificate, we can conclude that it is unreachable.

Figure 3b shows a certificate with only two states that replaces the two states for l_2 by the state with a dashed border. Note that this state is not part of the original zone graph. The certificate fulfills the same invariant property and thus also proves unreachability. We will use this technique of adding larger states to the certificate that are not part of the zone graph for our compression techniques in section 4.

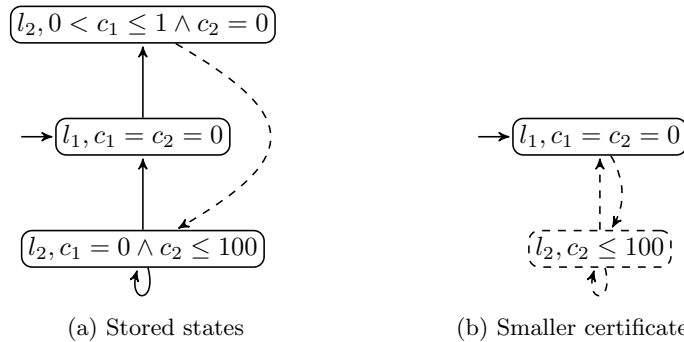


Fig. 3: Two certificates of unreachability for the automaton from Figure 1.

2 From Model Checking to Certifying Unreachability

This section first describes our approach to certification abstractly. Then, we detail how the existing formalization of a timed automata model checker was extended—with rather low effort—to a verified certifier. In practice, networks of timed automata with additional modeling features such as, e.g. shared state variables, are used. However, due to the existing verified product construction for such a formalism [31], it is sufficient to study the case of a single timed automaton here.

2.1 An Abstract Correctness Theorem

To work towards a rigorous justification of the certification process, we first study the problem on a more abstract level. Consider a transition system \rightarrow on states of type $'l \times 's$ where $'l$ corresponds to the finite state part of timed automata and $'s$ corresponds to zones. We assume an invariant P on states, i.e.:

$$P(l_1, s_1) \wedge (l_1, s_1) \rightarrow (l_2, s_2) \implies P(l_2, s_2).$$

This invariant essentially represents a restriction of \rightarrow to valid states. While this would usually be assumed implicitly, we explicate P here as it is technically more convenient to do so in the Isabelle/HOL formalization.

The interesting feature that sets timed automata model checking apart is subsumption. Recall that during the model checking process, it is possible to first discover some (symbolic) state (l, Z) (a pair of a discrete state l and a zone Z), and to find at some later point that another reachable state (l, Z') subsumes (l, Z) because Z' semantically contains Z , i.e. $Z \subseteq Z'$. At this point the state (l, Z) can be discarded as we know that anything that is reachable from (l, Z) is also reachable from (l, Z') . Abstractly, subsumption is modeled by some fixed preorder (i.e. a reflexive and transitive relation) \preceq on $'s$ which is a simulation relation between \rightarrow and itself:

$$\begin{aligned} & s_1 \preceq s_2 \wedge (l_1, s_1) \rightarrow (l_2, t_1) \wedge P(l_1, s_1) \wedge P(l_2, s_2) \\ \implies & \exists t_2. t_1 \preceq t_2 \wedge (l_1, s_2) \rightarrow (l_2, t_2) \end{aligned}$$

In the abstract setting, a certificate consists of a set of discrete states L of type $'l$ set, and a mapping M of type $'l \Rightarrow 's$ set that gives the set of reachable symbolic states that were computed for any discrete state $l \in L$. We say that (L, M) satisfies P if all states in the certificate (L, M) satisfy P :

$$l \in L \wedge s \in M l \implies P(l, s)$$

Moreover, the certificate needs to be *closed*. Following Herbreteau et al. [13], we call a state *covered* if it is subsumed by another state in the certificate. A certificate is closed if for each state in the certificate all its successors are covered:

$$l_1 \in L \wedge s_1 \in M l_1 \wedge (l_1, s_1) \rightarrow (l_2, s_2) \implies l_2 \in L \wedge (\exists s_3 \in M l_2. s_2 \preceq s_3) \quad (*)$$

The following key theorem states that all reachable states are covered if the initial state is covered:

Theorem 1. *Let (L, M) be closed and invariant under P . Assume $l_0 \in L$, $s'_0 \in M l_0$, $s_0 \preceq s'_0$, and $(l_0, s_0) \rightarrow^* (l, s)$. Then $l \in L$ and there exists s' such that $s' \in M l$ and $s \preceq s'$.*

Proof. By induction on the number of steps in $(l_0, s_0) \rightarrow^* (l, s)$. The following sketches how the run of covering states is constructed. The first line represents $(l_0, s_0) \rightarrow^* (l, s)$ and the states in the third line are all part of the certificate.

$$\begin{array}{ccccccc}
 (l_0, s_0) & \rightarrow & (l_1, s_1) & \rightarrow & \dots & \rightarrow & (l, s) \\
 & & \preceq & & & & \preceq \\
 & & (l_1, t_1) & & \dots & & (l, t) \\
 \preceq & \nearrow & \preceq & \nearrow & & \nearrow & \preceq \\
 (l_0, s'_0) & & (l_1, s'_1) & & \dots & & (l, s')
 \end{array}$$

From the assumptions on l_0 , s_0 , and s'_0 , we can first apply the self-simulation property of \rightarrow to $(l_0, s_0) \rightarrow (l_1, s_1)$ to obtain a t_1 such that $s_1 \preceq t_1$ and $(l_0, s'_0) \rightarrow (l_1, t_1)$. As the certificate is closed we thus get $l_1 \in L$ and we can find an $s'_1 \in M l_1$ such that $t_1 \preceq s'_1$ (and thus $s_1 \preceq s'_1$ by transitivity). The induction hypothesis can then be applied to l_1 , s_1 , and s'_1 . \square

We will now say that a certificate (L, M) is *admissible* iff

- it satisfies P ,
- it is closed,
- it covers the initial state (i.e. there is an $s'_0 \in M l_0$ such that $s_0 \preceq s'_0$),
- and there is no $l \in L$ with $F l$.

Corollary 1. *If F is monotone w.r.t. \preceq and the certificate (L, M) is admissible, then $\nexists l s. (l_0, s_0) \rightarrow^* (l, s) \wedge F l$.*

2.2 An Abstract Certificate Checker

In practice, the certification process has to consider one additional complication. A model is typically described in terms of human-readable identifiers, while most model checkers and the verified model checker Munta [30] in particular represent these as natural numbers internally to allow for efficient indexing. In our certifier, this is accounted for by relabeling the human-readable identifiers in a given model to natural numbers in a first (verified) pre-processing step. To save additional transformations of the certificate after it was emitted, we let the unverified model checker additionally emit a textual description of such a renaming. The certifier then just needs to check that the given renaming is injective to ensure that it can safely be applied.

Together with the theoretical analysis laid out in the last section, we can thus derive the following strategy for certifying unreachability:

- An unverified model checker explores the reachable state space of a given model symbolically and checks that none of the discovered states (l, s) fulfills $F l$.

```

1  definition check (L, M) ≡
2    monadic_list_all L (λl. do {
3      let S = M l;
4      let next = succs l S;
5      monadic_list_all next (λ(l', S'). do {
6        xs ← SPEC (λxs. set xs = S');
7        if xs = [] then return True else do {
8          b1 ← return (l' ∈ L);
9          ys ← SPEC (λxs. set xs = M l');
10         b2 ← monadic_list_all xs (λx.
11           monadic_list_ex ys (λy. return (x ≲ y))
12         );
13         return (b1 ∧ b2)
14       }
15     })
16 })

```

Listing 1.1: Monadic program to check whether a certificate is closed.

- The set of explored states is emitted as a certificate, possibly followed by compression (see section 4).
- The model, the final state predicate F , the certificate, and a description of the renaming that was used for the states are passed to the verified certifier.
- The certifier checks that the given renaming is injective, renames the model accordingly, applies the product construction and checks that the certificate is admissible.

If the process is successful, we can conclude by Corollary 1 that no “bad” state (l, s) (i.e. with $F l$) is reachable symbolically. We will argue that this really implies that the model is safe in the concrete case of timed automata in section 2.3.

We now lay out how a verified certificate checker that implements said strategy for an abstract transition system can be constructed in Isabelle/HOL. Listing 1.1 displays the definition of the core of the checker that checks whether the certificate is closed in the sense defined above. The program is defined in the non-determinism monad of the Imperative Refinement Framework (IRF) [20]. Some parts, such as checking set membership or converting a (finite) set to a list are still left abstract. A non-deterministic specification $\text{SPEC } Q$ returns some value v with $Q v$.

The body of the program (lines 2-16) iterates over all discrete states in the certificate L and checks that all corresponding symbolic states are covered. Line 3 retrieves the symbolic states that correspond to discrete state l and in line 4 their symbolic successor states are computed. The result ($next$) is a list of pairs of a discrete state and the set of its corresponding symbolic states. The loop ranging from lines 5 to 15 iterates over this list to ensure that all the successor states are covered. Given a discrete state l' and a set of symbolic states S' , line 6 first converts it into a list xs that can be iterated over. This turns into a vacuous

operation when the algorithm is refined to an executable version where sets are implemented as lists. Line 8 checks that l' is also part of the certificate. Then, in line 9 the set of corresponding symbolic states is retrieved and converted to a list ys . Finally, lines 10-12 ensure that all states in xs are subsumed by some state in ys .

To prove soundness of *check*, we mainly need correctness theorems for the monadic combinators *monadic.list_all* and *monadic.list_ex*. Given a list xs and a monadic implementation Q_i of a predicate Q , they check whether all states (at least one state) in xs satisfy (satisfies) Q . This is the correctness theorem for *monadic.list_all*, for instance:

$$\begin{aligned} & (\forall x. Q_i x \leq \text{SPEC } (\lambda r. r \longleftrightarrow Q x)) \\ \implies & \quad \text{monadic.list_all } xs \, Q_i \leq \text{SPEC } (\lambda r. r \longleftrightarrow \text{list_all } xs \, Q) \end{aligned}$$

where *list_all* $xs \, Q$ holds if and only if Q holds for all elements in xs . After setting up the IRF's verification condition generator with this rule and the corresponding rule for *monadic.list_ex*, it is easy to prove that *check* is sound:

$$\text{check } (L, M) \leq \text{SPEC } (\lambda r. r \implies \text{closed } (L, M))$$

where the property *closed* (L, M) corresponds to condition (*) from above.

We then use standard refinement techniques to obtain an algorithm *check_i* that refines *check*, replacing sets by lists. However, the algorithm is still specified in the non-determinism monad and therefore not executable. We use a simple technique to make it executable. Consider the following theorem for *monadic.list_all*:

$$\text{monadic.list_all } xs \, (\lambda x. \text{return } (P x)) = \text{return } (\text{list_all } xs \, P).$$

It allows us to replace the non-deterministic combinator *monadic.list_all* by the deterministic *list_all*, pushing *return* to the outside. By exhaustively applying a set of such rewrite rules we obtain an alternative definition of *check_i* where *return* appears only on the outermost level, and the inner term is deterministic and thus executable. Using these techniques, we obtain a simple certificate checker that is executable, provided that we can implement the elementary model checking primitives such as the subsumption check or computing the list of successors of a state.

2.3 Transferring the Correctness Theorem

For timed automata, the abstract transition system studied above is the zone graph $\rightarrow_{ZG(A)}$ of a given (single) automaton A . One can show that it simulates \rightarrow_A (completeness of $\rightarrow_{ZG(A)}$):

$$(l, u) \rightarrow_A (l', u') \wedge u \in Z \implies (\exists Z'. (l, Z) \rightarrow_{ZG(A)} (l', Z') \wedge u' \in Z').$$

This simulation property is sufficient to establish that if there is no reachable state (l, Z) in $\rightarrow_{ZG(A)}$ with $F l$, then no final state (l, u) is reachable in \rightarrow_A :

$$\begin{aligned} & (\nexists l, Z. (l_0, Z_0) \rightarrow_{ZG(A)}^* (l, Z) \wedge F l) \wedge u_0 \in Z_0 \\ \implies & \quad (\nexists l, u. (l_0, u_0) \rightarrow_A^* (l, u) \wedge F l) \end{aligned}$$

In the formalization, these proofs rely on instantiating a general theory of simulations in transition systems that is derived from the theory of Wimmer and Lammich [31]. From Corollary 1 we get that there is no reachable final state in $\rightarrow_{ZG(A)}$ if the certificate check is passed. Finally, by correctness of the renaming process and the product construction, we can conclude that there is no final reachable state in the input model if there is no final reachable state in \rightarrow_A .

2.4 Implementing a Concrete Checker

All the elementary model checking primitives we need for certification have already been implemented [31]. The abstract implementation presented above assumes that the model checking primitives are implemented in a purely functional manner (as they are just regular HOL functions). The existing (verified) model checker [31], however, is an imperative implementation in the Imperative HOL framework. Imperative HOL [7] is a framework for specifying and reasoning about imperative programs in Isabelle/HOL. It provides a *heap monad* in which one can use—analogously to the ML family of programming languages—imperative references and arrays to express imperative programs. Usually, once we have used an imperative implementation anywhere, the whole program would need to be stated in the heap monad. However, we can employ a technique similar to the one that is used for Haskell’s *ST monad* [21] to erase the heap monad in a safe way under certain circumstances.

More precisely, if it can be deduced from the type of an imperative computation that no information about references or arrays on the heap can be leaked to the outside of the computation in its result, then the heap monad can be erased for this computation, yielding a pure computation. In the certifier, this is primarily used for computing the symbolic successor of a zone Z for a certain transition. To that end, an immutable representation of the DBM M corresponding to Z is copied to the a newly allocated imperative array, then the imperative pipeline of computations to compute the successor M' is applied to M , and finally M' is copied back to an immutable array. Taken together, this whole computation does not contain the type of an array or reference in its result type, and thus can safely be turned into a pure computation. As a consequence, we are able to reuse the existing verified model checking primitives, while being able to state the certificate checking algorithm purely functionally.

In the concrete checker, the mapping M is implemented using a verified functional hash table implementation based on so-called *diff arrays* [19]. This data structure provides a purely functional interface to an underlying imperative array. When a diff array is updated, it performs the update on the imperative array, and stores a difference that can be used to re-compute the old state of the array. Reading from the most recent version of a diff array is fast as the value can directly be read from the underlying imperative array. If an old version is accessed, the whole array has to be copied to recompute the old version. This gives diff arrays good performance characteristics, as long as they are mostly used linearly. This is the case in our application as the hash table is filled in an initial phase, after which the hash table is used in a read-only manner.

2.5 Parallel Execution

The attentive reader may wonder why we care about a purely functional implementation of the certificate checker at all. Indeed, we could use existing techniques [31] to obtain an imperative implementation of the certificate checker in the heap monad. However, in this setting it would be hard to justify the soundness of executing parts of the checker in parallel. In the purely functional setting, this is much simpler. Our approach to parallel execution is minimalist: we only provide means to execute the *map* combinator on lists in parallel. This is achieved by another custom code translation that is part of the trusted code base. The parallel implementation of *map* uses a task queue that will contain the individual computations that need to be run for each element of *xs*, and uses a fixed number of threads to work through this list and assemble the final result.

We exploit this *map* implementation to work through the list of discrete states *L* in parallel, using the equivalence:

$$\text{list_all } Q \text{ } xs = \text{list_all id } (\text{map } Q \text{ } xs).$$

In doing so, we lose the ability to stop execution early once a list element does not satisfy *Q*. For the certificate checker, however, we assume that usually the certificate is correct, meaning that we have to go through the whole list anyway. We only parallelize the outermost loop of *check_i* because this should yield reasonably-sized work portions, given that the size of *L* will typically at least be in the hundreds.

3 Scaling Performance

In this section we discuss two techniques to improve the performance of the certificate checker without increasing the verification effort significantly.

3.1 Monomorphization

Isabelle/HOL supports polymorphism and type classes, which are valuable features for sizeable formalization efforts. Large parts of our formalization also make use of these features, e.g., most of the timed automata semantics are formalized for a general time domain, and operations on DBMs are applicable on DBMs whose entries are formed from more general algebraic structures than the ring of integers. While this yields an abstract and general formalized theory, it can get in our way when trying to obtain efficient code.

When generating SML code from HOL, Isabelle uses a so-called dictionary construction to compile out type classes, which are not supported by SML. This means that most functions carry a large number of additional parameters, which are used to look up elementary operations, such as addition of two numbers. These additional lookup operations degrade performance. One solution is to ensure that all relevant constants that are exported to SML are monomorphic (i.e. specialized to the integer type), eliminating the need for the dictionary construction in most places. Thus, we apply a semi-automated procedure to achieve this monomorphization.

3.2 Integer Representation

Types such as *int* or *nat* are unbounded in Isabelle/HOL meaning they are implemented with the help of big integers in the target languages. To improve performance, we want to use machine integers instead, and instruct Isabelle/HOL's code generator to do that. This is still sound: SML's standard integer operations throw an exception if an overflow occurs instead of silently wrapping around. The code generator can only achieve partial correctness anyway: if program execution does not fail, then its result is consistent with the evaluated HOL term.

3.3 Refined Code Equations

The last type of optimizations we use can be considered to belong to the category of micro-optimizations. These are improved code generator translations for elementary operations and combinators. We employ such improved translations to use native implementation language primitives to convert from mutable to immutable arrays and back. The other such optimizations we use, is to directly use integer values as counters in imperative loops instead of a natural number representation that would box the integers in a data constructor. In the same way, we use integers directly for array indexing.

4 Certificate Compression

In this section, we present two techniques to compress the unreachability certificate. By compression we mean reducing the number of zones that are present in the certificate for each discrete state, using the unverified model checker. The first technique relies on subsumption. As explained above, it is possible that the model checker adds a zone Z to the set of explored states and later another zone Z' with $Z \subseteq Z'$ (i.e. Z' subsumes Z). Thus the first technique simply filters the set w.r.t. \subseteq in the end.

The second technique relies on the following idea: we replace one or more zones by their union, and check that the state space is still closed. This means that we have to check that all the successors of the larger zone are still covered by the current set of states. In that case, we can discard the old zones, and replace them by their union. As the union of two zones is not necessarily convex and thus cannot be represented as a DBM, we do not compute a precise union of zones but their convex hull. This operation is rather cheap as it amounts to taking the pointwise maximum of DBM entries. After computing the convex hull of a number of zones (in canonical form), we only need to apply the expensive operation to restore a canonical form once.

The latter technique yields a whole family of compression algorithms by iterating one of the following operations for each discrete state until a fixed-point is reached:

- a) the convex hull of all zones is computed;
- b) the convex hull of the first two zones is computed;

- c) the convex hull of the first two zones that can successfully be joined is put to the front of the list;
- d) same as c) but considering only discrete states for which compression was successful in the last round;
- e) same as d) but iterating the operation until saturation.

The next section contains an experimental evaluation of these techniques.

Note that similar techniques for reducing the search-space could also be applied already during model checking. By doing so, the number of states explored and the runtime of model checking could be reduced. This, however, comes at the risk of producing spurious model checking results (i.e. a final state might be deemed reachable, although there is no corresponding reachable state in the timed automaton).

5 Experimental Evaluation

We evaluate the checker on a set of benchmarks that is derived from UPPAAL’s standard benchmark suite [22]. Additionally, to cover the advanced modeling features of committed locations and broadcast channels, we use a set of benchmarks that is derived from the pacemaker models of Jiang et al. [17] and a modified version of the FDDI benchmark with broadcast channels. A prototype SML implementation of a timed automata model checker (Mlunta) is used to compute the certificates. We use reachability properties of the form $\mathbf{E}\diamond \text{false}$ to enforce that the model checker explores the complete state space. The results are given in Table 1. The problem size is specified as the number of automata in the network. We report the total runtime (wall time) of:

1. the tandem consisting of Mlunta (using the first compression technique) and the (verified) certificate checker, both compiled with MLton;
2. the individual runtime of the (verified) certificate checker for a varying number of threads for parallel computation, compiled with Poly/ML as it is the only SML compiler that supports multi-threading;
3. the runtime of UPPAAL configured for depth-first search (like Mlunta);
4. the runtime of an unverified SML implementation of the certificate checker based on Mlunta (compiled with MLton);
5. and the runtime of the fully verified model checker Munta [31] extended with the improvements from sections 2 and 4 and compiled with MLton.

As can be seen from the results, the tandem is still one order of magnitude slower than UPPAAL, but certificate checking in isolation is also up to one order of magnitude faster than the previous verified model checker [31]. Note that Mlunta explores significantly more states than UPPAAL and Munta for “Pacemaker”. Multi-core scale beyond two threads is relatively unsatisfactory, however. In micro-benchmarks, we have identified that the problem appears to be with memory allocation on the heap, even if no data is shared among threads (in our case, only the certificate is shared but successors are computed locally). There does not

seem to be an obvious way to improve on this situation for SML implementations. Finally, one can see that the verified certifier is not drastically slower than the unverified implementation based on Mlunta, indicating that the verified certifier is not missing any obvious significant optimizations.

Model	Size	UPPAAL	Tandem	Munta	Unverif.	Certifier for #threads				
						1-MLton	1	2	3	4
FDDI	8	0.33	0.79	1.01	0.14	0.21	0.99	0.64	0.57	0.53
	10	5.93	1.77	2.50	0.40	0.45	2.13	1.36	1.26	1.20
	12	92.66	3.93	5.42	0.90	1.20	3.41	2.33	2.25	2.18
	14	1874.28	7.28	10.73	1.86	2.22	5.36	3.94	3.90	3.88
	16	***	12.80	19.51	3.47	3.72	11.09	6.49	6.50	6.51
FDDI broad	8	0.34	0.28	1.07	0.10	0.08	0.26	0.19	0.17	0.16
Fischer	5	0.24	2.78	6.33	0.72	0.53	1.76	1.07	0.98	0.91
	6	34.74	143.72	377.70	40.99	26.58	40.60	25.47	24.16	23.67
CSMA	5	0.04	0.94	4.42	0.31	0.28	1.44	0.87	0.80	0.76
	6	1.53	13.48	65.16	5.24	4.18	12.16	8.04	7.87	7.76
Mode										
Pacemaker	1	0.02	0.16	0.37	0.03	0.03	0.25	0.16	0.14	0.13
	2	0.02	0.75	3.20	0.17	0.26	1.23	0.75	0.68	0.65
	3	0.03	1.39	4.23	0.34	0.46	2.53	1.55	1.40	1.31
	4	0.02	11.80	0.70	3.24	3.71	12.13	8.38	6.69	6.66
	5	0.02	30.84	0.86	9.13	10.07	26.60	18.58	18.15	17.95

Table 1: Benchmarks results on a machine with 16 GB RAM and an Intel(R) Core(TM) i7-4610M CPU at 3.00GHz with two cores and two threads per core. The column labeled “Tandem” gives the runtime for a combination of the unverified SML tool and the verified certificate checker. The next column gives the runtime of the unverified SML certifier, followed by the runtimes of the verified checker for a varying number of threads. All times are given in seconds.

Table 2 gives the results of evaluating the different compression algorithms on the same set of benchmarks. The second variant is always applied to the compression result of the first variant to avoid trivial computations of the convex hull. Variant 2c) (the most expensive one) can produce drastically smaller certificates than any other variant, and its minimum compression factor is an order of a magnitude higher than for any other variant. Nevertheless, only variants 1 and 2a) appear to be useful in practice, as they are relatively cheap to compute. The other variants could prove useful if the certificates were produced by a significantly more efficient model checker, such as UPPAAL or TChecker [15]. On a final note, we have constructed a more than 95% smaller but valid certificate for the Fischer benchmark, suggesting that there is room for improvement on the compression algorithms.

Model	Size	Variant					
		1	2a	2b	2c	2d	2e
FDDI	8	0.21	0.21	1.72	69.53	3.65	3.43
FDDI broadcast	8	0.00	48.94	48.94	48.94	1.06	1.06
Fischer	5	22.03	22.03	22.72	43.06	30.40	30.40
CSMA/CD	5	26.06	41.54	43.84	81.16	58.94	47.54
	6	24.86	41.91	44.02	88.35	63.24	47.02
Mode							
Pacemaker	1	16.07	25.00	30.80	58.04	29.02	29.02
	2	24.00	26.38	30.37	58.68	35.87	35.22
	3	12.96	17.62	19.23	46.92	25.30	25.01
	4	13.82	20.02	23.60	41.48	26.16	24.71
	5	17.14	22.48	25.46	39.69	28.18	26.88
Average		15.71	26.61	29.07	57.58	30.18	27.03

Table 2: Certificate compression factors (given in %).

6 Conclusion and Future Work

We have presented a verified certifier of unreachability certificates for a timed automata. The certificates are ought to be produced by an unverified model checker. Experimentation shows that verified certificate checking in isolation is up to an order of magnitude faster than what was previously possible with a verified model checker [31]. The performance of a tandem of an unverified model checker and the verified certifier could be improved by replacing the certificate-producing part with a highly optimized tool, possibly opening room to use some of the more powerful certificate compression techniques we suggested above. As we pointed out above, there appears to be further room for improvement on the certificate compression algorithms as well.

Moreover, more sophisticated tools also employ more powerful abstraction techniques, for which our proposed certification technique is still suitable—to a large extent without requiring additional verification effort. An exception is the implicit abstraction technique studied by Herbreteau et al. [14] as it does not compute abstractions of zones explicitly but rather checks subsumptions of the form $Z \subseteq \alpha(Z')$ implicitly, meaning that one would have to prove correctness of the subsumption check to validate certificates produced by such a model checking process.

Finally, we intend to extend this work to certification of emptiness of timed Büchi automata in the future, using the idea of *subsumption graphs* [13] and relying on an unverified model checker implementation for timed Büchi automata to produce the certificates [13,18].

Data Availability Statement

The datasets generated and/or analyzed during the current study are available in the Zenodo repository [32]: <https://doi.org/10.5281/zenodo.3679245>. The artifact has been tested on the TACAS artifact evaluation VM [12].

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. pp. 254–270. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_18
3. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podolski, A. (eds.) TACAS 2004. pp. 312–326. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_25
4. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, vol. 3908, pp. 87–124. Springer (2004). https://doi.org/10.1007/978-3-540-27755-2_3
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: FSE 2016. p. 326–337. Association for Computing Machinery, New York, NY, USA (2016). <https://doi.org/10.1145/2950290.2950351>
6. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. *Journal of Automated Reasoning* **60**(1), 3–21 (2018). <https://doi.org/10.1007/s10817-017-9418-4>
7. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2008). pp. 134–149 (2008). https://doi.org/10.1007/978-3-540-71067-7_14
8. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLs 2001: Supplemental Proceedings (2001)
9. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer (2013)
10. Garnacho, M., Bodeveix, J., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: FORMATS 2013. pp. 106–120. LNCS 8053 (2013). https://doi.org/10.1007/978-3-642-40229-6_8
11. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. 2018 Formal Methods in Computer Aided Design (FMCAD) pp. 1–9 (2018). <https://doi.org/10.23919/FMCAD.2018.8603022>
12. Hartmanns, A., Seidl, M.: tacas20ae.ova (Aug 2019). <https://doi.org/10.6084/m9.figshare.9699839.v2>, https://figshare.com/articles/tacas20ae_ova/9699839/2
13. Herbreteau, F., Srivathsan, B., Tran, T.T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) FSTTCS 2016. LIPIcs, vol. 65, pp. 48:1–48:14. Schloss Dagstuhl (2016). <https://doi.org/10.4230/LIPIcs.FSTTCS.2016.48>

14. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. *Information and Computation* **251**, 67–90 (2016). <https://doi.org/10.1016/j.ic.2016.07.004>
15. Herbreteau, F., Point, G.: TChecker (2019), <https://github.com/fredher/tchecker>
16. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: SPIN 2014. p. 30–39. Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2632362.2632372>
17. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. pp. 188–203. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_14
18. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. pp. 968–983. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
19. Lammich, P.: Collections framework. *Archive of Formal Proofs* (Nov 2009), <http://isa-afp.org/entries/Collections.html>, Formal proof development
20. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015, Proceedings. LNCS, vol. 9236, pp. 253–269. Springer (2015). https://doi.org/10.1007/978-3-319-22102-1_17
21. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. *PLDI 1998* **29** (07 1998). <https://doi.org/10.1145/178243.178246>
22. Möller, M.O.: UPPAAL benchmarks (2017), <https://www.it.uu.se/research/group/darts/uppaal/benchmarks>
23. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. pp. 2–13. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2
24. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) *Verified Software: Theories, Tools and Experiments*. pp. 105–114. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_7
25. Nipkow, T., Lawrence C. Paulson, Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). <https://doi.org/10.1007/3-540-45949-9>
26. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: STACS 2001. pp. 298–315. LNCS 2215 (2001). https://doi.org/10.1007/3-540-45500-0_15
27. Peled, D., Pnueli, A., Zuck, L.: From falsification to verification. In: Hariharan, R., Vinay, V., Mukund, M. (eds.) FSTTCS 2001. pp. 292–304. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-45294-X_25
28. Sprenger, C.: A verified model checker for the modal μ -calculus in Coq. In: TACAS 1998. pp. 167–183. Springer, London, UK (1998). <https://doi.org/10.1007/BFb0054171>
29. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer (2016). https://doi.org/10.1007/978-3-319-43144-4_26
30. Wimmer, S.: Munta: A fully verified model checker for realtime systems. <https://github.com/wimmers/munta> (2019)
31. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4

32. Wimmer, S., von Mutius, J.: Artifact for "Verified Certification of Reachability Checking for Timed Automata" (Feb 2020). <https://doi.org/10.5281/zenodo.3679245>, <https://doi.org/10.5281/zenodo.3679245>
33. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: Proc. of the IASTED International Conference on Software Engineering, 2006. pp. 107–112 (2006)
34. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: SNPD 2009. pp. 555–560 (2009). <https://doi.org/10.1109/SNPD.2009.69>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



F Certifying Emptiness of Timed Büchi Automata

Original publication. This chapter was originally published as a full paper in the proceedings of a peer-reviewed conference as:

Simon Wimmer, Frédéric Herbreteau, and Jaco van de Pol. “Certifying Emptiness of Timed Büchi Automata.” In: *Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020*. Vol. 12288 LNCS. Lecture Notes in Computer Science. Springer, 2020, pp. 58–75. DOI: 10.1007/978-3-030-57628-8_4

Synopsis. Building on the work for certifying unreachability in timed automata (Paper E), this work studies certificates for the emptiness of timed Büchi automata, which are useful for model checking liveness properties of timed automata. The certificates can be extracted from practical model checkers based on different algorithms. The certificates can be validated by an independent certifier, which was formally verified in Isabelle/HOL. Certificates for the emptiness of timed Büchi automata are studied in an abstract setting and it is proved that the proposed approach is sound and complete. It is shown that the approach can also be made compatible with the model checking technique of implicitly exploiting abstractions in subsumptions. To also demonstrate feasibility of the approach, certificates for several models checked by the tools TChecker and Imitator were extracted and validated with the verified certifier.

Contribution. I have contributed most of the theory for the certification approach. With Jaco van de Pol I have researched our specific use of topological numberings in the certificates. With Frédéric Herbreteau I have studied the use of implicit abstraction techniques. I have formalized the theory in Isabelle/HOL and verified the certifier. We have all contributed to the evaluation to equal parts. I have written most of Sections 3, 4, and 6.

Copyright notice. On the following pages the full article is reprinted by permission from Springer Nature Customer Service Centre GmbH. The original publication can be found under the DOI cited above.

Note. An extended version of the article including proof sketches was published on arXiv:

Simon Wimmer, Frédéric Herbreteau, and Jaco van de Pol. “Certifying Emptiness of Timed Büchi Automata.” In: *CoRR* abs/2007.04150 (2020). arXiv: 2007.04150



Certifying Emptiness of Timed Büchi Automata

Simon Wimmer¹, Frédéric Herbreteau², and Jaco van de Pol³

¹ Fakultät für Informatik, Technische Universität München, Munich, Germany
wimmers@in.tum.de

² Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, 33400 Talence, France

³ Department of Computer Science, Aarhus University, Aarhus, Denmark

Abstract. Model checkers for timed automata are widely used to verify safety-critical, real-time systems. State-of-the-art tools achieve scalability by intricate abstractions. We aim at further increasing the trust in their verification results, in particular for checking liveness properties. To this end, we develop an approach for extracting certificates for the emptiness of timed Büchi automata from model checking runs. These certificates can be double checked by a certifier that we formally verify in Isabelle/HOL. We study liveness certificates in an abstract setting and show that our approach is sound and complete. To also demonstrate its feasibility, we extract certificates for several models checked by TChecker and Imitator, and validate them with our verified certifier.

Keywords: Timed automata · Certification · Model checking

1 Introduction

Real-time systems are notoriously hard to analyze due to intricate timing constraints. A number of model checkers for timed automata (TA) [1] have been implemented and successfully applied to the verification of safety-critical timed systems. Checking liveness properties of timed automata has revealed to be particularly important, as emphasized by a bug in the standard model of the CSMA/CD protocol that has been discovered only recently [16]. Several algorithms have been implemented to scale the verification of liveness specifications to larger systems [16, 22, 26, 32, 33]. Users of timed automata model checkers put a high amount of trust in their verification results. However, as verification algorithms get more complex, it becomes highly desirable to justify the users' confidence in their correctness.

There are two main approaches to ensure high degrees of trustworthiness of automated tools: verification and certification. In the first approach, correctness of the verification tool (its implementation and its theory) is proved using another semi-automated method. This technique has been applied to model checkers [13, 34] and SAT solvers [7]. In the second approach, the automated

tool produces a certificate, i.e. a proof for its verification result. Then an independent tool, the certifier, checks that the proof is indeed valid. In the best case, the certifier itself is formally verified. Examples include SAT certificate checking [20, 23] and unreachability checking of TA [37].

The certification approach promises many advantages over verification, since certificate checking is much simpler than producing the certificate. This drastically reduces the burden of semi-automated verification, which is a laborious task. While proving correctness of a competitive verification tool might be prohibitively complicated, it may be feasible to construct an efficient verified certifier instead (in the case of SAT [23], the verified certifier was even faster than the original SAT solvers). Finally, there is a wide variety of model checking algorithms and high-performance implementations, which are suited for different situations. Instead of verifying them one by one, these tools could produce certificates in a common format, so they can be checked by a single verified certifier.

1.1 Related Work

Model checking LTL properties for timed automata [1, 16, 22, 26, 32, 33] consists of three conceptual steps: the LTL formula is transformed into a Büchi automaton, the semantics of the TA is computed as a (finite) zone graph, and the cross-product of these objects is checked for accepting cycles. The two main alternative algorithms for detecting accepting cycles are Nested Depth-First Search (NDFS) and the inspection of the Strongly Connected Components (SCC). The NDFS algorithm was generalized to TA in LTSmin [21, 22] and extended to parametric TA in Imitator [2, 28]. The SCC-based algorithm has also been generalized to TA in TChecker [16, 19]. Both algorithms support *abstraction and subsumption between states* to reduce the state space.

Verified Model Checking. An early approach targeted the verification of a μ -calculus model checker in Coq [27]. The NDFS algorithm was checked in the program verifier Dafny [25, 31], while a multi-core version of it was checked in the program verifier Vercors [8, 30]. A complete, *executable* LTL model checker was verified in the interactive theorem prover Isabelle/HOL [13] and later extended with partial-order reduction [9]. A verified model checker for TA, Munta [34], has also been constructed in Isabelle/HOL [29, 36].

Certification. A certifier for reachability properties in TA has been proposed very recently [37]. A certification approach for LTL model checking was proposed in [15]. It uses k-liveness to reduce the problem to IC3-like invariant checking.

Contributions. In this paper, we extend certificates for unreachability of TA [37] to certificates for liveness properties, i.e. emptiness of timed Büchi automata (TBA). We propose a common certification approach for tools using different algorithms and various abstractions [16, 22]. These certificates can be much smaller than the original state space, due to the use of subsumption and abstraction. The difficulty here is that a careless application of subsumption can introduce spurious accepting cycles. Our new contributions are¹:

¹ An artifact containing our code and benchmarks is available on [figshare](#) [35].

- We introduce an abstract theory for certificates of Büchi emptiness, which can be instantiated for zone graphs of TBA with subsumptions.
- We developed a fully, mechanically verified certifier in Isabelle/HOL. In particular, our certifier retains the ability to check certificates in parallel.
- We show that the previous certifier for reachability and our extension to Büchi emptiness are compatible with implicit abstraction techniques for TA.
- We demonstrate feasibility by generating and checking certificates for two external model checkers, representing the NDFS and the SCC approach.

Note that checking counter-examples is easy in practice, but checking “true” model checking results is much harder. This is exactly what we address with certifying emptiness of TBA. The main application would be to increase the confidence in safety-critical real-time applications, which have been verified with an existing model checker. Another possible application of the certifier would be to facilitate a new model checking contest for liveness properties of TA.

2 Timed Automata and Model Checking

In this section, we set the stage for the rest of the paper by recapitulating the basic notions of TA and summarizing the essential concepts of TBA verification.

2.1 Verification Problems for Timed Automata

A TA $A = (Q, q_0, F, I, T, X)$ is a finite automaton extended with a finite set of *clocks* X . Q is a finite set of states with initial state $q_0 \in Q$ and accepting states $F \subseteq Q$. I associates an *invariant* constraint to every state and T associates a *guard* constraint g and *clock reset* $R \subseteq X$ to each transition. Here (*clock*) *constraints* are conjunctions of formulas $x\#c$, where x is a clock, $c \in \mathbb{N}$ and $\# \in \{<, \leq, =, \geq, >\}$. Observe that we exclude diagonal constraints of the form $x - y\#c$. An example of a timed automaton is depicted in Fig. 1.

A clock valuation $v : X \rightarrow \mathbb{R}_{\geq 0}$ associates a non-negative real value to each clock $x \in X$. A configuration is a pair (q, v) where q is a state and v is a clock valuation. The initial configuration is $(q_0, \mathbf{0})$. Without loss of generality, we assume that the initial clock valuation $\mathbf{0}$ satisfies the invariant $I(q_0)$. There are two kind of steps from a configuration (q, v) :

delay $(q, v) \rightarrow_\delta (q, v')$ for a delay $\delta \in \mathbb{R}_{\geq 0}$ if for every clock $x \in X$, $v'(x) = v(x) + \delta$, and v' satisfies the invariant $I(q)$;

transition $(q, v) \rightarrow_t (q', v')$ for transition $t = (q, g, R, q') \in T$ if v satisfies the guard g , $v'(x) = 0$ if $x \in R$ and $v'(x) = v(x)$ otherwise, and v' satisfies $I(q')$.

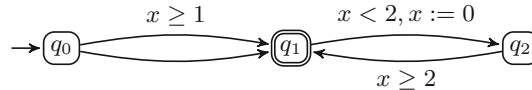


Fig. 1. Timed (Büchi) automaton with initial state q_0 and accepting state q_1 .

We write $(q, v) \rightarrow_{\delta, t} (q', v')$ if there exists a configuration (q, v'') such that $(q, v) \rightarrow_{\delta} (q, v'') \rightarrow_t (q', v')$. A run of a timed automaton is an (infinite) sequence of transitions of the form: $(q_0, \mathbf{0}) \rightarrow_{\delta_0, t_0} (q_1, v_1) \rightarrow_{\delta_1, t_1} \dots$. A run is *non-Zeno* if the sum of its delays is unbounded.

The *reachability problem* asks, given a timed automaton A , if there exists a finite run from the initial configuration $(q_0, \mathbf{0})$ to an accepting configuration (q_n, v_n) such that $q_n \in F$.

In timed Büchi automata (TBA), F is interpreted as a Büchi acceptance condition. The *liveness problem* then asks, whether a given TBA A is non-empty, i.e. if there is an infinite non-Zeno run from the initial configuration $(q_0, \mathbf{0})$ that visits infinitely many accepting configurations (q_i, v_i) with $q_i \in F$. In this paper, we work under the common assumption that TA only admit non-Zeno runs (see [33] for a construction to enforce this on every TA).

Both problems are known to be PSPACE-complete [1]. Due to density of time, these two verification problems cannot be solved directly from the transition system induced by configurations and steps. A well-known solution to this problem is the region graph construction of Alur and Dill [1]. Yet, it is not used in practice, as the region graph is enormous even for rather simple automata.

2.2 Zone Graph and Abstractions

The practical solution that is implemented in state-of-the-art tools like UPPAAL [24], TChecker [19] and the Imitator tool [2] is based on zones. Let us fix a set of clocks X . A zone Z is a set of valuations represented as a conjunction of constraints of the form $x \# c$ or $x - y \# c$ for $x, y \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{Z}$. Zones can be efficiently represented using Difference Bound Matrices (DBMs) [12]. Moreover, zones admit a canonical representation, hence equality and inclusion of two zones can be checked efficiently [6].

We now define the symbolic semantics [11] of a TA A . Let q, q' be two states of A , and let $W, W' \subseteq \mathbb{R}_{\geq 0}^X$ be two non-empty sets of clock valuations. We have $(q, W) \Rightarrow^t (q', W')$ for some transition $t \in T$, if W' is the set of all clock valuations v' for which there exists a valuation $v \in W$ and a delay $\delta \in \mathbb{R}_{\geq 0}$ such that $(q, v) \rightarrow_{\delta, t} (q', v')$. In other words, W' is the strongest postcondition of W along transition t . The symbolic semantics of A , denoted by \Rightarrow , is the union of all \Rightarrow^t over $t \in T$. The symbolic semantics is a sound and complete representation of the finite and infinite runs of A . Indeed, A admits a finite (resp. infinite) run $(q_0, v_0) \rightarrow_{\delta_0, t_0} (q_1, v_1) \rightarrow_{\delta_1, t_1} \dots (q_n, v_n) \rightarrow_{\delta_n, t_n} \dots$ if and only if there exists a finite (resp. infinite) path $(q_0, W_0) \Rightarrow^{t_0} (q_1, W_1) \Rightarrow^{t_1} \dots (q_n, W_n) \Rightarrow^{t_n} \dots$ such that $v_i \in W_i$ for all $i \geq 0$ and $W_0 = \{\mathbf{0}\}$ [11]. It is well-known that if Z is a zone, and $(q, Z) \Rightarrow (q', W')$ then W' is a zone as well [6]. Since $\{\mathbf{0}\}$ is a zone, all the reachable nodes in \Rightarrow are zones as well. The reachable part of \Rightarrow is called the *zone graph* of A . The nodes of the zone graph are denoted as (q, Z) in the sequel and the zone graph is simply denoted by its transition relation \Rightarrow . Figure 2a depicts the zone graph of the automaton in Fig. 1.

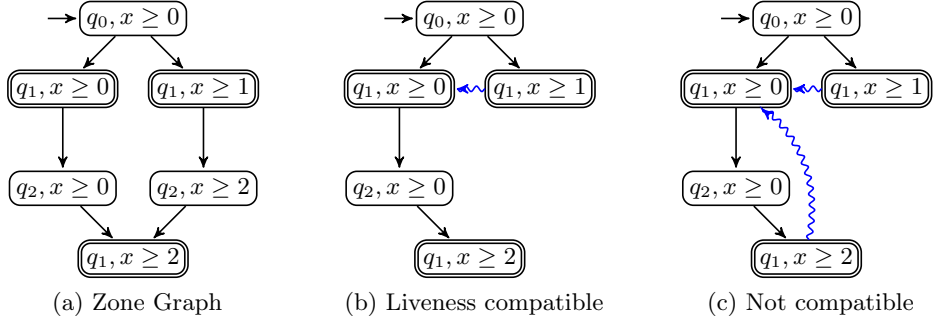


Fig. 2. Three subsumption graphs for the automaton in Fig. 1.

Still, the zone graph of a timed automaton may be infinite. As a remedy, finite abstractions have been introduced in the literature [4, 5, 11].

An *abstraction* α transforms a zone Z into a zone $\alpha(Z)$ such that $Z \subseteq \alpha(Z)$, $\alpha(\alpha(Z)) = \alpha(Z)$, and every run that is feasible from a valuation $v' \in \alpha(Z)$ is simulated by a run from a valuation $v \in Z$. Such abstractions are called *extrapolations* in the literature [5]. An abstraction is finite when the set of abstracted zones $\{\alpha(Z) \mid Z \text{ is a zone}\}$ is finite. Given an abstraction α , the *abstracted zone graph* has initial node $(q, \alpha(\{\mathbf{0}\}))$ and transitions of the form $(q, Z) \Rightarrow_{\alpha}^t (q', \alpha(Z'))$ for each transition $(q, Z) \Rightarrow^t (q', Z')$. Let \Rightarrow_{α} denote the union of all \Rightarrow_{α}^t over $t \in T$. The abstracted zone graph is sound and complete: there is a run $(q_0, v_0) \xrightarrow{\delta_0, t_0} (q_1, v_1) \xrightarrow{\delta_1, t_1} \dots (q_n, v_n) (\xrightarrow{\delta_n, t_n} \dots)$ in A if and only if there is an infinite path $(q_0, Z_0) \Rightarrow_{\alpha}^{t_0} (q_1, Z_1) \Rightarrow_{\alpha}^{t_1} \dots (q_n, Z_n) (\Rightarrow_{\alpha}^{t_n} \dots)$ with $v_i \in Z_i$ for all $i \geq 0$. Hence, when α is a finite abstraction, the verification problems for a TA A can be algorithmically solved from its abstracted zone graph. The abstraction Extra_{LU}^+ [5] is implemented by state-of-the-art verification tools UPPAAL [24] and TChecker [19]. Our results hold for any finite, sound and complete abstraction. The abstracted zone graph is denoted \Rightarrow_{α} in the sequel.

2.3 Subsumption

Consider the TA in Fig. 1 and its zone graph in Fig. 2a. Observe that every run that is feasible from node $(q_1, x \geq 1)$ is also feasible from $(q_1, x \geq 0)$ since the zone $x \geq 1$ is included in the zone $x \geq 0$ (recall that zones are sets of clock valuations). We say that $(q_1, x \geq 1)$ is *subsumed* by the node $(q_1, x \geq 0)$. As a result, if an accepting node is (repeatedly) reachable from $(q_1, x \geq 1)$, then an accepting node is also (repeatedly) reachable from $(q_1, x \geq 0)$.

This leads to a crucial optimization for the verification of TA: reachability and liveness verification problems can be solved without exploring subsumed nodes. This optimization is called *inclusion abstraction* in [11]. Figure 2b shows the graph obtained when the exploration is stopped at node $(q_1, x \geq 1)$. All the runs that are feasible from $(q_1, x \geq 1)$ are still represented in this graph, as they can be obtained by first taking the subsumption edge from $(q_1, x \geq 1)$

to $(q_1, x \geq 0)$ (depicted as a blue squiggly arrow), and then any sequence of (actual or subsumption) edges from $(q_1, x \geq 0)$. Such graphs with both actual and subsumption edges are called *subsumption graphs* in the sequel.

It is tempting to use subsumption as much as possible, and only explore maximal nodes (w.r.t. zone inclusion). While this is correct for the verification of reachability properties, subsumption must be used with care for liveness verification. The bottom node $(q_1, x \geq 2)$ in Fig. 2b is also subsumed by the node $(q_1, x \geq 0)$. A subsumption edge can thus be added between these two nodes as depicted in Fig. 2c. However, due to this new subsumption edge, the graph has a Büchi accepting path (of actual and subsumption edges) that does not correspond to any run of the timed automaton in Fig. 1. Indeed, subsumption leads to an overapproximation of the runs of the automaton. While all the runs from node $(q_1, x \geq 2)$ are feasible from node $(q_1, x \geq 0)$, the converse is not true: the transition $q_1 \xrightarrow{x < 2, x := 0} q_2$ is not feasible from $(q_1, x \geq 2)$.

The subsumption graphs in Fig. 2b and 2c can be seen as certificates issued by verification algorithms. The graph in Fig. 2b is a valid certificate for liveness verification as 1) it contains no accepting paths, and 2) every run of the automaton is represented in the graph. In contrast, the graph in Fig. 2c is not a valid certificate for liveness verification as it has an accepting path that does not correspond to any run of the automaton. In the next sections, we introduce an algorithm to check the validity of certificates produced by liveness verification algorithms, as well as a proven implementation of the algorithm.

3 Certificates for Büchi Emptiness

In this section, we study certificates for Büchi emptiness in the setting of a slight variation of well-structured transition systems [14]. First, we present reachability invariants, which certify that every run in the original system can be simulated on the states given in the invariant. Next, we show that the absence of certain cycles in the invariant is sufficient to prove that the original transition system does not contain accepting runs. Then, we add a proof of absence of these cycles to the certificate. Finally, we instantiate this framework for the case of TA.

3.1 Self-simulating Transition Systems

A *transition system* (S, \rightarrow) consists of a set of states S and a transition relation $\rightarrow \subseteq S \times S$. If S is clear from the context, we simply write \rightarrow . We say that $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n$ is a path or that $s_1 \rightarrow s_2 \rightarrow \dots$ is an (infinite) run in \rightarrow if $s_i \rightarrow s_{i+1}$ for all i . Given an initial state s_0 and a predicate for accepting states ϕ , the path $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$ is accepting if $\phi(s_n)$. A run $s_0 \rightarrow s_1 \rightarrow \dots$ is an (accepting) Büchi run if $\phi(s_i)$ for infinitely many i .

A transition system \rightarrow is simulated by the transition system \rightarrow' if there exists a simulation relation \sqsubseteq such that:

$$\forall s, s', t. s \sqsubseteq s' \wedge s \rightarrow t \longrightarrow \exists t'. s' \rightarrow' t' \wedge t \sqsubseteq t'$$

This *simulation property* can be lifted to paths and runs:

Proposition 1. *If $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n (\rightarrow \dots)$ is a path (run) and $s_1 \sqsubseteq t_1$, then there is a path (run) $t_1 \rightarrow' t_2 \rightarrow' \dots \rightarrow' t_n (\rightarrow' \dots)$ with $s_i \sqsubseteq t_i$ for all i .*

Definition 1. A self-simulating transition system (SSTS) $(S, \rightarrow, \preceq)$ consists of a transition system (S, \rightarrow) and a quasi-order (a reflexive and transitive relation) $\preceq \subseteq S \times S$ on states such that \rightarrow is simulated by \rightarrow itself for \preceq .

In comparison to well-structured transition systems [14], our definition is slightly more relaxed, as we only demand that \preceq is a quasi order, not a well-quasi order. Intuitively, transitivity of \preceq is needed to allow for correct simulation by arbitrary “bigger” nodes. In TA, \preceq corresponds to subsumption \sqsubseteq , and \rightarrow corresponds to \Rightarrow .

3.2 Reachability Invariants on Abstract Transition Systems

In this section, we introduce the concept of *reachability invariants* for SSTS.

Definition 2. A set $I \subseteq S$ is a reachability invariant of an SSTS $(S, \rightarrow, \preceq)$ iff for all $s \in I$ and t with $s \rightarrow t$, there exists a $t' \in I$ such that $t \preceq t'$.

A useful invariant is also fulfilled by some initial state. Such states will show up in theorems below. In the remainder, unless noted otherwise, $(S, \rightarrow, \preceq)$ is an SSTS and I is a reachability invariant of it. Figures 2a to 2c all form a reachability invariant for the zone graph from Fig. 2a.

As was observed by Wimmer and von Mutius [37], reachability invariants can directly be applied as certificates for *unreachability*.

Definition 3. A predicate ϕ (for accepting states) is compatible with an SSTS $(S, \rightarrow, \preceq)$ iff for all $s, s' \in S$, if $\phi(s)$ and $s \preceq s'$, then also $\phi(s')$.

An invariant I can now certify that no accepting state s with $\phi(s)$ is reachable:

Theorem 1. *If $\forall s \in I. \neg\phi(s)$, for some compatible ϕ , $s_0 \in S$ and $s'_0 \in I$ with $s_0 \preceq s'_0$, then there is no accepting path $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$ with $\phi(s_n)$.*

Note that this approach to certifying unreachability is also complete: if no accepting state is reachable from s_0 in $(S, \rightarrow, \preceq)$, we can simply set $I := S$. However, this is not practical for infinite transition systems, of course. Thus we will revisit the question of completeness for TA below.

Finally, we observe that the invariant can be limited to a restriction of \preceq .

Definition 4. A pair (I, \trianglelefteq) of a set $I \subseteq S$ and a binary relation \trianglelefteq is a restricted reachability invariant of an SSTS $(S, \rightarrow, \preceq)$ iff:

1. For all $s \in I$ and t with $s \rightarrow t$, there exists a $t' \in I$ such that $t \trianglelefteq t'$.
2. For all s, t , if $s \trianglelefteq t$, then also $s \preceq t$.

In Figure 2, the \rightsquigarrow -arrows would play the role of \trianglelefteq . In Figure 2b, $(q_1, x \geq 0)$ is subsumed by both $(q_1, x \geq 1)$ and $(q_1, x \geq 2)$, but as we have seen in Figure 2c, it is crucial to disregard these subsumptions. Therefore we need to consider restricted reachability invariants.

For any restricted reachability invariant, we can define a simulating transition system $\rightarrow_{\trianglelefteq}$:

Definition 5. The transition system $(S, \rightarrow_{\trianglelefteq})$ is defined such that $s \rightarrow_{\trianglelefteq} t'$ iff there exists a t such that $s \rightarrow t$ and $t \trianglelefteq t'$.

This simulation theorem is the key property of restricted reachability invariants²:

Theorem 2. Given $s_1 \trianglelefteq t_1$ with $t_1 \in I$, if $s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n (\rightarrow \dots)$ is a path (run), then there is a path (run) $t_1 \rightarrow_{\trianglelefteq} t_2 \rightarrow_{\trianglelefteq} \dots \rightarrow_{\trianglelefteq} t_n (\rightarrow \dots)$ such that $s_i \trianglelefteq t_i$ and $t_i \in I$ for all i .

Analogously to $\rightarrow_{\trianglelefteq}$, the transition system \rightarrow_{\preceq} can be defined, and Theorem 2 can be proved for \rightarrow_{\preceq} . This is used for the proof of Theorem 1 (see [37]).

3.3 Büchi Emptiness on Abstract Transition Systems

In this section, we first give a general means of certifying that a transition system does not contain a cycle, and then combine the idea with reachability invariants to certify the absence of Büchi runs on SSTS.

Definition 6. Given a transition system \rightarrow and an accepting state predicate ϕ , a topological numbering of \rightarrow is a function f with an integer range such that:

1. For all s, t , if $s \rightarrow t$, then $f(s) \geq f(t)$.
2. For all s, t , if $s \rightarrow t$ and $\phi(s)$, then $f(s) > f(t)$.

Proposition 2. Let f be a topological numbering of \rightarrow and ϕ . If there exists a path of the form $s \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s$, then $\neg\phi(s)$.

These certificates are also complete:

Proposition 3. If there is no path $s \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s$ with $\phi(s)$ in \rightarrow , then the following are topological numberings for \rightarrow .

1. The number of accepting states that are reachable from a node: $f(s) := |\{x \mid s \rightarrow^* x \wedge \phi(x)\}|$ (assuming $\{x \mid s \rightarrow^* x \wedge \phi(x)\}$ is finite for any s).

² All proofs are omitted for brevity and can be found in the appendix of the online version of this paper on arXiv: <https://arxiv.org/abs/2007.04150>.

2. If h is a topological numbering (in the classical sense) of the strongly connected components (SCCs) of \rightarrow , then set $g(s) := h(C)$ if $s \in C$.

We now lift this idea to the case of (restricted) reachability invariants.

Definition 7. Given an SSTS $(S, \rightarrow, \preceq)$, an accepting state predicate ϕ , and a corresponding restricted reachability invariant (I, \trianglelefteq) , a restricted topological numbering of $(S, \rightarrow, \preceq)$ is a function f with an integer range such that:

1. For all $s, t' \in I$ and $t \in S$, if $s \rightarrow t$, and $t \trianglelefteq t'$, then $f(s) \geq f(t')$.
2. For all $s, t' \in I$ and $t \in S$, if $s \rightarrow t$, $t \trianglelefteq t'$, and $\phi(s)$, then $f(s) > f(t')$.

Moreover, let $\rightsquigarrow_{\trianglelefteq}$ be the restriction of $\rightarrow_{\trianglelefteq}$ to I , i.e. the transition system such that $s \rightsquigarrow_{\trianglelefteq} t'$ iff $s, t' \in I$ and there exists a t such that $s \rightarrow t$ and $t \trianglelefteq t'$.

Now, f is clearly a topological numbering for $\rightsquigarrow_{\trianglelefteq}$. Thus $\rightsquigarrow_{\trianglelefteq}$ is free of accepting cycles. Additionally, the transition system $\rightsquigarrow_{\trianglelefteq}$ trivially simulates $\rightarrow_{\trianglelefteq}$ with $s \sqsubseteq s'$ iff $s' = s$ and $s \in I$. Therefore, any accepting cycle $s \rightarrow_{\trianglelefteq}^+ s$ in $\rightarrow_{\trianglelefteq}$ with $s \in I$ and $\phi(s)$ yields an accepting cycle $s \rightsquigarrow_{\trianglelefteq}^+ s$. Hence $\rightarrow_{\trianglelefteq}$ is free of accepting cycles.

From this, we conclude our main theorem that allows one to certify absence of Büchi runs in a transition system \rightarrow .

Theorem 3. Let f be a restricted topological numbering of $(S, \rightarrow, \preceq)$ for a compatible predicate ϕ and a finite restricted reachability invariant (I, \trianglelefteq) . Then, for any initial state $s_0 \in S$ with $s_0 \trianglelefteq t_0$ for $t_0 \in I$, there is no Büchi run from s_0 .

In practice, a certificate can now be given as a finite restricted reachability invariant I as described above, and a corresponding restricted topological numbering f . Both properties can be checked locally for each individual state in I .

3.4 Instantiation for Timed Automata

We now want to instantiate this abstract certification framework for the concrete case of TBA. Our goal is to certify that the zone graph \Rightarrow does not contain any Büchi runs. As the zone graph is complete, this implies that the underlying TBA is empty. Thus we set $\rightarrow := \Rightarrow$. Subsumptions in the zone graph shall correspond to the self-simulation relation of the SSTS. Hence we define \preceq such that $(q, Z) \preceq (q', Z')$ iff $q' = q$ and $Z \subseteq Z'$.

To certify unreachability, it is sufficient to consider arbitrary subsumptions in the zone graph, i.e. $\trianglelefteq := \preceq$ [37]. In other words it is sufficient to check that the given certificate I is a reachability invariant for $(S, \rightarrow, \preceq)$. We have not yet given the set of states S . Abstractly, S is simply the set of non-empty states, i.e. $S := \{(q, Z) \mid Z \neq \emptyset\}$. If it was allowed to reach empty zones, then soundness of the zone graph would not be given. In practice, the certifier needs to be able to compute \Rightarrow effectively, typically using the DBM representation of zones. To this end one wants to add the assumption on states that all DBMs are in canonical form. One needs to ensure that states are split according to ϕ , i.e. $\forall (q, Z) \in S. Z \subseteq \Phi(q) \vee Z \cap \Phi(q) = \emptyset$ where $\Phi(q) = \{v \mid \phi(q, v)\}$. This is trivial for commonly used properties that concern only the finite state part.

Following these considerations, we propose the following certifier for the emptiness of TBA. A certificate C is a set of triplets (q, Z, i) where q is a discrete state, Z is a corresponding zone, and i is the topological number for (q, Z) . The certifier runs Algorithm 1 on this certificate. The algorithm extends the one by Wimmer and Mutius [37] with the topological numbers for liveness checking.

Theorem 4. *If $\text{BÜCHI-EMPTINESS}(\phi, C, q_0)$ accepts the certificate, then \Rightarrow_{DBM} has no Büchi run for ϕ . Consequently, the underlying TBA is empty.*

The proof constructs a suitable \preceq such that $(q, Z) \preceq (q, Z')$ if $Z \subseteq Z'$ and $(q, Z', k) \in C$, where k is selected to be minimal. Setting $I := \{(q, Z) \mid \exists i. (q, Z, i) \in C\}$ and $f(q, Z) := \min\{i \mid (q, Z, i) \in C\}$, Theorem 3 can be applied.

Algorithm 1. Certifier for the emptiness of TBA

```

1: procedure BÜCHI-EMPTINESS( $\phi, C, q_0$ )
2:   for all  $(q, Z, i) \in C$  do                                     ▷ All DBMs are well-formed
3:     if  $Z = \emptyset \vee Z$  is not canonical
4:       then reject certificate
5:   if  $\nexists (q_0, Z_0, i) \in C. \{\mathbf{0}\} \subseteq Z_0$                        ▷ The initial state is covered
6:     then reject certificate
7:   for all  $(q, Z, i) \in C$  do                                       ▷ The certificate is:
8:     for all  $(q_1, Z_1)$  s.t.  $(q, Z) \Rightarrow (q_1, Z_1)$  do
9:       if  $(\nexists (q_1, Z_1, j) \in C. Z_1 \subseteq Z_1'$ 
           $\wedge (\phi(q) \longrightarrow i > j) \wedge i \geq j)$ 
           $\wedge (\phi(q) \longrightarrow i > j) \wedge i \geq j)$ 
          do
10:        then reject certificate
11:   accept certificate
    
```

The algorithm inherits several beneficial properties from [37]. First, it can easily be parallelized. Most importantly however, the certifier does not need to compute an abstraction operation α . Suppose the model checker starts with a state $(q_0, \{\mathbf{0}\})$ and explores the transition $(q_0, \{\mathbf{0}\}) \Rightarrow (q_1, Z_1)$. The model checker could then abstract zone Z_1 to $\alpha(Z_1)$, and explore more edges from $(q_1, \alpha(Z_1))$, e.g. $(q_1, \alpha(Z_1)) \Rightarrow (q_2, Z_2)$. The certificate just needs to include $(q_0, \{\mathbf{0}\})$, $(q_1, \alpha(Z_1))$, and $(q_2, \alpha(Z_2))$, and the certificate checker just needs to check the following inclusions: $\{\mathbf{0}\} \subseteq \{\mathbf{0}\}$, $Z_1 \subseteq \alpha(Z_1)$, and $Z_2 \subseteq \alpha(Z_2)$. The checker does not need to compute α as $\alpha(Z_1)$ and $\alpha(Z_2)$ are part of the certificate.

It is rather easy to see that these certificates are also complete for timed automata. For any finite abstraction α , the abstracted zone graph \Rightarrow_α is finite and complete. Thus, for a starting state $(q_0, \{\mathbf{0}\})$ the set

$$I := \{(q, Z) \mid (q_0, \{\mathbf{0}\}) \Rightarrow_\alpha^* (q, Z)\}$$

is a trivial finite reachability invariant that can be computed effectively for common abstractions α . Moreover, if the underlying TBA is empty, then \Rightarrow_α cannot contain a Büchi run either, since the abstract zone graph is complete. Because \Rightarrow_α is finite, this means it cannot contain a cycle through ϕ . Hence a forward

numbering of I can be given by computing the strongly connected components of I . However, this type of certificate is not of practical interest as subsumptions are not considered. How certificates can be obtained for model checking algorithms that make use of subsumption is the topic of Sect. 5.1.

4 Incorporating Advanced Abstraction Techniques

We have already discussed that the techniques that were presented above are in principle agnostic to the concrete abstraction α used. This, however, is only true for standard verification algorithms for T(B)A that use zone inclusion $Z \subseteq Z'$ as a simulation relation on the abstract zone graph. There is also the noteworthy abstraction $\alpha_{\preceq LU}$ [5], which is the coarsest zone abstraction that can be defined from clock bounds L, U [18]. Herbretau et al. have shown that even though $\alpha_{\preceq LU}(Z)$ is usually not a zone, it can be checked whether $Z \subseteq \alpha_{\preceq LU}(Z')$ directly from the DBM representation of Z and Z' , without computing $\alpha_{\preceq LU}(Z')$ [18]. Hence, one can use $\alpha_{\preceq LU}$ -subsumption over zones, $Z \subseteq \alpha_{\preceq LU}(Z')$, instead of standard inclusion $Z \subseteq Z'$ to explore fewer symbolic states. This technique can also be integrated with our certification approach. This time, we will need more knowledge about the concrete abstraction α , however.

We first describe the concept of time-abstract simulations, on which the definition of $\alpha_{\preceq LU}$ is based.

Definition 8. *A time-abstract simulation between clock valuations is a quasi-order \preceq such that if $v \preceq v'$ and $(q, v) \rightarrow_{\delta, t} (q_1, v_1)$ then there exist δ' and v'_1 such that $(q, v') \rightarrow_{\delta', t} (q_1, v'_1) \wedge v_1 \preceq v'_1$.*

Behrmann et al. defined the simulation \preceq_{LU} based on the clock bounds L and U , and showed that it is a time-abstract simulation [5] (in fact one can show that \preceq_{LU} is even a simulation, i.e. $\delta' = \delta$). For any \preceq , one can define the corresponding abstraction $\alpha_{\preceq}(Z) = \{v \mid \exists v' \in Z. v \preceq v'\}$. This yields a sound and complete abstraction for any time-abstract simulation \preceq [5]. Observe that $\alpha_{\preceq}(Z)$ is the set of all valuations that are simulated by a valuation in Z w.r.t. \preceq . As a result, every sequence of transitions feasible from $\alpha_{\preceq}(Z)$ is also feasible from Z (although with different delays).

The implicit abstraction technique based on the subsumption check $Z \subseteq \alpha_{\preceq}(Z')$ is compatible with our certification approach for any α_{\preceq} for which \preceq is a time-abstract simulation, and in particular $\alpha_{\preceq LU}$. Actually, we are still able to use algorithm BÜCHI-EMPTYNESS with the only modification that the condition $Z_1 \subseteq Z'_1$ is replaced with $Z_1 \subseteq \alpha_{\preceq}(Z'_1)$. We will justify this by showing that if the algorithm accepts the certificate, then it represents a restricted reachability invariant with a suitable topological numbering for $\Rightarrow_{\alpha_{\preceq}}$. This means that $\Rightarrow_{\alpha_{\preceq}}$ does not have a Büchi run (Theorem 3), which, as α_{\preceq} is a complete abstraction, implies that the underlying TBA does not have a Büchi run either.

We first prove the following monotonicity property (which can be seen as a generalization of Lemma 4 in the work of Herbretau et al. [18]).

Proposition 4. *Let \preceq be a time-abstract simulation. If $\alpha_{\preceq}(W) \subseteq \alpha_{\preceq}(W')$, $(q, W) \Rightarrow^t (q_1, W_1)$, and $(q, W') \Rightarrow^t (q_1, W'_1)$, then $\alpha_{\preceq}(W_1) \subseteq \alpha_{\preceq}(W'_1)$.*

Reminding ourselves that α_{\preceq} is idempotent, it follows that if $(q, W) \Rightarrow^t (q_1, W_1)$ and $(q, \alpha_{\preceq}(W)) \Rightarrow^t (q_1, W'_1)$ for some states q, q_1 , and sets of valuations W, W_1 , and W'_1 , then $\alpha_{\preceq}(W_1) = \alpha_{\preceq}(W'_1)$. In other words, \Rightarrow simulates $\Rightarrow_{\alpha_{\preceq}}$ for \sqsupseteq defined as $(q, W) \sqsupseteq (q, Z) \iff W = \alpha_{\preceq}(Z)$.

Now, we show that the conditions of Definitions 4 and 7 can be transferred along this simulation.

Theorem 5. *Assume that the following conditions hold:*

1. *For all states q , and zones Z, Z', Z'' , if $(q, Z) \trianglelefteq (q, Z')$, then $Z \subseteq \alpha_{\preceq}(Z')$.
Moreover, if $\alpha_{\preceq}(Z) = \alpha_{\preceq}(Z')$ and $(q, Z) \trianglelefteq (q, Z'')$, then $(q, Z') \trianglelefteq (q, Z'')$.*
2. *For all q, Z , if $\phi((q, \alpha_{\preceq}(Z)))$, then $\phi((q, Z))$.*
3. *(I, \trianglelefteq) satisfies condition (1) of Definition 4 for \Rightarrow .*
4. *f is a restricted topological numbering for $\Rightarrow, (I, \trianglelefteq)$, and ϕ .*

Let $(q, W) \trianglelefteq' (q, W') \iff \exists Z, Z'. W = \alpha_{\preceq}(Z) \wedge W' = \alpha_{\preceq}(Z') \wedge (q, Z) \trianglelefteq (q, Z')$, $I' := \{s' \mid \exists s \in I. s \sqsubseteq s'\}$ and $f'(s') := \text{Min} \{f(s) \mid s \in I \wedge s \sqsubseteq s'\}$. Then

1. *(I', \trianglelefteq') is a restricted reachability invariant for $(\Rightarrow_{\alpha_{\preceq}}, \subseteq)$.*
2. *f' is a restricted topological numbering for $\Rightarrow_{\alpha_{\preceq}}, (I', \trianglelefteq')$, and ϕ .*

Algorithm BÜCHI-EMPTINESS ensures that there exist an invariant (I, \trianglelefteq) and a numbering f that fulfill the conditions of Theorem 5 for \Rightarrow (as indicated after Theorem 4). Thus, if the algorithm accepts the certificate, there is a restricted reachability invariant (I', \trianglelefteq') with a corresponding topological numbering f' for $\Rightarrow_{\alpha_{\preceq}}$. Hence $\Rightarrow_{\alpha_{\preceq}}$ does not have a Büchi run.

5 Evaluation

In this section, we first give a brief description of the model checking algorithms we consider and describe how certificates can be extracted from them. Then, we outline the general architecture of our certification tool chain, and finally we present some experiments on standard TA models.

5.1 Extracting Certificates from Model Checkers

We consider the two state-of-the-art algorithms for checking Büchi emptiness for TA: the NDFS-based algorithm by Laarman et al. [22] and the iterative SCC-based algorithm by Herbreteau et al. [16]. Both algorithms can be applied to any abstracted zone graph \Rightarrow_{α} for a finite, sound and complete abstraction α . As was noted by Herbreteau et al. [16], they also have in common that their correctness can be justified on the basis that they both compute subsumption graphs that are *liveness compatible*, in the sense that they do not contain any cycle with an accepting node and a subsumption edge.

Considering NDFS for TA from [22] more closely, it prunes the search space by using subsumption in certain safe places. In particular, the outer (blue) search is pruned when it reaches a state s that is subsumed by a state on which the inner (red) search has been called, i.e. $s \sqsubseteq t$ and t is red. In order to generate a liveness-compatible subsumption graph, the blue search exports all the states which are not subsumed along with their \rightarrow -successors. Moreover, the algorithm exports \rightsquigarrow -edges as soon as the pruning by subsumption is applied.

The iterative algorithm from [16] interleaves reachability analysis and SCC decompositions. The reachability analysis computes a subsumption graph with maximal subsumption: a subsumption edge $s \rightsquigarrow t'$ is added whenever a new state t is visited from s , and t is subsumed by some visited state t' . The resulting graph $\rightarrow \cup \rightsquigarrow$ is a subsumption graph that preserves state reachability, but that may not be liveness compatible. Therefore, an SCC decomposition is run, and all subsumption edges from SCCs that contain both an accepting node and a subsumption edge are removed. States which are not subsumed anymore are re-explored in the next iteration of the main loop. Upon termination, the subsumption graph $\rightarrow \cup \rightsquigarrow$ is liveness compatible.

Both algorithms compute liveness compatible subsumption graphs. In order to obtain a certificate we run one extra SCC decomposition of the graph with $\rightarrow \cup \rightsquigarrow$ -edges from which we compute a topological ordering.

5.2 General Architecture

Our certifier is implemented as an extension of the tool Munta [34], which has been fully verified in Isabelle/HOL [36,37]. Figure 3 depicts the architecture of our tool chain to certify the emptiness of a given TBA. The model (a TBA) and the acceptance property are given in the input format of Munta. For the model checker in the middle, we used Imitator and TChecker. In a first step, the Munta model is translated to an input model for the model checker. The model checker decides whether the given TBA is empty. If not, then either the model checker's answer is correct or it has found a spurious counterexample; in both cases no certificate can be extracted. Otherwise, the model checker emits a certificate consisting of a number of symbolic states and the set of edges in the subsumption graph. The latter can either include proper transitions (\rightarrow) and subsumptions (\rightsquigarrow) (this is done for Imitator with NDFS and subsumption), or the edges that merge these two types (\rightsquigarrow') (which is done for TChecker and for Imitator with state merging enabled, see Sect. 5.3). In either case, in the next step where the certificate is translated to Munta's binary input format for certificates, the SCC numbers (c.f. Proposition 3) are re-computed blindly from these edges. This step additionally makes use of a renaming dictionary to map from human readable labels for states, actions, etc., to natural numbers.

Finally, the TBA model, the translated certificate, and the renaming are given to Munta. If it accepts the certificate, then there is an Isabelle/HOL theorem that guarantees that the given TBA is indeed empty. If the certificate is rejected, any of the steps in the tool chain could have failed. Note that the basis of trust is minimal. One just needs to ensure that the model represents what one has

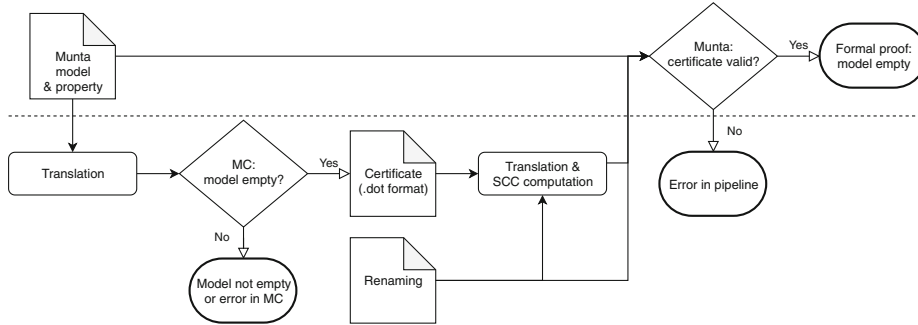


Fig. 3. Workflow of the certifier pipeline. The dashed line is the trust boundary. If the correct model is given, then the answer on the right can be trusted.

in mind, and to trust the correctness of Munta. To trust Munta, one essentially needs to trust its TBA semantics, which is less than 200 lines long, some core parts of Isabelle/HOL, and an SML compiler (MLton in our case). For details, we refer the interested reader to previous publications on Munta [34, 37].

5.3 Experiments

We have evaluated our approach on the TBA models that were also used by Herbreteau et al. [16]. These are inspired by standard TA benchmarks, and all consist of the product of a TA model and an additional Büchi automaton that encodes the complement of the language of a given LTL formula that one wants to check. Details are given by Herbreteau et al. [16].

For Imitator we tried two methods: NDFS with subsumption and reachability analysis with merging [3]. Imitator does not apply abstractions (since it was designed for parametric TA), so the full zone graph is often infinite and most NDFS runs fail. The one that succeeds generates a valid certificate. *Merging* tries to reduce the number of zones, by computing the exact convex hull of zones. This creates new zones that could subsume several existing ones, and often yields a finite zone graph. The certificate produced by merging is always a reachability invariant but not necessarily a subsumption graph. Merging may introduce spurious cycles, in which case the certificate is not liveness compatible; these cases are caught by the Munta certifier. If there are no (spurious) accepting cycles, we obtain a valid and quite small certificate. Note that the generalization from subsumption graphs to our certificates is crucial to allow for merging.

Table 1 summarizes our experimental results. TChecker was run with the algorithm from [16] and [22] and Imitator with the algorithm from [22], and with a reachability procedure with full merging. The *** entries indicate cases where Imitator did not terminate within 30 s. The results show that the certifier accepts those certificates that we expect it to accept, but also rejects those that stem from subsumption graphs that are not liveness compatible. Moreover, the certifier was fully verified in Isabelle/HOL and still yields reasonable performance, certifying models with more than a 100 k symbolic states in under 230 s.

Table 1. Benchmark results on a 2017 MacBook Pro with 16 GB RAM and a Quad-Core Intel Core i7 CPU at 3.1 GHz. For each algorithm, we show whether the certificate was accepted, the number of DBMs in the certificate, and the time for certificate checking on a single core in seconds.

Model	TChecker						Imitator					
	Iterative SCC			NDFS			Merge			NDFS		
CC1	✓	57	0.01	✓	3281	0.06	✓	58	0.01		***	
CC4	✓	195858	221.56	✓	32575	7.75		***			***	
CC5	✓	65639	30.63	✓	143057	218.98		***			***	
FD1	✓	214	0.02	✓	677	0.03	✗	294	0.02	✓	1518 0.11	
FI1	✓	65	0.01	✓	71	0.00	✓	136	0.00		***	
FI2	✓	314	0.01	✓	344	0.01	✓	589	0.01		***	
FI4	✓	204	0.00	✓	224	0.01	✓	793	0.01		***	
FI5	✓	3091	0.13	✓	2392	0.09	✗	863	0.03		***	

6 Conclusion

Starting from an abstract theory on self-simulating transition systems, we have presented an approach to extract certificates from state-of-the-art model checking algorithms (including state-of-the-art abstraction techniques) that decide emptiness of timed Büchi automata. The certificates prove that a given model is indeed Büchi empty. We have verified the theory and a checker for these certificates in Isabelle/HOL, using the tool Munta as a basis. We demonstrated that our approach is feasible by extracting certificates for some standard benchmark models from the tools TChecker and Imitator. We hope that our work can help to increase confidence in safety-critical systems that have been verified with timed automata model checkers. Furthermore, we envision that our tool could help in the organization of future competitions for such model checkers.

To close, we want to illuminate some potential future directions of research. First, one is usually not only interested in the emptiness of TBA per se, but more generally in the question if a TA model satisfies some LTL requirements. Thus, our tool would ideally be combined with a verified translation from LTL formulas to Büchi automata or with a certifier for such a construction. The former has been realized by the CAVA project [13], while an avenue towards the latter is opened by the recent work of Seidl et al. [10].

Second, Herbreteau et al. have developed a technique of computing abstractions for TA on the fly, starting from very coarse abstractions and refining them as needed [17]. It seems that our approach is in principle compatible with this technique when augmenting certificates with additional information on the computed abstractions, whose validity would have to be checked by the certifier.

Third, one could attempt to reduce the size of the certificates. In one approach, reachability certificates have been compressed after model checking (c.f. [37]). On the other hand, model checking algorithms could speculate that the given TBA is empty, and use this fact to use additional subsumptions to reduce the search space, while risking to miss accepting runs. However, given the certification step afterwards, this is of no concern. For instance, one could remove the red search from the NDFS algorithm, and use subsumption on blue nodes instead of red nodes, as a quick pre-check. If the result passes the certifier, we are done.

Finally, as our theory is not specific to timed automata per se, it could be interesting to find other application domains for this approach to certification. In light of the large body of existing work on well-structured transition systems, this looks particularly promising as any such system is also self-simulating.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: a tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) *FM 2012*. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_6
3. André, É., Soulat, R.: Synthesis of timing parameters satisfying safety properties. In: Delzanno, G., Potapov, I. (eds.) *RP 2011*. LNCS, vol. 6945, pp. 31–44. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24288-5_5
4. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Gavel, H., Hatcliff, J. (eds.) *TACAS 2003*. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_18
5. Behrmann, G., Bouyer, P., Larsen, K.G., Pelanek, R.: Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transfer (STTT)* **8**(3), 204–215 (2006)
6. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*. LNCS, vol. 3908, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
7. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reasoning* **61**(1-4), 333–365 (2018). <https://doi.org/10.1007/s10817-018-9455-7>
8. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) *IFM 2017*. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
9. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reason.* **60**(1), 3–21 (2018)
10. Brunner, J., Seidl, B., Sickert, S.: A verified and compositional translation of LTL to deterministic Rabin automata. In: Harrison, J., O’Leary, J., Tolmach, A. (eds.) *ITP 2019*, September 9–12, 2019, Portland, OR, USA. *LIPICs*, vol. 141, pp. 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPICs.ITP.2019.11>

11. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054180>
12. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17
13. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
14. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoret. Comput. Sci. **256**(1), 63 – 92 (2001). [https://doi.org/10.1016/S0304-3975\(00\)00102-X](https://doi.org/10.1016/S0304-3975(00)00102-X), iSS
15. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: 2018 Formal Methods in Computer Aided Design (FMCAD) pp. 1–9 (2018)
16. Herbreteau, F., Srivathsan, B., Tran, T.T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) FSTTCS. LIPIcs, vol. 65, pp. 48:1–48:14. Schloss Dagstuhl (2016)
17. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, pp. 990–1005. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_71
18. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Inf. Comput. **251**, 67–90 (2016)
19. Herbreteau, F., Point, G.: TCHecker (2019). <https://github.com/fredher/tchecker>
20. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
21. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
22. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV, pp. 968–983. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
23. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) Automated Deduction - CADE 26, pp. 237–254. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_15
24. Larsen, G.K., Pettersson, P., Yi, W.: Uppaal in a nutshell. Software Tools for Technology Transfer **1**(1), 134–152 (1997)
25. Leino, K.R.M.: Developing verified programs with Dafny. In: ICSE, pp. 1488–1490. IEEE Computer Society (2013)
26. Li, G.: Checking timed büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_18
27. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2

28. Nguyen, H.G., Petrucci, L., van de Pol, J.: Layered and collecting NDFS with subsumption for parametric timed automata. In: ICECCS, pp. 1–9. IEEE Computer Society (2018)
29. Nipkow, T., Lawrence C. Paulson, Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer, Cham (2002). <https://doi.org/10.1007/3-540-45949-9>
30. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: Biere, A., Parker, D. (eds.) TACAS 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 247–265. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-45190-5_14
31. Pol, J.C.: Automated verification of nested DFS. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 181–197. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_12
32. Tripakis, S.: Checking timed Büchi emptiness on simulation graphs. *ACM Trans. Comput. Logic* 10(3) (2009)
33. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. *Formal Methods Syst. Des.* **26**(3), 267–292 (2005)
34. Wimmer, S.: Munta: a verified model checker for timed automata. In: André, É., Stoelinga, M. (eds.) FORMATS 2019, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 236–243. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29662-9_14
35. Wimmer, S., Herbreteau, F., van de Pol, J.: Certifying emptiness of timed büchi automata: Artifact (2020). <https://doi.org/10.6084/m9.figshare.12620582.v1>
36. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4
37. Wimmer, S., von Mutius, J.: Verified certification of reachability checking for timed automata. In: Biere, A., Parker, D. (eds.) TACAS 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 425–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_24