Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Abstraction Refinement-Based Verification of Timed Automata

Ph.D. Dissertation

Tamás Tóth

Thesis supervisor:
István Majzik, Ph.D.

Budapest
2021

Tamás Tóth
http://www.mit.bme.hu/general/staff/totht

May 2021

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

doi: 10.5281/zenodo.4759047

Declaration of own work and references

I, Tamás Tóth, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Tóth Tamás kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2021. 05. 18.

Tóth Tamás

Acknowledgements

Throughout the time this dissertation was in the making, and the years of research preceding it, I had the good fortune to receive support from a great number of colleagues and friends.

First and foremost, I'd like to express my gratitude towards my advisor István Majzik for guiding my work with his wealth of knowledge and attention to detail.

I would also like to thank my colleagues at BME's Fault Tolerant Systems Research Group for their continuous help and support, and for all the stimulating discussions. I would like to acknowledge my coauthors who I was lucky enough to work with on the papers related to this dissertation, in chronological order, András Vörös, Ákos Hadju, Zoltán Micskei, and Rebeka Farkas.

I would like to thank Richter Gedeon's Talentum Foundation for generously providing a scholarship throughout my years as a PhD student.

Last but not least, I would like to say heartfelt thanks to my family for the long years of continuous support.

Summary

Formal methods are mathematical techniques that enable the rigorous specification and verification of hardware and software systems, typically in design time. *Formal verification* techniques are formal methods for reasoning about the correctness of systems with respect to a formal specification or property. *Model checking* is an automatic formal verification technique that is based on exhaustive traversal of the design model's state space. Its main advantage to more conventional verification methods (e.g. testing) is that it is not only able to detect faults in faulty systems, but can also show that a correct system is fault-free. However, a major difficulty in the successful application of model checking to verification of practical systems is its high computational cost: the cardinality of a system's state space is typically exponential in the size of the input specification describing the system's behavior, a phenomenon commonly known as *state space explosion*. In addition, the state space is not necessarily finite, in particular for real-time systems, where continuous variables with time dimension are part of the specification.

Therefore, to make the problem more tractable, advanced model checkers rely on *symbolic techniques*, where, instead of individual states, sets of states are considered during state space traversal; and *abstraction*, where only parts of the system that are relevant for the requirement are considered. As a result, the abstracted system is a simpler system whose behavior overapproximates that of the original system, therefore, if the abstract system is correct, so is the original one. However, as the abstracted system might admit false negatives, that is, spurious faulty behavior that is not present in the original system, the key challenge is finding the right abstraction granularity. This process can be automated using *abstraction refinement* techniques: in case of a false negative, the abstraction is refined in a way that excludes the discovered faulty behavior.

Our goal is to provide a generic, modular and configurable *model checking framework* that supports the development and evaluation of *abstraction refinement-based algorithms* for checking *safety properties* over different formalisms. In particular, by specific instantiations of our framework, we aim to provide efficient algorithms for the model checking of *real-time systems*. We focus primarily on classical *timed automata* with continuous *clock variables*, a formalism prominently used in the area of model checking real-time systems, and its extension with *discrete variables*. Moreover, we investigate methods for proving *liveness properties* of industrial real-time systems with asynchronous message passing. We propose several contributions towards these goals.

First, we introduce Theta, a generic, modular and configurable *model checking framework* for abstraction refinement-based reachability analysis of different formalisms. For the specific case of *timed automata with discrete variables*, we present a specialization of our framework that enables the combination of various abstraction and refinement strategies for the location reachability problem.

Second, we propose an abstraction technique for timed automata based on *interpolation for zones*. We propose two refinement strategies, both a combination of forward search, backward search and interpolation. We show that our method is competitive in performance with the state of the art.

Third, we propose an abstraction technique for timed automata with discrete variables, where refinement is based on controlling the visibility of discrete variables using *interpolation for valuations*. We demonstrate that our method, combined with methods for the abstraction of clock variables, can achieve a significant reduction in the size of the state space.

Fourth, we investigate methods for liveness checking of industrial real-time protocols with asynchronous message passing. We propose the *calendar system* formalism, and suggest a k-induction based approach for checking liveness properties of such models. For systems with a hierarchical structure in functionality, we propose a *decomposition method* that can be used to split the original liveness checking problem into more tractable ones.

Összefoglaló

A *formális módszerek* olyan matematikai módszerek, melyek hardver-szoftver rendszerek precíz specifikációját és tipikusan tervezési idejű verifikációját célozzák. A *formális verifikációs* technikák olyan formális módszerek, melyek lehetővé teszik rendszerek egy adott specifikáció vagy tulajdonság szerint értelmezett helyességéről való érvelést. A *modellellenőrzés* a tervezési modell állapotterének kimerítő bejárásán alapuló automatikus formális verifikációs technika, melynek a hagyományos ellenőrzési módszerekkel (például a teszteléssel) szemben előnye, hogy nemcsak hibás rendszerek hibáit képes detektálni, hanem hibamentes rendszerek helyességét is képes igazolni. A modellellenőrzés a gyakorlatban előforduló rendszerek ellenőrzésére történő alkalmazásának nehézsége ugyanakkor a módszer magas számítási költsége: egy rendszer állapotterének számossága tipikusan a rendszer viselkedését leíró bemeneti specifikáció méretében exponenciális – ez közismert nevén az *állapottérrobbanás* problémája. Ráadásul az állapottér nem is feltétlenül véges, például valósidejű rendszerek esetében, ahol az idő dimenziójú folytonos változók a specifikáció részét képezik.

Ezért a probléma kezelhetőbbé tételének érdekében a fejlett modellellenőrző eszközök gyakran alkalmaznak *szimbolikus technikákat*, ahol az egyes állapotok helyett állapotok halmazai képezik az állapottérbejárás alapját; valamint *absztrakciót*, ahol az ellenőrzés a rendszer a vizsgált tulajdonság szempontjából releváns részleteire fókuszál. Az absztrakció eredménye egy egyszerűbb, az eredeti rendszer viselkedését felülbecslő rendszer, így ha az absztrakt rendszer helyes, akkor az eredeti is az. Ugyanakkor, mivel az absztrakt rendszer hamis ellenpéldákat produkálhat – azaz az eredeti rendszerben nem megfigyelhető hibás viselkedéseket – a megfelelő absztrakciós granularitás megtalálása kulcsfontosságú. Ezt a folyamatot automatizálja az *absztrakciófinomítás* módszere: hamis ellenpélda esetén az absztrakció oly módon kerül hangolásra, mely kizárja a felfedezett hamis viselkedést.

Célunk egy generikus, moduláris és konfigurálható *modellellenőrző keretrendszer* biztosítása, mely támogatja különböző formalizmusok *biztonságossági tulajdonságokat* vizsgáló *absztrakciófinomításalapú algoritmusainak* fejlesztését és kiértékelését. Célunk továbbá valósidejű rendszerek modellellenőrzésére hatékony algoritmusokat adni e keretrendszer konkrét megpéldányosításai által. Főként a *valósidejű rendszerek* modellellenőrzésére elterjedten alkalmazott, folytonos *óraváltozókkal* rendelkező klasszikus *időzített automatákra*, valamint ezek *diszkrét változókkal* kiegészített változatára összpontosítunk. Ezen felül aszinkron üzeneteket küldő ipari valósidejű protokollok *előségi tulajdonságainak* bizonyítására adunk módszereket. E célok mentén számos kontribúciót fogalmazunk meg.

Egyrészt bemutatjuk a Theta eszközt, mely egy különböző formalizmusok elérhetőségi tulajdonságainak absztrakciófinomítás-alapú ellenőrzésére szolgáló generikus, moduláris és konfigurálható *modellellenőrző keretrendszer*. A *diszkrét változókkal rendelkező időzített automaták* ellenőrzésére a keretrendszer egy olyan specializációját javasoljuk, mely támogatja a helyelérhetőségi probléma megoldására szolgáló különböző absztrakciós és finomítási stratégiák kombinálását.

Másrészt *zónák feletti interpoláción* alapuló absztrakciós módszert javaslunk időzített automaták ellenőrzésére. Két finomítási stratégiát adunk, melyek az előre- és hátrafelé keresés, valamint az interpoláció ötvözetei. Megmutatjuk, hogy módszerünk teljesítményben a legkorszerűbbekkel versenyez.

Harmadrészt olyan, *változóértékelések feletti interpoláción* alapuló absztrakciós módszert javaslunk diszkrét változókkal rendelkező időzített automaták ellenőrzésére, ahol a finomítás a változók láthatóságának szabályozásán alapul. Demonstráljuk, hogy módszerünk az óraváltozók feletti absztrakciós módszerekkel kombinálva jelentős csökkenést tud produkálni az állapottér méretében.

Negyedrészt módszereket javaslunk aszinkron üzeneteket küldő ipari valósidejű protokollok élőségi tulajdonságainak ellenőrzésére. Bemutatjuk a *naptárrendszer* formalizmust, és k-indukció alapú módszert adunk ilyen modellek élőségi vizsgálatára. Funkcionalitásukban hierarchikus rendszerekre *dekompozíciós módszert* adunk, mellyel az eredeti élőségi probléma kezelhetőbb részekre bontható.

Contents

Chapter 1

Introduction

The prevalence of *embedded systems* in everyday use is ever increasing. This also includes their application in *safety critical systems*, e.g. in the automotive, railway or avionic domain. Often, safety critical systems are also *real-time systems* with time-dependent behavior and requirements. The correctness of such systems is crucial, as a system level failure might lead to disastrous consequences, such as environmental harm, loss of valuable equipment, or even human injury. Therefore, in order to reduce the probability and seriousness of faults, these systems are specified in detail, and conformance to the specification is thoroughly verified.

Formal methods are mathematical techniques that enable the rigorous specification and verification of hardware and software systems, typically in design time. *Formal verification* techniques are formal methods for reasoning about the correctness of systems with respect to a formal specification or property. *Model checking* [EC82; QS82] is an automatic formal verification technique that is based on exhaustive traversal of the design model's state space. Its main advantage to more conventional verification methods (e.g. testing) is that it is not only able to detect faults in faulty systems, but can also show that a correct system is fault-free. However, a major difficulty in the successful application of model checking to verification of practical systems is its high computational cost: the cardinality of a system's state space is typically exponential in the size of the input specification describing the system's behavior, a phenomenon commonly known as *state space explosion*. In addition, the state space is not necessarily finite, in particular for real-time systems, where continuous variables with time dimension are part of the specification.

Therefore, to make the problem more tractable, advanced model checkers rely on *symbolic techniques* [Bur+92], where, instead of individual states, sets of states are considered during state space traversal; and *abstraction* [CGL94], where only parts of the system that are relevant for the requirement are considered. As a result, the abstracted system is a simpler system whose behavior overapproximates that of the original system, therefore, if the abstract system is correct, so is the original one. However, as the abstracted system might admit false negatives, that is, spurious faulty behavior that is not present in the original system, the key challenge is finding the right abstraction granularity. This process can be automated using *abstraction refinement* [Cla+00] techniques: in case of a false negative, the abstraction is refined in a way that excludes the discovered faulty behavior.

1.1 Goals

In order for model checking to be applicable for the verification of a given system, one has to model the examined aspects of the system's behavior in a suitable modeling formalism beforehand. Most model checking algorithms solve a particular verification task for a given formalism. However, as new designs to verify emerge, more generic tools are also needed since the appropriate formalism and algorithm may vary based on the characteristics of the task itself, and might not be known initially. Our goal is to provide a generic, modular and configurable *model checking framework* that supports the development and evaluation of *abstraction refinement-based algorithms* for checking *safety properties* over different formalisms. In particular, by specific instantiations of our framework, we aim to provide efficient algorithms for the model checking of *real-time systems*. We focus primarily on classical *timed automata* with continuous *clock variables*, a formalism prominently used in the area of model checking real-time systems, and its extension with *discrete variables*. Moreover, we investigate methods for proving *liveness properties* of industrial real-time systems with asynchronous message passing.

1.2 Summary of Challenges

In this dissertation, we aim to address the following challenges.

> **Challenge 1.** *Configurable abstraction refinement-based model checking.* Most tools focus on a specific algorithm and formalism to solve a particular verification task. Is it possible to provide a generic, modular and configurable model checking framework that supports the development, evaluation and application of abstraction refinement-based algorithms for the reachability analysis of models in different formalisms?
>
> **Challenge 2.** *Abstraction refinement for timed automata.* Abstraction refinement has been successfully used in model checking, and in particular for model checking software. Is it possible to provide abstraction refinement algorithms that are efficient in the domain of real-time systems?
>
> **Challenge 3.** *Model checking timed automata with discrete variables.* For practical real-time systems, design models typically contain discrete data variables with nontrivial data flow besides real-valued clock variables. Is it possible to provide methods for alleviating state space explosion in such models?
>
> **Challenge 4.** *Liveness checking for industrial real-time systems.* Requirements for industrial real-time systems are often formalized in terms of liveness properties. Is it possible to provide methods for liveness checking of such systems, while still supporting the various semantic features that are present in such models?

1.3 Structure of the Dissertation

The broad topic of this dissertation is thus model checking, in particular model checking real-time systems. In Chapter 2, we briefly summarize the theoretical background of model checking relevant to our work, and define the notations used throughout the dissertation. The remaining, core part of the dissertation can be conceptually divided into two parts, each focusing on a different aspect of the model checking flow.

In the first, more theoretical part, comprised of Chapter 3, Chapter 4, Chapter 5, and Chapter 6, we treat the model checker as a white box, and work on the internals of several model checking algorithms. These chapters are related to each other, and exposition follows a top-down approach. In Chapter 3, we introduce a formalism-agnostic model checking framework for abstraction refinement based model checking of reachability properties. In Chapter 4, we develop a specialization of this framework by fixing the formalism to timed automata with discrete variables, and the property to location reachability. In Chapter 5 and Chapter 6, our goal is to build efficient methods in this algorithmic framework.

In the second, more practice-oriented part, constituted by Chapter 7 and Chapter 8, we treat the model checker as a black box, and develop methods around it that enable its successful use for verifying practical systems. In both chapters, exposition is based upon the verification of a respective industrial case study, and we start from a high level specification that we formalize in a suitable modeling formalism. We extend our investigations to liveness properties, as these are often required for formalizing requirements of practical systems. In these chapters, we devise methods that enable the derivation of the queries to be posed to the model checker in a way that the tool has a higher chance to converge on them to a definite answer, and at the same time, a positive answer to all queries together implies correctness of the system with respect to the high-level requirement.

The organization of the dissertation with respect to the challenges outlined in Section 1.2 is summarized in Table 1.1.

Table 1.1: Organization of the dissertation

Background	Chapter 2	We present the theoretical background of our work and define the notations used throughout the dissertation.
Challenge 1	Chapter 3	We introduce THETA, a generic, modular and configurable model checking framework for abstraction refinement-based reachability checking of different formalisms.
	Chapter 4	We present an algorithmic framework for the lazy abstraction based location reachability checking of timed automata with discrete variables.
Challenge 2	Chapter 5	We propose abstraction refinement strategies for the location reachability checking problem of timed automata based on interpolation for zones over clock variables.
Challenge 3	Chapter 6	We propose abstraction refinement strategies for the location reachability problem of timed automata with discrete variables based on visible variables abstraction for discrete variables.
Challenge 4	Chapter 7	We propose the calendar system formalism that allows convenient modeling of the core protocols of communicating real-time systems, and an extension of k-induction based techniques to support the verification of both safety and liveness properties of calendar systems.
	Chapter 8	We devise an approach for the verification of real-time protocols which combines the decomposition of the temporal specification with abstraction.
Summary	Chapter 9	We conclude our work by summarizing the contributions of this dissertation.

Chapter **2**

Background

In this chapter, we summarize the theoretical background of our work. Moreover, we define the notation used throughout the dissertation.

2.1 Transition Systems

For a wide range of modeling formalisms, an operational semantics is defined in terms of transition systems.

> **Definition 2.1 (Transition system).** *A transition system (TS for short) is a tuple* $\mathcal{S} = (S, A, T, I)$ *where*
> - S *is a set of* states*,*
> - A *is a set of* actions*,*
> - $T \subseteq S \times A \times S$ *is the* transition relation*, and*
> - $I \subseteq S$ *is the set of* initial states.
>
> $\mathcal{S}$ *is* finite *iff* S *and* A *are finite. We will denote by* $s \xrightarrow{\alpha} s'$ *iff* $(s, \alpha, s') \in T$*, and by* $s \to s'$ *iff* $s \xrightarrow{\alpha} s'$ *for some action* $\alpha \in A$*. We will say that an action* $\alpha \in A$ *is* enabled *from a state* $s \in S$ *iff* $s \xrightarrow{\alpha} s'$ *for some state* $s' \in S$*, otherwise it is* disabled*. An action* $\alpha \in A$ *is enabled from a set of states* $S' \subseteq S$ *iff* α *is enabled from some state* $s \in S'$*. A state* $s \in S$ *is* terminal *iff no action* $\alpha \in A$ *is enabled from it.* ▪

Sometimes, the formalism is extended by a labeling function $L : S \to \mathcal{P}(AP)$ over some set of atomic propositions AP to express observable properties of system states [BK08]. Throughout this dissertation, we assume that the only observable property of the state is the state itself, and are thus going to omit state labels to simplify exposition. Moreover, we are often going to abstract over the structure of states and properties expressed over them, and will write $s \models \varphi$ to express that state $s \in S$ satisfies some property φ over S. In particular cases, it should be clear from the context what sort of objects s and φ are, and how the relation $\models$ is defined. (For example, s might be a first order interpretation over some signature, and φ a ground first order formula over the same signature.)

> **Definition 2.2 (Run).** *A* finite run *of length* n *of a transition system is an alternating sequence of states and actions of the form* $\rho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n$ *such that* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ *for all* $0 \leq i < n$*. An* infinite run *of a transition system is an alternating sequence of states and actions of the form* $\rho = s_0\alpha_1 s_1\alpha_2 \ldots$ *such that* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ *for all* $i \geq 0$*. A* run *of a transition system is either a*

finite run, or an infinite run. A run is initial *iff* $s_0 \in I$*. A run is* maximal *if it is an infinite run, or if it is a finite run with* s_n *terminal.* ▪

For convenience, we will denote runs as $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$. A state $s \in S$ is *reachable* iff there exists an initial run such that $s_n = s$. We are going to denote the set of reachable states of a transition system $\mathcal{S}$ by $\mathit{Reach}(\mathcal{S})$.

Definition 2.3 (Trace). *Let* $\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ *be a run. We will call the sequence of states* $\tau = s_0 s_1 \ldots$ *the trace induced by run* ρ*. If* ρ *is finite / infinite / initial / maximal, then* τ *too is called* finite / infinite / initial / maximal*, respectively.* ▪

Let $\mathit{Traces}(\mathcal{S}) = \{\tau \mid \tau \text{ is a maximal initial trace of } \mathcal{S}\}$.

Definition 2.4 (Path). *A finite or infinite sequence of actions* π *is a* path*. Let* $\pi = \alpha_1 \alpha_2 \ldots$*. If there exists an initial run* $\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$*, then* π *is called* feasible*, otherwise it is* infeasible*. If* π *is feasible, we will call such a run* ρ *the* run induced by path π*. Similarly, if* π *is feasible,* ρ *is the run* π *induces, and* τ *is the trace* ρ *induces, then* τ *will also be referred to as the* trace induced by path π. ▪

2.2 Linear-Time Properties

Without loss of generality, assume that $\mathcal{S}$ is such that s is not terminal for all $s \in S$, and thus $\mathit{Traces}(\mathcal{S}) \subseteq S^\omega$. (Each system can be transformed to this form by angelic completion [Tre08] where a transition $s \xrightarrow{\epsilon} s$ is introduced for any terminal state $s \in S$). In this context, we are going to treat $\mathit{Traces}(\mathcal{S})$ as a language over S, and refer to sequences $\sigma \in S^\omega$ as *words* accordingly.

Linear-time properties define the correct traces of a system. That is, given a linear-time property $P \subseteq S^\omega$ over a transition system $\mathcal{S}$, we say that $\mathcal{S}$ satisfies P, denoted by $\mathcal{S} \models P$, iff $\mathit{Traces}(\mathcal{S}) \subseteq P$. Any linear time property P can be decomposed as $P = P_{\mathit{safe}} \cup P_{\mathit{live}}$, where P_{safe} is a so-called *safety property*, and P_{live} is a *liveness property* [AS85].

Definition 2.5 (Safety property). *A linear time property* P_{safe} *over* $\mathcal{S}$ *is a* safety property *iff for all words* $\sigma \in S^\omega \setminus P_{\mathit{safe}}$ *there exists a finite prefix* $\hat{\sigma}$ *of* σ *such that* $P_{\mathit{safe}} \cap \{\hat{\sigma}\sigma' \mid \sigma' \in S^\omega\} = \emptyset$*. Such a word* $\hat{\sigma}$ *is called a* bad prefix *for* P_{safe}. ▪

Definition 2.6 (Liveness property). *A linear time property* P_{live} *over* $\mathcal{S}$ *is a* liveness property *iff for all finite words* $\hat{\sigma} \in S^*$ *there exists an infinite word* $\sigma' \in S^\omega$ *such that* $\hat{\sigma}\sigma' \in P_{\mathit{live}}$. ▪

The only linear-time property that is both a safety and a liveness property is S^ω. (For let P be a property that is both a safety and a liveness property, and assume $\sigma \notin P$. As P is a safety property, there exists a bad prefix $\hat{\sigma}$ of σ. As P is also a liveness property, there exists a word σ' such that $\hat{\sigma}\sigma' \in P$. But then $\hat{\sigma}$ is not a bad prefix, a contradiction.)

The simplest safety properties are so-called invariant properties that require some property to hold for all states along a trace.

Definition 2.7 (Invariant property). *A linear time property P_{inv} over $\mathcal{S}$ is an* invariant property *iff there exists a property ϕ over states S such that $P_{inv} = \{s_0 s_1 \ldots \in S^\omega \mid s_i \models \phi$ for all $i \in \mathbb{N}\}$. Here, ϕ is called the* invariant condition.

Clearly, invariants are safety properties, as for a word $s_0 s_1 \ldots \in S^\omega \setminus P_{inv}$ with $s_i \not\models \phi$ for some $i \in \mathbb{N}$, the word $s_0 s_1 \ldots s_i$ is a bad prefix.

An example for a liveness property is a persistence property that asserts that from some moment on a condition holds continuously.

Definition 2.8 (Persistence property). *A linear time property P_{pers} over $\mathcal{S}$ is a* persistence property *iff there exists a property ϕ over set of states S such that $P_{pers} = \{s_0 s_1 \ldots \in S^\omega \mid$ there exists $i \in \mathbb{N}$ such that $s_j \models \phi$ for all $j \geq i\}$. Here, ϕ is called the* persistence condition.

It is easy to see that persistence properties are liveness properties, as given a finite word $\hat{\sigma} \in S^*$, we have $\hat{\sigma}\sigma' \in P_{pers}$ for some $\sigma' = s_0 s_1 \ldots$ with $s_i \models \phi$ for all $i \in \mathbb{N}$.

2.3 ω-Regular Model Checking

An important class of linear-time properties is the class of *ω-regular properties*. An ω-regular property is a linear time property that is also an *ω-regular language*. An important property of ω-regular languages is closure under complementation [Büc62; McN66; Saf88; Kla02]. We are going to define this class of languages using so-called *Büchi automata* [Büc62], as the class of ω-regular languages coincides with class of languages accepted by Büchi automata [McN66].

Definition 2.9 (Nondeterministic Büchi automaton). *A* nondeterministic Büchi automaton *(NBA for short) is a tuple $(Q, \Sigma, \Delta, Q_0, F)$ where*

- *Q is a finite set of states,*
- *Σ is a set of symbols, called the alphabet,*
- *$\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation,*
- *$Q_0 \subseteq Q$ is the set of initial states, and*
- *$F \subseteq Q$ is the acceptance condition.*

We will denote by $q \xrightarrow{\alpha} q'$ iff $(q, \alpha, q') \in \Delta$. An automaton is nonblocking *iff for all $q \in Q$ and $\alpha \in \Sigma$ there exists $q' \in Q$ such that $q \xrightarrow{\alpha} q'$.*

Definition 2.10 (Run of an NBA). *Given an input word $\sigma = A_0 A_1 A_2 \ldots$, a* run *of an NBA over a word σ is a sequence of states $q_0 q_1 q_2 \ldots$ such that $q_0 \in Q_0$ and $(q_i, A_i, q_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$. The run is* accepting *if $q_i \in F$ for infinitely many $i \in \mathbb{N}$.*

Definition 2.11 (Language accepted by an NBA). *A word σ is accepted by an NBA $\mathcal{B}$ iff $\mathcal{B}$ has an accepting run over σ. The* language accepted by the automaton *$\mathcal{B}$ is $\mathcal{L}(\mathcal{B}) = \{\sigma \mid \sigma$ is accepted by $\mathcal{B}\}$.*

Applying the above notation, in our case, an ω-regular property P is such that $P = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$ over a set of states S of a transition system $\mathcal{S}$. In the following, without loss of generality, we are going to assume NBAs to be nonblocking. (A blocking NBA can be easily transformed to a nonblocking NBA that accepts the same language by introducing a transition to a "sink" state for all undefined transitions.) Moreover, given it makes sense to do so, in graphical notation we are going to admit a formula φ of some logic to label an edge from a state q to q' for conciseness, encoding that for all symbols $\alpha \in \Sigma$ such that $\alpha \models \varphi$ there is a transition $(q, \alpha, q') \in \Delta$ of the automaton. (For example, α might be a first order interpretation over some signature, encoding a state of a transition system, and φ a ground first order formula over the same signature.)

Example (Infinitely often p). *Let p be a formula, expressing some property over Σ, and $\mathcal{B} = (Q, \Sigma, \Delta, q_0, F)$ the Büchi automaton depicted in Figure 2.1. Here, $Q = \{q_0, q_1\}$ and $F = \{q_1\}$ and $\Delta = \{(q, \alpha, q_0) \mid \alpha \models \neg p\} \cup \{(q, \alpha, q_1) \mid \alpha \models p\}$. Moreover, $\mathcal{L}(\mathcal{B}) = \{\alpha_0\alpha_1 \ldots \mid$ for all $i \geq 0$ there exists $j \geq i$ such that $\alpha_j \models p\}$.*

Example (Eventually forever p). *Let p be a formula, expressing some property over Σ, and $\mathcal{B} = (Q, \Sigma, \Delta, q_0, F)$ the Büchi automaton depicted in Figure 2.2. Then, $Q = \{q_0, q_1, q_2\}$ and $F = \{q_1\}$ and $\Delta = \{(q, \alpha, q) \mid q \neq q_1\} \cup \{(q, \alpha, q_1) \mid q \neq q_2$ and $\alpha \models p\} \cup \{(q_1, \alpha, q_2) \mid \alpha \models \neg p\}$. Moreover, $\mathcal{L}(\mathcal{B}) = \{\alpha_0\alpha_1 \ldots \mid$ there exists $i \geq 0$ such that for all $j \geq i$ we have $\alpha_j \models p\}$.*

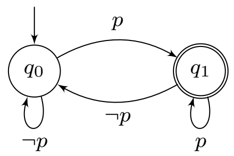


Figure 2.1: Infinitely often p

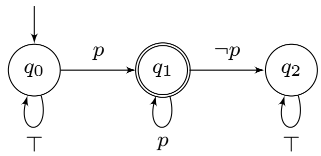


Figure 2.2: Eventually forever p

Definition 2.12 (Product of a TS and an NBA). *Let $\mathcal{S} = (S, A, T, I)$ and $\mathcal{B} = (Q, \Sigma, \Delta, Q_0, F)$ with $\Sigma = S$. Then $\mathcal{S} \otimes \mathcal{B}$ is a transition system $\mathcal{S}' = (S', A', T', I')$ such that*

- $S' = S \times Q$,
- $A' = A$,
- $I' = \left\{(s_0, q) \mid s_0 \in I \text{ and } q_0 \xrightarrow{s_0} q \text{ for some } q_0 \in Q_0\right\}$, *and*
- *the transition relation T' is defined by the following rule.*

$$\frac{s \xrightarrow{\alpha} s' \qquad q \xrightarrow{s'} q'}{(s, q) \xrightarrow{\alpha} (s', q')}$$

The language $\mathit{Traces}(\mathcal{S} \otimes \mathcal{B})$ encodes the runs of $\mathcal{B}$ over the traces of $\mathcal{S}$. According to the automata-theoretic approach to model checking [VW86], the product system enables the checking of ω-regular properties as follows.

Proposition 1. *Let $\mathcal{S}$ be a transition system with sets of states S. Let P be an ω-regular property over $\mathcal{S}$, and $\mathcal{B}$ a Büchi automaton with set of states Q, set of accepting states F, and with $\mathcal{L}(\mathcal{B}) = S^\omega \setminus P$. Let moreover P_{pers} be the persistence property defined by condition ϕ over $S \times Q$ such that $(s, q) \models \phi$ iff $q \notin F$ for all $s \in S$ and $q \in Q$. Then $\mathcal{S} \models P$ iff $\mathcal{S} \otimes \mathcal{B} \models P_{pers}$.*

Here, P_{pers} encodes that along each run of $\mathcal{B}$ over some trace of $\mathcal{S}$, eventually only nonaccepting states are reached, and thus accepting states occur only finitely many times. For finite state spaces, this induces a special circle detection problem, as in that case each infinite run can be represented by a finite prefix forming a lasso [Bie+99]. A counterexample for the property is then a lasso-shaped run for which there is an accepting state inside the loop of the lasso, and the absence of such lassos guarantees the property.

2.4 Linear Temporal Logic

Linear temporal logic [Pnu77], LTL for short, is a widely used logic for specifying linear time properties.

Definition 2.13 (Syntax). *An LTL formula over a set of states S is defined by the grammar*

$$\varphi \quad ::= \quad \top \;\big|\; P \;\big|\; \neg\varphi \;\big|\; \varphi_1 \wedge \varphi_2 \;\big|\; \mathsf{X}\varphi \;\big|\; \varphi_1\, \mathsf{U}\, \varphi_2$$

where P is some property over S. ▪

Boolean connectives $\bot$ and $\vee$ and $\rightarrow$ and $\leftrightarrow$ are defined in terms of $\top$ and $\neg$ and $\wedge$ as usual. Moreover, let $\mathsf{F}\varphi \doteq \top\, \mathsf{U}\, \varphi$ and $\mathsf{G}\varphi \doteq \neg\mathsf{F}\neg\varphi$.

Given a sequence of states $\sigma = (s_0 s_1 \ldots s_i s_{i+1} \ldots)$, let σ^i denote the suffix $(s_i s_{i+1} \ldots)$.

Definition 2.14 (Semantics). *Let $\sigma = s_0 s_1 s_2 \ldots$ a sequence of states. The satisfaction relation $\models$ for LTL is defined as follows.*

$$\begin{array}{lllll}
\sigma & \models & \top & & \\
\sigma & \models & P & \textit{iff} & s_0 \models P \\
\sigma & \models & \neg\varphi & \textit{iff} & \sigma \not\models \varphi \\
\sigma & \models & \varphi_1 \wedge \varphi_2 & \textit{iff} & \sigma \models \varphi_1 \textit{ and } \sigma \models \varphi_2 \\
\sigma & \models & \mathsf{X}\varphi & \textit{iff} & \sigma^1 \models \varphi \\
\sigma & \models & \varphi_1\, \mathsf{U}\, \varphi_2 & \textit{iff} & \textit{there exists } j \geq 0 \textit{ such that } \sigma^j \models \varphi_2 \textit{ and} \\
 & & & & \textit{for all } 0 \leq i < j \textit{ we have } \sigma^i \models \varphi_1
\end{array}$$
▪

Accordingly,

$$\begin{array}{lllll}
\sigma & \models & \mathsf{G}\varphi & \text{iff} & \text{for all } i \geq 0 \text{ we have } \sigma^i \models \varphi \\
\sigma & \models & \mathsf{F}\varphi & \text{iff} & \text{there exists } i \geq 0 \text{ such that } \sigma^i \models \varphi
\end{array}$$

An invariant property with condition ϕ is thus expressed by $\mathsf{G}\phi$. A persistence property with condition ϕ is expressed by formula $\mathsf{FG}\phi$. According to the semantics of the temporal connectives F and G,

$$\sigma \quad \models \quad \mathsf{FG}\varphi \quad \text{iff} \quad \text{there exists } i \geq 0 \text{ such that for all } j \geq i \text{ we have } \sigma^j \models \varphi$$

Similarly, we get

$$\sigma \quad \models \quad \mathsf{GF}\varphi \quad \text{iff} \quad \text{for all } i \geq 0 \text{ there exists } j \geq i \text{ such that } \sigma^j \models \varphi$$

We will denote the language induced by an LTL formula φ as $Words(\varphi) = \{\sigma \mid \sigma \models \varphi\}$. Given a transition system $\mathcal{S}$ and an LTL-formula φ over S, the model checking problem is hence to show that $Traces(\mathcal{S}) \subseteq Words(\varphi)$, or give a counterexample.

LTL corresponds to the class of star-free languages (see e.g. [Coh91]), a proper subclass of ω-regular languages. Thus the model checking problem for LTL can be solved by translating the formula to check to a Büchi automaton accepting the corresponding language [WVS83; VW94]. Formally, $\mathcal{S} \models \varphi$ iff $\mathcal{S} \otimes \mathcal{B} \models P_{pers}$ where $\mathcal{L}(\mathcal{B}) = Words(\neg\varphi)$ and P_{pers} is as defined in Proposition 1.

Example. *Let $\mathcal{B}_1$ be the Büchi automaton depicted on Figure 2.1, and $\mathcal{B}_2$ be the Büchi automaton depicted on Figure 2.2. Then $\mathcal{L}(\mathcal{B}_1) = Words(\mathsf{GF}p)$, and $\mathcal{L}(\mathcal{B}_2) = Words(\mathsf{FG}p)$, respectively.*

2.5 Timed Automata with Discrete Variables

In the area of modeling and verifying time-dependent behavior, timed automata [AD94] is the most prominent formalism. To make the specification of practical systems more convenient, the traditional formalism is often extended with various syntactic and semantic constructs, in particular with the handling of discrete variables. This section describes the formalization of one such extension, what we call a timed automaton with discrete variables [*c*11]. Results in Chapter 4 and Chapter 6 are based on this formalization of timed automata. Results in Chapter 5 have been developed for classical timed automata [*c*9], but for a more uniform exposition, we present the results adapted to the more general definition.

2.5.1 Valuations

Let C be a set of *clock variables* over $\mathbb{R}_{\geq 0}$, and D a set of *data variables* over $\mathbb{Z}$. Let $V = C \cup D$ denote the set of all variables.

A *clock constraint* is a formula $\varphi \in Constr_C$ that is a conjunction of atoms of the form $c \prec m$ and $c_i - c_j \prec m$ where $c, c_i, c_j \in C$ and $m \in \mathbb{Z}$ and $\prec \; \in \{<, \leq, >, \geq, \doteq\}$. In the latter case, if $i \neq j$, then a constraint is called a *diagonal constraint.* A *data constraint* is a well-formed formula $\varphi \in Constr_D$ built from variables in D and arbitrary function and predicate symbols interpreted over $\mathbb{Z}$. Let $Constr = Constr_C \cup Constr_D$ denote the set of all constraints.

A *clock update* (clock reset) is an assignment $u \in Update_C$ of the form $c := m$ where $c \in C$ and $m \in \mathbb{Z}$. A *data update* is an assignment $u \in Update_D$ of the form $d := t$ where $d \in D$ and t is a term built from variables in D and function symbols interpreted over $\mathbb{Z}$. Let $Update = Update_C \cup Update_D$ denote the set of all updates.

The set of variables appearing in a term t (in a formula φ) is denoted by $\mathsf{vars}(t)$ (by $\mathsf{vars}(\varphi)$). Similarly, the set of variables occurring in an update is denoted by $\mathsf{vars}(u)$, that is, $\mathsf{vars}(x := t) = \mathsf{vars}(t) \cup \{x\}$.

A *valuation* over a finite set of variables is a function that maps variables to their respective domains. We will denote by $\mathcal{V}(X)$ the set of valuations over a set of variables X. Throughout the dissertation we will allow partial functions as valuations. We will denote by $\mathsf{def}(\sigma)$ the domain of definition of a valuation σ, that is, $\mathsf{def}(\sigma) = \{x \mid \sigma(x) \neq \bot\}$. We extend valuations to range over terms and formulas the usual way, with the possibility that the value of a term is undefined over a valuation.

We will denote by $\sigma \models \varphi$ iff formula φ is satisfied under valuation σ. Let $[\![\varphi]\!]$ stand for the set of models of a formula φ, formally defined as $[\![\varphi]\!] = \{\sigma \in (\mathcal{V} \circ \mathsf{vars})(\varphi) \mid \sigma \models \varphi\}$, where $\circ$ denotes function composition as usual. Given a valuation σ, we denote by $\mathsf{form}(\sigma)$ the formula characterizing the valuation, that is, $\mathsf{form}(\sigma) = \bigwedge_{x \in \mathsf{def}(\sigma)} x \doteq \sigma(x)$.

Remark 1. *Note that in the context of partial valuations, $\sigma \models \neg\varphi$ is a strictly stronger statement than $\sigma \not\models \varphi$. For example, $\{x \hookleftarrow 1\} \not\models y \doteq 1$ but it is not the case that $\{x \hookleftarrow 1\} \models y \not\doteq 1$.*

Let $\sigma \preceq \sigma'$ iff $\sigma(x) = \sigma'(x)$ for all $x \in \mathsf{def}(\sigma')$. Moreover, let $A \preceq B$ iff for all $\sigma \in A$ there exists $\sigma' \in B$ such that $\sigma \preceq \sigma'$. Clearly, $\preceq$ is a partial order over sets of valuations. We will denote the restriction of valuation σ to a set of variables X by $\sigma\!\restriction_X$, that is, $(\sigma\!\restriction_X)(x) = \sigma(x)$ if $x \in X$ and $(\sigma\!\restriction_X)(x) = \bot$ if $x \notin X$. We lift the notion to sets of valuations with the obvious meaning. Let moreover $(\!|\sigma|\!) = \{\sigma' \in \mathcal{V}(V) \mid \sigma' \preceq \sigma\}$, also defined for sets of valuations in the obvious way.

We state the following lemmas (without proof).

Lemma 1. $\sigma \preceq \sigma' \Rightarrow \sigma' \models \varphi \Rightarrow \sigma \models \varphi$

Lemma 2. $\sigma \preceq \sigma' \Leftrightarrow \sigma \models \mathsf{form}(\sigma')$

Lemma 3. $A \preceq B \Rightarrow A\!\restriction_X \preceq B\!\restriction_X$

We will denote by $\otimes$ the partial function over valuations that is defined as

$$(\sigma \otimes \sigma')(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathsf{def}(\sigma) \\ \sigma'(x) & \text{if } x \in \mathsf{def}(\sigma') \\ \bot & \text{otherwise} \end{cases}$$

if $\sigma(x) = \sigma'(x)$ for all $x \in \mathsf{def}(\sigma) \cap \mathsf{def}(\sigma')$, and is undefined otherwise. We extend this function to sets of valuations in both parameters in the obvious way.

Finally, given a valuation σ and an update $x := t$, we denote by $\sigma\{x := t\}$ the valuation σ' such that $\sigma'(x) = \sigma(t)$ and $\sigma'(x') = \sigma(x')$ for all $x' \neq x$. For a sequence of updates, let $\sigma\{\epsilon\} = \sigma$ and $\sigma\{u \cdot \mu\} = \sigma\{u\}\{\mu\}$, where u is an update and μ is a sequence of updates.

2.5.2 Timed Automata

Definition 2.15 (Syntax). *Syntactically, a timed automaton with discrete variables is a tuple $\mathcal{A} = (L, C, D, T, \ell_0)$ where*

- *L is a finite set of locations,*
- *C is a finite set of continuous clock variables over $\mathbb{R}_{\geq 0}$,*
- *D is a finite set of discrete data variables over $\mathbb{Z}$,*
- *$T \subseteq L \times \mathcal{P}(Constr) \times Update^* \times L$ is a finite set of transitions, where for a transition (ℓ, G, μ, ℓ'), the set $G \subseteq Constr$ is a set of guards, and $\mu \in Update^*$ is a sequence of updates, and*
- *$\ell_0 \in L$ is the initial location.* ▪

Remark 2. *According to the above definition, clearly $C \cap D = \emptyset$. Note that given a guard $g \in G$, either $\mathsf{vars}(g) \subseteq C$, or $\mathsf{vars}(g) \subseteq D$. Similarly, given an update u, either $\mathsf{vars}(u) \subseteq C$, or $\mathsf{vars}(u) \subseteq D$.*

Example (Fischer's Protocol for Mutual Exclusion). *As an example, consider the automaton Fischer$_i$, depicted in Figure 2.3. Given some $i > 0$ and $a, b \in \mathbb{N}$, this automaton is formally defined as $Fischer_i = (L_i, C_i, D_i, T_i, \ell_i^0)$ where*

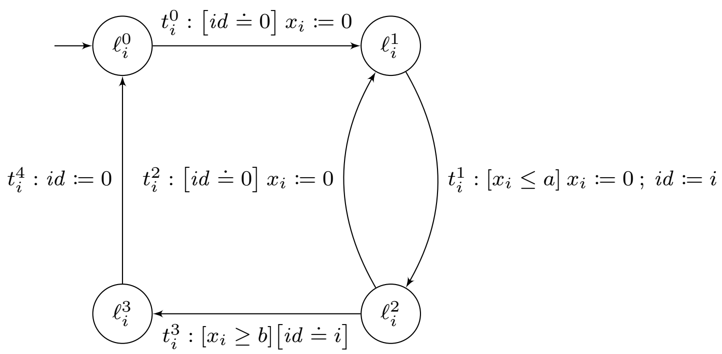


Figure 2.3: Timed automaton model *Fischer$_i$*

- $L_i = \{\ell_i^0, \ell_i^1, \ell_i^2, \ell_i^3\}$,
- $C_i = \{x_i\}$,
- $D_i = \{id\}$, *and*
- $T_i = \{t_i^0, t_i^1, t_i^2, t_i^3, t_i^4\}$ *where*
  - $t_i^0 = (\ell_i^0, \{id \doteq 0\}, x_i := 0, \ell_i^1)$,
  - $t_i^1 = (\ell_i^1, \{x_i \leq a\}, (x_i := 0, id := i), \ell_i^2)$,
  - $t_i^2 = (\ell_i^2, \{id \doteq 0\}, x_i := 0, \ell_i^1)$,
  - $t_i^3 = (\ell_i^2, \{x_i \geq b, id \doteq i\}, \varepsilon, \ell_i^3)$, *and*
  - $t_i^4 = (\ell_i^3, \emptyset, id := 0, \ell_i^0)$.

Here, $x_i \leq a$ *is a clock constraint,* $id \doteq 0$ *is a data constraint,* $x_i := 0$ *is a clock update, and* $id := i$ *is a data update.*

It is possible to compose automata to obtain a more complex system using interleaving.

Definition 2.16 (Interleaving of Timed Automata). *Let* $\mathcal{A}_i = (L_i, C_i, D_i, T_i, \ell_i^0)$. *Then the interleaving of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *is* $\mathcal{A}_1 \parallel \mathcal{A}_2 = (L, C, D, T, \ell_0)$ *where*
- $L = L_1 \times L_2$,
- $C = C_1 \cup C_2$,
- $D = D_1 \cup D_2$,
- $\ell_0 = (\ell_1^0, \ell_2^0)$, *and*
- T *is defined by the following rules.*

$$\frac{(\ell_1, G, \mu, \ell_1') \in T_1 \qquad \ell_2 \in L_2}{((\ell_1, \ell_2), G, \mu, (\ell_1', \ell_2)) \in T} \ \textit{transition of } \mathcal{A}_1$$

$$\frac{(\ell_2, G, \mu, \ell_2') \in T_2 \qquad \ell_1 \in L_1}{((\ell_1, \ell_2), G, \mu, (\ell_1, \ell_2')) \in T} \ \textit{transition of } \mathcal{A}_2$$

Definition 2.17 (Semantics). *Let* σ_0 *be the unique total function* $\sigma_0 : V \mapsto \{0\}$. *The operational semantics of a timed automaton is given by a labeled transition system with initial state* (ℓ_0, σ_0) *and two kinds of transitions:*

- Delay: $(\ell, \sigma) \xrightarrow{\delta} (\ell', \sigma')$ *for some real number* $\delta \geq 0$ *where* $\ell' = \ell$ *and* $\sigma' = delay_\delta(\sigma)$ *with*

$$delay_\delta(\sigma)(x) = \begin{cases} \sigma(x) + \delta & \text{if } x \in C \\ \sigma(x) & \text{otherwise} \end{cases}$$

- Action: $(\ell, \sigma) \xrightarrow{t} (\ell', \sigma')$ *for some transition* $t = (\ell, G, \mu, \ell')$ *where* $\sigma' = action_t(\sigma)$ *with*

$$action_t(\sigma) = \begin{cases} \bot & \text{if } \sigma \models \neg g \text{ for some } g \in G \\ \sigma\{\mu\} & \text{otherwise} \end{cases}$$

In case $D = \emptyset$, the above definition for semantics coincides with the semantics of timed automata in the usual sense [BY04]. Throughout the dissertation, we will refer to a timed automaton with discrete variables simply as a timed automaton.

We will use the notation $\mathcal{C} = \mathcal{V}(V)$, and refer to a valuation $\sigma \in \mathcal{C}$ as a *concrete state*. A *state* of a timed automaton is a state of its semantics, that is, a pair (ℓ, σ) where $\ell \in L$ and $\sigma \in \mathcal{C}$. A *run* (*path*) of a timed automaton is a run (path) of its semantics. A location $\ell \in L$ is *reachable* iff state (ℓ, σ) is reachable for some concrete state $\sigma \in \mathcal{C}$. Clearly, if a location is reachable then it is reachable along a run of the form $\cdot \xrightarrow{\delta_0} (\ell_0, \sigma_0') \xrightarrow{t_1} \cdot \xrightarrow{\delta_1} (\ell_1, \sigma_1) \xrightarrow{t_2} \cdot \xrightarrow{\delta_2} \ldots \xrightarrow{t_k} \cdot \xrightarrow{\delta_k} (\ell_k, \sigma_k)$. This observation enables the definition of a symbolic semantics for timed automata as follows.

Definition 2.18 (Symbolic semantics). *Let* $\Sigma_0 = \{delay_\delta(\sigma_0) \mid \delta \geq 0\}$, *that is, the set of concrete states reachable from* σ_0 *by a delay transition. The symbolic semantics of a timed automaton is a labeled transition system with initial state* (ℓ_0, Σ_0) *and transitions of the form* $(\ell, \Sigma) \xrightarrow{t} (\ell', \Sigma')$ *where* $t = (\ell, \cdot, \cdot, \ell')$ *and* $\Sigma' = post_t(\Sigma)$ *with the concrete post-image operator*

$$post_t(\sigma) = \{(delay_\delta \circ action_t)(\sigma) \mid \delta \geq 0\},$$

defined for paths as $post_\epsilon = id$ *and* $post_{t \cdot \pi} = post_\pi \circ post_t$.

We will refer to a pair (ℓ, Σ) with $\ell \in L$ and $\Sigma \subseteq \mathcal{C}$ as a *symbolic state.*

Definition 2.19 (Symbolic run). *A* symbolic run *of a timed automaton is an initial run of its symbolic semantics* $(\ell_0, \Sigma_0) \xrightarrow{t_1} (\ell_1, \Sigma_1) \xrightarrow{t_2} \ldots \xrightarrow{t_k} (\ell_k, \Sigma_k)$ *where* $\Sigma_k \neq \emptyset$.

Example. *The following is a symbolic run of Fischer*$_1 \parallel$ *Fischer*$_2$.

$$\big((\ell_1^0, \ell_2^0), \{\{id \leftarrow 0, x_1 \leftarrow v, x_2 \leftarrow v\} \mid v \geq 0\}\big)$$
$$\xrightarrow{t_1^0}$$
$$\big((\ell_1^1, \ell_2^0), \{\{id \leftarrow 0, x_1 \leftarrow v_1, x_2 \leftarrow v_2\} \mid 0 \leq v_1 \leq v_2\}\big)$$
$$\xrightarrow{t_2^0}$$
$$\big((\ell_1^1, \ell_2^1), \{\{id \leftarrow 0, x_1 \leftarrow v_1, x_2 \leftarrow v_2\} \mid 0 \leq v_2 \leq v_1\}\big)$$
$$\xrightarrow{t_1^1}$$
$$\big((\ell_1^2, \ell_2^1), \{\{id \leftarrow 1, x_1 \leftarrow v_1, x_2 \leftarrow v_2\} \mid v_2 \geq 0, 0 \leq v_2 - v_1 \leq a\}\big)$$
$$\xrightarrow{t_1^3}$$
$$\big((\ell_1^3, \ell_2^1), \{\{id \leftarrow 1, x_1 \leftarrow v_1, x_2 \leftarrow v_2\} \mid v_1 \geq b, 0 \leq v_2 - v_1 \leq a\}\big)$$

Let $\sigma \in \{\{id \leftarrow\!\shortmid 1, x_1 \leftarrow\!\shortmid v_1, x_2 \leftarrow\!\shortmid v_2\} \mid v_1 \geq b, 0 \leq v_2 - v_1 \leq a\}$, *and assume* $a < b$. *Then the run described above can not be extended by the transition* t_2^2, *as in this case,* $\sigma \models x_2 > a$, *and thus* $action_{t_2^2}(\sigma) = \bot$.

Proposition 2. *For a timed automaton, a location* $\ell \in L$ *is reachable iff there exists a symbolic run with* $\ell_k = \ell$ *[DT98].*

Let $pre_t = post_t^{-1}$ and $post_t^X(\Sigma) = post_t(\Sigma)\restriction_X$ for $X \in \{C, D\}$. Let moreover $pre_t^C = (post_t^C)^{-1}$. Furthermore, let $\nu_0 = \Sigma_0\restriction_D$ and $Z_0 = \Sigma_0\restriction_C$.

Remark 3. *As a consequence of Remark 2, it can be shown that in general, a symbolic state* (ℓ, Σ) *occurring in a symbolic run of timed automaton is such that* $\Sigma = \nu \otimes Z$, *where* $\nu = \Sigma\restriction_D$ *is a data valuation, and* $Z = \Sigma\restriction_C$ *is a special set of clock valuations, called a zone (see Section 5.2). Moreover,* $post_t(\nu \otimes Z) = post_t^D(\nu) \otimes post_t^C(Z)$.

Clearly, a transition $t = (\ell, \cdot, \cdot, \ell')$ is enabled from a symbolic state (ℓ, Σ) iff $post_t(\Sigma) \neq \emptyset$. Moreover, given a path $\pi = t_1 t_2 \ldots t_n$ such that $t_i = (\ell_{i-1}, \cdot, \cdot, \ell_i)$ for all $0 < i \leq n$, clearly, π is feasible iff $post_\pi(\Sigma_0) \neq \emptyset$. Later on in the dissertation, we are often going to use these terms in the more specific sense, as the necessary assumptions are going to hold by construction. This enables us to disregard the location component in a symbolic state or a symbolic run, simplifying exposition. Moreover, we define the following similar terms.

Definition 2.20 (Data-feasible path). *We will say that a path* π *is* data-feasible *iff* $post_\pi^D(\nu_0) \neq \emptyset$ *otherwise it is* data-infeasible. ∎

Definition 2.21 (Clock-feasible path). *We will say that a path* π *is* clock-feasible *iff* $post_\pi^C(Z_0) \neq \emptyset$, *otherwise it is* clock-infeasible. ∎

Remark 4. *Let* $\pi = t_1 t_2 \ldots t_n$ *such that* $t_i = (\ell_{i-1}, \cdot, \cdot, \ell_i)$ *for all* $0 < i \leq n$. *By Remark 3 and induction,* π *is feasible iff it is data-feasible and clock-feasible.*

Chapter 3

Architecture of a Configurable Model Checking Framework

To tackle state space explosion and make model checking tractable, model checkers typically rely on some sort of abstraction [CC77], where only relevant aspects of system behavior are considered during state space traversal [CGL94]. The abstract system obtained so is then a less complex system whose behavior overapproximates that of the original, concrete system. As a result, if no faulty behavior is present in the abstract model, then neither is there one in the original model. On the other hand, the abstract system might admit false negatives, that is, spurious faulty behavior that is not present in the original system. The key challenge is thus finding the right abstraction granularity that is coarse enough to make model checking efficient, yet fine enough to exclude spurious counterexamples.

Counterexample-guided abstraction refinement (CEGAR) [Cla+00] is a well known, generic approach that automates this process. It is based on an abstraction refinement loop, roughly consisting of the following steps.

1. *Abstract.* Build the abstract model based on the current abstraction granularity.
2. *Check.* Perform model checking on the abstract model. If no counterexample is found, then the original model is correct.
3. *Concretize.* Otherwise, try to concretize the counterexample, that is, check if it corresponds to some execution in the original model. If so, the execution found this way is a counterexample in the original model.
4. *Refine.* Otherwise, the counterexample is spurious. Automatically refine the abstraction by adding details to the analysis, and start over.

There are several model checking tools that implement some variant of the above scheme. Approaches vary in many aspects, including the following.

- *Formalism.* What sort of model does the tool take as input? Examples include simple imperative programs and timed automata.
- *Abstract domain.* What sort of abstraction is the algorithm based on, i.e. what sort of syntactic objects are used to represent abstractions and abstract states, and how do they map to semantics? Examples include *predicates* [GS97; CU98; BPR01], where abstract states are expressed as Boolean formulas (typically restricted to conjunctive literals) over a predefined set of predicates over state variables; and *explicit values* [BL13], where abstract states are projections of concrete states over a set of tracked or visible variables.
- *Abstraction strategy.* How is the abstract state space constructed? When is search pruned, or refinement invoked? These details typically vary between approaches.

- *Refinement strategy.* How are refinements computed? Examples include *weakest precondition* computation [Hen+02], where in case of a spurious counterexample (predicate) analysis is enriched with new predicates from the unsat core of the intersection of the current abstraction and the "bad region", obtained by iteratively computing backwards the weakest formula expressing states that can take the transition towards the error; and *interpolation* [Hen+04], which is a more general approach based on the computation of Craig interpolants [Cra57], formulas that – similarly to the unsat core – certify unsatisfiability but whose atoms might not appear in the weakest precondition and thus might converge better to an invariant. (Also, see [Die+17] for a detailed comparison of these two basic approaches.)

Generally, most tools focus on a specific algorithm and formalism to solve a particular verification task efficiently. However, as new tasks emerge, more generic tools are also needed since the appropriate formalism and algorithm are usually not known initially. Theta[1] is a generic, modular and configurable model checking framework, aiming to support the development and evaluation of abstraction refinement-based algorithms for the reachability analysis of different formalisms. The main distinguishing characteristic of Theta is its architecture that allows the combination of various abstract domains and strategies for abstraction and refinement, applied to models of various formalisms with higher level language frontends.

Theta primarily aims to support researchers by providing a framework where new components and combinations can easily be implemented, evaluated and compared. Concrete tools have also been built for the verification of transition systems, control flow automata and timed automata, combining different abstract domains (including predicates, explicit values and zones) and refinement strategies (including interpolation and unsat cores). Measurement results show strong dependency on the models and analysis components, motivating the need for a configurable framework. Furthermore, we also used Theta for education at our university, where students implemented model checkers using components from the framework.

3.1 Related Tools

Abstraction refinement is a widely used approach for model checking software. Several tools, e.g. Slam [BR01], Blast [Bey+07] and SatAbs [Cla+05] are based on predicate abstraction. Lazy abstraction tools like Impact [McM06] and Wolverine [KW11] use Craig interpolation [McM03] to compute abstractions over the predicate domain without expensive post-image computation. Some tools apply abstraction refinement over domains other than predicates: the tool Dagger [Gul+08] supports refinement for octagon and convex polyhedra domains, and the algorithm Vinta [AGC12] applies abstraction refinement over intervals. Frameworks CPAchecker [BK11] and Ufo [Alb+12] support configurability by the definition of abstract domains, post operators and refinement strategies, but only targeting software models. The LTSmin tool supports various formalisms through its Partitioned Next-State Interface (PINS) [Kan+15], but instead of abstraction refinement, its main focus is on symbolic and parallel model checking algorithms.

Novelty in the Theta framework is that it aims to combine the concept of configurability with formalism independence: the core analysis algorithms can be implemented independently of the input formalisms, and relevant combinations of them can be selected to verify models of several input formalisms. In this chapter we focus on the architecture of Theta and the use cases demonstrating the efficient use of the tools that are derived from the framework.

[1] https://github.com/FTSRG/theta

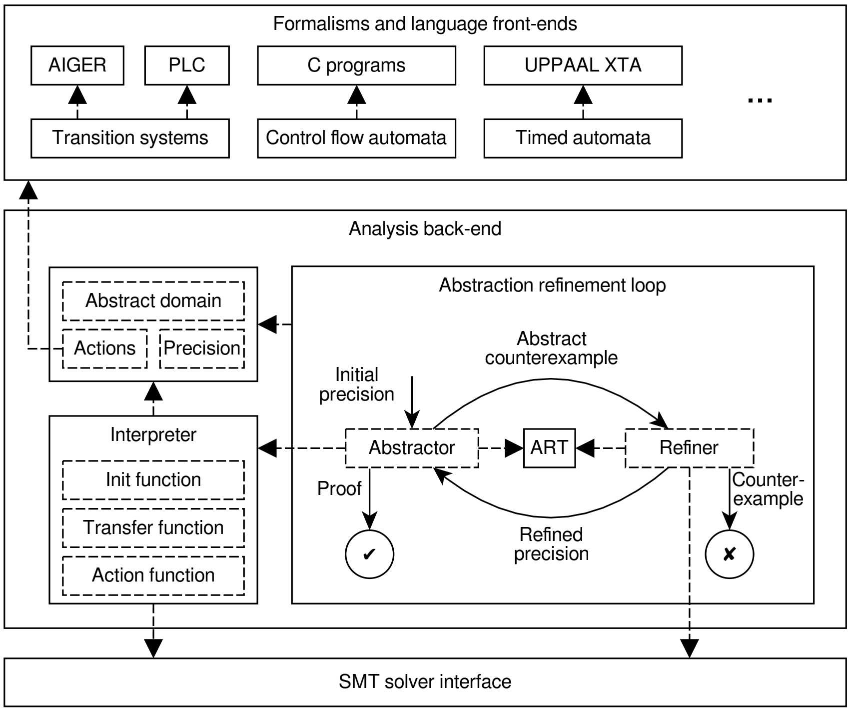


Figure 3.1: Architecture of the THETA framework

3.2 Architecture and Implementation

Figure 3.1 shows the architecture of THETA (with continuous arrows representing data flow, and dashed arrows representing dependence). The main parts of the framework are the formalism and language frontends, the analysis backend and the SMT solver interface.

3.2.1 Formalisms and Language Frontends

One goal of the THETA framework is to enable the analysis of several formalisms. Formalisms are usually low level, mathematical representations based on first order logic formulas and graph like structures. Each formalism supports higher level languages that can be mapped to that particular formalism by a language frontend (consisting of a specific parser and possibly reductions for simplification of the model). Currently, transition systems, control flow automata and timed automata are the supported formalisms with frontends for higher level languages as AIGER, PLC, C programs and UPPAAL XTA models. Section 3.3 describes instantiations of the framework for each of these formalisms.

3.2.2 Analysis Backend

The core of the framework, the analysis backend consists of three main parts: the abstract domain, the interpreter and the abstraction refinement loop for reachability analysis, with only the interpreter being strictly dependent on the formalisms.

Abstract domain. The semantic basis of the analysis is an *abstract domain* with a set of abstract states, its bottom element and a preorder over the states. The accuracy of a given analysis is formally represented by an element of a set of precisions. (As a typical example, the precision might be a set of state variables that the analysis is expected to track – the larger the cardinality of this set is, the more precise the analysis is.) The formalism for which the analysis is performed defines a set of actions, that serve as input to post-image computation.

Interpreter. Given a precision, an *interpreter* defines an abstract operational semantics over the abstract domain and set of actions. The abstract initial states are given by an *init function*. For an action, the abstract successors of a state are computed by a *transfer function*. An *action function* determines for an abstract state a set of actions that are enabled from that state. The interpreter plays an important role in the generality of the framework, as it decouples the notion of abstraction (represented by the abstract domain over which it is defined) from the notion of formalism (represented by the set of actions over which it is defined).

Abstraction refinement loop. The reachability analysis is performed by the *abstraction refinement loop*. As usual for lazy abstraction methods [McM06], its central data structure is an *abstract reachability tree* (ART), with nodes annotated with abstract states that represent overapproximations of reachable states along a given path, and edges annotated with actions. The ART is manipulated by the two main components of the loop. Using an interpreter, the *abstractor* constructs the ART w.r.t the current precision and an abstraction strategy, the latter of which is determined by the following basic operations.

- *Expand.* When should the abstractor expand a node, i.e. grow the tree by computing and adding to the tree all its abstract successors?
- *Cover.* How should the abstractor attempt to prune the search by looking for covering nodes, i.e. nodes that represent abstract states that entail the abstract state of the current node?
- *Terminate.* Under what conditions should the state space exploration terminate?

If no target nodes – nodes that are deemed unsafe based on the input model – are encountered, the constructed ART serves as an evidence for the safety of the input model. Otherwise, given a target node, the *refiner* is invoked to analyze the abstract path for feasibility. If the path is feasible, it is a counterexample to safety. Otherwise, the refiner carries out its refinement strategy to ensure that the analysis can continue without encountering the same spurious counterexample again (refinement progress). This can typically be achieved by pruning nodes and computing a new analysis precision (overapproximation-driven approach), or by uncovering nodes and strengthening labels (underapproximation-driven approach), both of which includes partial deconstruction of the ART.

Currently, built-in domains in Theta include predicates, explicit values, zones, and the Cartesian abstract domain that allows the sound combination of abstract domains. There are custom interpreters provided for actions of transition systems, control flow automata and timed automata. For predicates and explicit values, given an action function, there are also interpreters based on SMT solving over a generic symbolic transition system interface where the set of initial states and the transition relation are expressed in terms of FOL formulas. A default abstractor implementation is built-in that relies on the domain and the interpreter, also parameterizable with a search strategy. Besides some custom refiner implementations, for symbolic transition systems, interpolation and unsat core-based refinement strategies for predicates and explicit values are provided out-of-the-box.

3.2.3 SMT Solver Interface

The framework provides a general SMT solver interface that supports incremental solving, unsat cores, and the generation of binary and sequence interpolants. The solver interface can be used by the analysis components. Typically, the preorder over states and the transfer function are implemented in terms of queries to an SMT solver. A refiner component may use the interface to check feasibility of an abstract path and to generate interpolants or unsat cores for abstraction refinement. Currently, the interface is implemented by the SMT solver Z3 [MB08], but it can easily be extended with new solvers.

3.2.4 Extending and Instantiating the Framework

The framework can easily be extended with new formalisms and analyses. As an example, suppose that one wants to add support for the reachability checking of Petri nets [Mur89]. First, the formalism has to be implemented, which is a collection of simple classes representing places, transitions and arcs of Petri nets. A possible language frontend could be the standard PNML format for Petri nets.

In order to perform reachability checking, the analysis backend has to be extended as well. The semantics of Petri nets can be described as a symbolic transition system, for example by representing places (marked with tokens) with integer variables and transitions as FOL formulas adding/subtracting from places. Therefore, some abstract domains (such as predicates and explicit values) along with abstraction and refinement strategies (such as interpolation) work out of box if the action function is implemented. An action of a Petri net can be represented as the formula describing a Petri net transition and the action function as a function that returns all such transitions. The init and transfer functions thus work out of the box for the abstract domains mentioned before.

Instantiating an executable tool from the framework (see examples in Section 3.3) is also straightforward. A (command line or GUI) application has to be written that takes the parameters (path of the input model, domain, abstraction and refinement strategies, etc.), parses the input model using the language frontends and instantiates and runs the analysis.

3.3 Use Cases

The following section presents three use cases for tools that are built on top of the Theta framework. We point out that the measurements and a part of the implementation described in Section 3.3.1 and in Section 3.3.2 are not results of the author of this dissertation, and thus should not be considered as such. The inclusion of these results serve as an illustration for the utility of the framework.

Furthermore, we would like to refer to some other lines of research unrelated to this dissertation but related to the Theta framework[2]. Theta has been integrated as a verification backend in Gamma [Mol+18], a tool for modeling and model integration based on statecharts. This way, Theta enabled the verification of selected protocols and algorithms of an electronic railway interlocking system modeled in Gamma. For the verification of C programs, the tool Gazer-Theta has been proposed [ÁSH21]. The tool has been submitted to the 10th International Competition on Software Verification (SV-COMP 2021) [Bey21], where it competed in 9 subcategories.

[2] For a complete list of related papers, visit https://ftsrg.mit.bme.hu/theta/publications/.

3.3.1 Theta for Transition Systems

The tool Theta-sts is an instantiation of the Theta framework for reachability analysis of (symbolic) transition systems, based on an earlier, preliminary version [c6]. As input language, the tool supports the AIGER format (also used in the Hardware Model Checking Competition [Cab+16]) and an intermediate language for describing PLC models [Fer+15]. The tool relies on the built-in predicate and explicit value domains and refinement strategies based on binary interpolation, sequence interpolation and formulas from unsat cores. Some additional utilities are also implemented, for example inferring the initial precision and simplifying the input system.

Figure 3.2 (from [HM17]) shows a heatmap of the execution time of 20 analysis configurations on 12 hardware (hw) and 6 PLC models. White squares correspond to a timeout. Configurations are abbreviated with the first letter of the domain (predicate, explicit), the refinement strategy (binary interpolation, sequence interpolation, unsat cores), the initial precision (empty, property-based) and the exploration strategy (DFS, BFS). The heatmap shows that no single configuration can verify all models and the execution time is very diverse, motivating the need for a configurable framework.

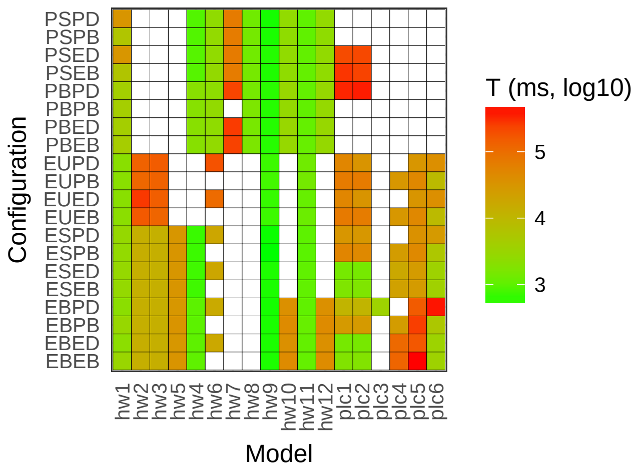


Figure 3.2: Heatmap of execution time (ms) for transition systems (logarithmic scale)

3.3.2 Theta for Control Flow Automata

The tool Theta-cfa is an instantiation of the Theta framework for the reachability analysis of control flow automata. As input language, the tool supports a subset of C, enhanced by various size reduction techniques such as compiler optimizations and program slicing methods [c15]. This tool uses the same built-in abstract domains and refinement strategies as the Theta-sts tool, only the interpreter differs.

Figure 3.3 (from [c15]) presents a heatmap of the verification time of 16 analysis configurations on 9 models from SV-COMP [Bey16], selected from those categories that are currently supported by our C frontend. Configurations are abbreviated with the first letter of the slicing method (none, backward, value, thin), the compiler optimizations (true, false) and the exploration strategy (DFS, BFS). Similarly to transition systems, different configurations are more suitable for different input models.

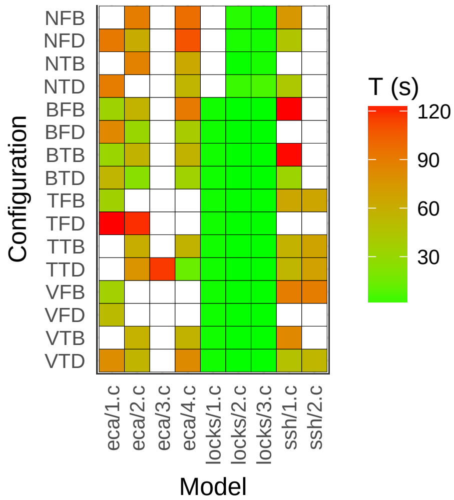


Figure 3.3: Heatmap of execution time (s) for C programs

3.3.3 THETA for Timed Automata

The tool THETA-XTA is an instantiation of the THETA framework for reachability checking of timed automata with discrete variables. As input language, the tool supports a reasonable subset of the UPPAAL 4.x XTA format[3,4]. The results of Chapter 4, of Chapter 5, and of Chapter 6 are implemented in THETA-XTA. For details, we refer the reader to the respective chapters.

3.4 Conclusions

In this chapter we introduced THETA, a configurable model checking framework for abstraction refinement-based reachability analysis for different formalisms. We described the architecture that helps to implement, evaluate and combine various algorithms in a modular way for different formalisms. We also demonstrated the applicability of the framework by use cases for the verification of hardware, PLC, software and timed automata models. Results of the evaluation with configuring and combining different analysis modules support the need for a generic framework, such as THETA. Subsequent results in Chapter 4 are built on top of our framework.

3.4.1 Thesis Summary

This concludes Thesis 1.1 of this dissertation. We summarize it as follows.

> **Thesis 1.1** *Architecture of a configurable model checking framework.* I designed the architecture, interfaces and generic algorithmic components of THETA, a generic, modular, and configurable model checking framework that enables the combination of various abstract domains, interpreters, and strategies for abstraction and refinement, applied to models of various formalisms.

[3] Not supporting procedures and composite types other than arrays of synchronization channels.
[4] See the web help on http://www.uppaal.org for a language reference.

Chapter 4

A Uniform Formalization of Abstraction Refinement Strategies for Timed Automata

We address the location reachability problem of timed automata with discrete variables. Overall, we propose a *formal algorithmic framework* that enables the uniform formalization of several abstract domains and refinement strategies for both clock and discrete variables. The main elements are a generic algorithm for lazy reachability checking and an abstract reachability tree as its central data structure. The main advantage of the framework is that, based on the notion of the direct product abstract domain, it allows the *seamless combination* of various lazy abstraction methods, resulting in many distinct algorithm configurations that together admit efficient verification of a wide range of timed automata models. This algorithmic framework allows a straightforward implementation of these strategies in our open source model checking framework Theta [*c*10], this way enabling the practical *evaluation of the proposed algorithm configurations*. The configurability of this framework also allows the integration of existing efficient lazy abstraction algorithms for clock variables based on LU-bounds [HSW13], thus admitting the combination and comparison of our methods with the state-of-the-art in Chapter 5 and in Chapter 6.

4.1 Algorithm for Lazy Reachability Checking

In this section we present our uniform approach, a lazy reachability checking algorithm that allows the combination of various abstract domains and refinement strategies. It is based on the notion of Abstract Reachability Tree, which is defined in the sequel. Then the algorithm itself is described.

4.1.1 Abstract Reachability Tree

The central data structure of the algorithm is an abstract reachability tree.

Definition 4.1 (Abstract domain). *For our purposes, an abstract domain for a timed automaton $\mathcal{A}$ is a tuple $\mathbb{D} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ such that*

- *$\mathcal{S}$ is set of abstract states,*
- *$\sqsubseteq\ \subseteq \mathcal{S} \times \mathcal{S}$ is a preorder,*

- $\mathsf{init} \in \mathcal{S}$ *is the abstract initial state,*
- $\mathsf{post} : T \times \mathcal{S} \to \mathcal{S}$ *is the abstract post-image operator, and*
- $[\![\cdot]\!] : \mathcal{S} \to \mathcal{P}(\mathcal{C})$ *is the concretization function.* ▪

For soundness, we assume the following properties to hold.

Definition 4.2 (Sound abstraction). *An abstract domain* $(\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ *is sound iff*

- $s_1 \sqsubseteq s_2 \Rightarrow [\![s_1]\!] \subseteq [\![s_2]\!]$,
- $\Sigma_0 \subseteq [\![\mathsf{init}]\!]$, *and*
- $post_t[\![s]\!] \subseteq [\![\mathsf{post}_t(s)]\!]$. ▪

The tree structure of an abstract reachability tree is given by an unwinding.

Definition 4.3 (Unwinding). *An unwinding of a timed automaton* $\mathcal{A}$ *is a tuple* $U = (N, E, n_0, M_N, M_E, \triangleright)$ *where*

- (N, E) *is a directed tree rooted at node* $n_0 \in N$,
- $M_N : N \mapsto L$ *is the node labeling,*
- $M_E : E \mapsto T$ *is the edge labeling and*
- $\triangleright \subseteq N \times N$ *is the covering relation.*

For an unwinding we require that the following properties hold:

- $M_N(n_0) = \ell_0$,
- *for each edge* $e \in E$ *with* $e = (n, n')$ *the transition* $M_E(e) = (\ell, \cdot, \cdot, \ell')$ *is such that* $M_N(n) = \ell$ *and* $M_N(n') = \ell'$,
- *for all nodes* n *and* n' *such that* $n \triangleright n'$ *it holds that* $M_N(n) = M_N(n')$. ▪

The term $n \triangleright n'$ marks that search from node n of the unwinding is to be pruned, as another node n' admits all runs that are feasible from n. We define the following shorthand notations for convenience: $\ell_n = M_N(n)$ and $t_e = M_E(e)$.

Definition 4.4 (Abstract reachability tree). *An abstract reachability tree (ART) for a timed automaton* $\mathcal{A}$ *over a sound abstract domain* $\mathbb{D}$ *is a labeled unwinding, that is, a pair* $\mathcal{G} = (U, \psi)$ *where*

- U *is an unwinding of* $\mathcal{A}$, *and*
- $\psi : N \mapsto \mathcal{S}$ *is a labeling of nodes by abstract states.* ▪

We will use the following shorthand notation: $s_n = \psi(n)$. We define the following properties for nodes.

Definition 4.5 (Properties of nodes). *A node* n *is* expanded *iff for all transitions* $t \in T$ *such that* $t = (\ell, \cdot, \cdot, \cdot)$ *and* $\ell_n = \ell$, *either* t *is disabled from* $[\![s_n]\!]$, *or* n *has a successor for* t. *A node* n *is* covered *iff* $n \triangleright n'$ *for some node* n'. *It is* excluded *iff it is covered or it has an excluded parent. A node is* complete *iff it is either expanded or excluded. A node* n *is* ℓ*-safe iff* $\ell_n \neq \ell$.

For an ART to be useful for reachability checking, we have to ensure that the tree represents an over-approximation of the set of reachable states. Therefore we introduce restrictions on the labeling, as formalized in the next definition.

Definition 4.6 (Well-labeled node). *A node* n *of an ART* $\mathcal{G}$ *for a timed automaton* $\mathcal{A}$ *is well-labeled iff the following conditions hold:*

- *(initiation) if* $n = n_0$, *then* $\Sigma_0 \subseteq [\![s_n]\!]$,

- *(consecution) if $n \neq n_0$, then for its parent m and the transition $t = t_{(m,n)}$ it holds that $post_t[\![s_m]\!] \subseteq [\![s_n]\!]$*
- *(coverage) if $n \rhd n'$ for some node n', then $[\![s_n]\!] \subseteq [\![s_{n'}]\!]$ and n' is not excluded.* ▪

Besides preserving reachable states, we will also ensure that nodes represent runs of the automaton. We formalize this in the following definitions.

Definition 4.7 (Feasible node and transition). *Let n be a node of an ART $\mathcal{G}$, and π the path from n_0 to n in $\mathcal{G}$. Then n is feasible iff π is feasible. Moreover, a transition t is feasible from n iff the path $\pi \cdot t$ is feasible.* ▪

The above definitions for nodes can be extended to trees.

Definition 4.8 (Properties of ARTs). *An ART is complete, ℓ-safe, well-labeled or feasible iff all its nodes are complete, ℓ-safe, well-labeled, or feasible, respectively.* ▪

A well-labeled ART preserves reachable states, which is expressed by the following proposition.

Proposition 3. *Let $\mathcal{G}$ be a complete, well-labeled ART for a timed automaton $\mathcal{A}$. If $\mathcal{A}$ has a symbolic run $(\ell_0, \Sigma_0) \xrightarrow{t_1} (\ell_1, \Sigma_1) \xrightarrow{t_2} \ldots \xrightarrow{t_k} (\ell_k, \Sigma_k)$ then $\mathcal{G}$ has a non-excluded node n such that $\ell_k = \ell_n$ and $\Sigma_k \subseteq [\![s_n]\!]$.*

Proof. We prove the statement by induction on the length k of the symbolic run. If $k = 0$, then $\ell_0 = \ell_{n_0}$ and $\Sigma_0 \subseteq [\![s_{n_0}]\!]$ by condition *initiation*, thus n_0 is a suitable witness. Suppose the statement holds for runs of length at most $k-1$. Hence there exists a non-excluded node m such that $\ell_{k-1} = \ell_m$ and $\Sigma_{k-1} \subseteq [\![s_m]\!]$.

Clearly transition t_k is not disabled from $[\![s_m]\!]$, as then by the induction hypothesis it would also be disabled from Σ_{k-1}, which contradicts our assumption. As m is complete and not excluded, it is expanded, and thus has a successor n for transition t_k with $\ell_n = \ell_k$. By condition *consecution*, we have $post_{t_k}[\![s_m]\!] \subseteq [\![s_n]\!]$. As $\Sigma_{k-1} \subseteq [\![s_m]\!]$, by the monotonicity of images in $\subseteq$, we obtain $\Sigma_k \subseteq [\![s_n]\!]$.

Thus if n is not covered, then it is a suitable witness for the statement. Otherwise there exists a node n' such that $n \rhd n'$. By condition *coverage*, we know that $[\![s_n]\!] \subseteq [\![s_{n'}]\!]$ and n' is not excluded, thus n' is a suitable witness. □

4.1.2 Reachability Algorithm

The pseudocode of the algorithm is shown in Algorithm 1. The algorithm gets as input a timed automaton $\mathcal{A}$ and a distinguished error location $\ell_e \in L$. The goal of the algorithm is to decide whether ℓ_e is reachable for $\mathcal{A}$. To this end the algorithm gradually constructs an ART for $\mathcal{A}$ and continually maintains its well-labeledness and feasibility. Upon termination, it either witnesses reachability of ℓ_e by a feasible node n such that $\ell_n = \ell_e$, which by Definition 4.7 corresponds to a symbolic run of $\mathcal{A}$ to ℓ_e, or produces a complete, well-labeled, ℓ_e-safe ART that proves unreachability of ℓ_e by Proposition 3.

The main data structures of the algorithm are the ART $\mathcal{G}$ and sets $passed$ and $waiting$. Set $passed$ is used to store nodes that are expanded, and $waiting$ stores nodes that are incomplete. The algorithm consists of two subprocedures, CLOSE and EXPAND. Procedure CLOSE attempts to cover a node n by some other node. It calls a procedure COVER that tries to force cover the node by adjusting its label so that it is subsumed by the label of some candidate node n'. Procedure EXPAND expands a node n by creating its successors. To avoid creating infeasible nodes, it calls a procedure DISABLE that checks feasibility of a given transition t, and adjusts the labeling of n so that if t is infeasible from n, then

Algorithm 1 Reachability algorithm

ensure $\rho = \textsc{safe}$ iff ℓ_e is unreachable for $\mathcal{A}$
function $\textsc{explore}(\mathcal{A}, \ell_e)$ **returns** $\rho \in \{\textsc{safe}, \textsc{unsafe}\}$
  let n_0 be a node with $\ell_{n_0} = \ell_0$ and $s_{n_0} = \mathsf{init}$
  $N \leftarrow \{n_0\}, E \leftarrow \emptyset, \triangleright \leftarrow \emptyset$
  let $\mathcal{G}$ be an ART for $\mathcal{A}$ over N, E and $\triangleright$

  $\mathit{passed} \leftarrow \emptyset$, $\mathit{waiting} \leftarrow \{n_0\}$
  invariant $\mathcal{G}$ is well-labeled and feasible
  while $n \in \mathit{waiting}$ for some n **do**
    $\mathit{waiting} \leftarrow \mathit{waiting} \setminus \{n\}$
    if $\ell_n = \ell_e$ **then**
      return $\textsc{unsafe}$
    else
      $\textsc{close}(n)$
      if n is not covered **then**
        $\textsc{expand}(n)$
  return $\textsc{safe}$

invariant $\mathcal{G}$ is well-labeled and feasible
procedure $\textsc{close}(n)$
  for all $n' \in \mathit{passed}$ such that $\ell_n = \ell_{n'}$ **do**
    $\textsc{cover}(n, n')$
    if $s_n \sqsubseteq s_{n'}$ **then**
      $\triangleright \leftarrow \triangleright \cup \{(n, n')\}$
      return

invariant $\mathcal{G}$ is well-labeled and feasible
ensure n is expanded
procedure $\textsc{expand}(n)$
  for all $t \in T$ such that $t = (\ell, \cdot, \cdot, \ell')$ with $\ell = \ell_n$ **do**
    if not $\textsc{disable}(n, t)$ **then**
      let $s' = \mathsf{post}_t(s_n)$
      let n' be a new node with $\ell_{n'} = \ell'$ and $s_{n'} = s'$
      let $e = (n, n')$ be a new edge with $t_e = t$
      $N \leftarrow N \cup \{n'\}$
      $E \leftarrow E \cup \{e\}$
      $\mathit{waiting} \leftarrow \mathit{waiting} \cup \{n'\}$
  $\mathit{passed} \leftarrow \mathit{passed} \cup \{n\}$

invariant $\mathcal{G}$ is well-labeled and feasible
procedure $\textsc{cover}(n, n')$

invariant $\mathcal{G}$ is well-labeled and feasible
ensure β iff t is disabled from $[\![s_n]\!]$
ensure $\neg\beta$ iff t is feasible from n
function $\textsc{disable}(n, t)$ **returns** β

it also becomes disabled from $[\![s_n]\!]$. Both CLOSE and EXPAND potentially modify the labeling of some nodes as a side effect, but in a way that maintains well-labeledness and feasibility of the ART. Naturally, the implementation of procedures COVER and DISABLE depends on the abstract domain, and are described in Section 4.2 in detail.

The algorithm consists of a single loop in line 9 that employs the following strategy. The loop consumes nodes from *waiting* one by one. If *waiting* becomes empty, then $\mathcal{A}$ is deemed safe. Otherwise, a node n is removed from *waiting*. If the node represents the error location, then $\mathcal{A}$ is deemed unsafe. Otherwise, in order to avoid unnecessary expansion of the node, the algorithm tries to cover it by a call to CLOSE. If there are no suitable candidates for coverage, then the algorithm establishes completeness of the node by expanding it using EXPAND, which puts it in *passed*, and puts all its successors in *waiting*.

We show that EXPLORE is correct with respect to the procedure contracts listed in Algorithm 1. We focus on partial correctness, as termination depends on the particular abstract domain and refinement method used. We note that in general, termination can be easily ensured using the right extrapolation operator for clock variables [HSW13; WJ15][*c*9].

Proposition 4. *Procedure* EXPLORE *is partially correct: if* EXPLORE$(\mathcal{A}, \ell_e)$ *terminates, then the result is* SAFE *iff* ℓ_e *is unreachable for* $\mathcal{A}$.

Sketch. Let $covered = \{n \in N \mid n \text{ is covered}\}$. It is easy to verify that the algorithm maintains the following invariants:

- $N = passed \cup waiting \cup covered$,
- *passed* is a set of non-excluded, expanded, ℓ_e-safe nodes,
- *waiting* is a set of non-excluded, non-expanded nodes,
- *covered* is a set of covered, non-expanded, ℓ_e-safe nodes.

It is easy to see that under the above assumptions sets *passed*, *waiting* and *covered* form a partition of N. Assuming that $\mathcal{G}$ is well-labeled and feasible, partial correctness of the algorithm is then a direct consequence: At line 12 a node is encountered that is not ℓ_e-safe, thus by Definition 4.7 there is a symbolic run of $\mathcal{A}$ to ℓ_e; conversely, at line 17 the set *waiting* is empty, so $\mathcal{G}$ is complete and ℓ_e-safe, and as a consequence of Proposition 3 the location ℓ_e is indeed unreachable for $\mathcal{A}$.

What remains to show is that the algorithm maintains well-labeledness and feasibility of $\mathcal{G}$. We assume that procedures COVER and DISABLE maintain well-labeledness and feasibility, which we prove to hold in Section 4.2.

Initially, node n_0 is well-labeled, as $\Sigma_0 \subseteq [\![\mathsf{init}]\!] = [\![s_{n_0}]\!]$, thus n_0 satisfies *initiation*. It also trivially satisfies feasibility, as $post_\epsilon(\Sigma_0) = \Sigma_0 \neq \emptyset$. Procedure CLOSE trivially maintains well-labeledness and feasibility, as it just possibly adds a covering edge for two nodes such that condition *coverage* is not violated. In procedure EXPAND, if DISABLE(n, t) for a transition t, then t is not feasible from n, and the labeling is adjusted so that t is disabled from $[\![s_n]\!]$. Otherwise, t is feasible from n, and a successor node n' is created. Clearly, n' is feasible as t is feasible. Moreover, $post_t[\![s_n]\!] \subseteq [\![\mathsf{post}_t(s_n)]\!] = [\![s_{n'}]\!]$, thus n' satisfies *consecution*. Thus according to the contract, n becomes expanded, and all its successors are well-labeled and feasible, so well-labeledness and feasibility of $\mathcal{G}$ is preserved. ☐

4.2 Abstraction Refinement

Algorithm 1 is abstracted over the particular abstract domain used to well-label the constructed ART. Moreover, it declares two procedures, COVER and DISABLE, that perform forced covering and abstraction refinement over the abstract domain, respectively. In Chapter 5 and Chapter 6, we describe several possible abstract domains, and corresponding abstraction refinement strategies, that can be used for model checking timed automata with discrete variables.

In the listings of the given refinement strategies, we are going to refer to a simple procedure UPDATE that enables safely updating the labeling for a given node in the ART.

Algorithm 2 Safely updating the abstraction

invariant $\mathcal{G}$ is well-labeled and feasible
require n root $\Rightarrow \Sigma_0 \subseteq [\![s]\!]$
require $(m, n) \in E$ with $t = t_{(m,n)}$ for some $m \Rightarrow post_t[\![s_m]\!] \subseteq [\![s]\!]$
ensure $s_n = s$
procedure UPDATE(n, s)
  for all m such that $m \triangleright n$ and $s_m \not\sqsubseteq s$ **do**
    $\triangleright \leftarrow \triangleright \setminus (m, n)$
    $waiting \leftarrow waiting \cup \{m\}$
  $s_n \leftarrow s$

Proposition 5. *UPDATE is totally correct: If either n is the root and $\Sigma_0 \subseteq [\![s]\!]$, or there exists an edge $e = (m, n)$ with $t_e = t$ for some m and $post_t[\![s_m]\!] \subseteq [\![s]\!]$, then UPDATE$(n, s)$ terminates and ensures $s_n = s$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. At the end of the procedure, $s_n = s$ is ensured. Clearly, n is well-labeled: *initiation* and *consecution* is ensured by contract, and *coverage* is ensured by the loop due to soundness of the abstract domain. □

4.3 Combination of Abstractions

Our approach is based on the direct product of abstract domains [CC79], as described below.

> **Definition 4.9 (Direct product domain).** *Let $\mathbb{D}_i = (\mathcal{S}_i, \sqsubseteq_i, \mathsf{init}_i, \mathsf{post}^i, [\![\cdot]\!]_i)$ for $i \in \{1, 2\}$. Then their direct product is the abstract domain $\mathbb{D}_1 \times \mathbb{D}_2 = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ where*
> - $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$,
> - $(s_1, s_2) \sqsubseteq (s_1', s_2')$ *iff* $s_1 \sqsubseteq_1 s_1'$ *and* $s_2 \sqsubseteq_2 s_2'$ *(thus $\sqsubseteq$ is a preorder),*
> - $\mathsf{init} = (\mathsf{init}_1, \mathsf{init}_2)$,
> - $\mathsf{post}_t(s_1, s_2) = (\mathsf{post}_t^1(s_1), \mathsf{post}_t^2(s_2))$, *and*
> - $[\![(s_1, s_2)]\!] = [\![s_1]\!]_1 \cap [\![s_2]\!]_2$. ▪

In later descriptions, when it is clear from the context, we are going to omit indexes when referring to components of a direct product (and write e.g. $(\mathsf{post}_t(s_1), \mathsf{post}_t(s_2))$ instead of $(\mathsf{post}_t^1(s_1), \mathsf{post}_t^2(s_2))$).

Proposition 6. *If $\mathbb{D}_1$ and $\mathbb{D}_2$ are sound, then $\mathbb{D}_1 \times \mathbb{D}_2$ is sound.*

In case of timed automata with discrete variables according to Definition 2.15, abstraction and refinement can be conveniently defined compositionally, where clock variables and discrete variables are handled by separate abstractions. Algorithm 3 describes a straightforward method for achieving this separation.

Algorithm 3 Combination of abstractions

1: **procedure** $\text{COVER}_\times(n, n')$
2: $\quad \text{COVER}_D(n, n')$
3: $\quad \text{COVER}_C(n, n')$

4: **invariant** $\mathcal{G}$ is well-labeled and feasible
5: **procedure** $\text{COVER}_D(n, n')$

6: **invariant** $\mathcal{G}$ is well-labeled and feasible
7: **procedure** $\text{COVER}_C(n, n')$

8: **function** $\text{DISABLE}_\times(n, t)$ **returns** β
9: $\quad$ **return** $\text{DISABLE}_D(n, t)$ or $\text{DISABLE}_C(n, t)$

10: **invariant** $\mathcal{G}$ is well-labeled and feasible
11: **define** $(s_1, s_2) = s_n$
12: **ensure** β iff t is disabled from $[\![s_1]\!]$
13: **ensure** $\neg\beta$ iff t is data-feasible from n
14: **function** $\text{DISABLE}_D(n, t)$ **returns** β

15: **invariant** $\mathcal{G}$ is well-labeled and feasible
16: **define** $(s_1, s_2) = s_n$
17: **ensure** β iff t is disabled from $[\![s_2]\!]$
18: **ensure** $\neg\beta$ iff t is clock-feasible from n
19: **function** $\text{DISABLE}_C(n, t)$ **returns** β

In the above description, in line 10 and line 15, we refer to the preservation of well labeledness for the two projections of the ART. This weaker assumption will simplify proofs of correctness for the component refiners. We show that this implies well-labeledness in the original sense.

Total correctness of $\text{COVER}_\times$ follows from total correctness of COVER_D and COVER_C. We show total correctness of $\text{DISABLE}_\times$ as follows.

Proposition 7. *$\text{DISABLE}_\times$ is totally correct: $\text{DISABLE}_\times(n, t)$ terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* false *iff t is feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes.

First we show that $\text{DISABLE}_\times$ maintains well-labeledness. By contract, DISABLE_C and DISABLE_D preserve well-labeledness of $\mathcal{G}$ (in the weaker sense described above). Let $s_n = (s_1, s_2)$ for root node n. As $\Sigma_0 \subseteq [\![s_1]\!]$ and $\Sigma_0 \subseteq [\![s_2]\!]$, clearly $\Sigma_0 \subseteq [\![s_1]\!] \cap [\![s_2]\!] = [\![(s_1, s_2)]\!]$, thus *initiation* is preserved. Now let $s_m = (s_1, s_2)$ and $s_n = (s'_1, s'_2)$ for nodes m and n such that $(m, n) \in E$ and $t = t_{(m,n)}$. As $post_t$ is an image and $post_t[\![s_1]\!] \subseteq [\![s'_1]\!]$ and $post_t[\![s_2]\!] \subseteq [\![s'_2]\!]$, we have $post_t[\![(s_1, s_2)]\!] = post_t([\![s_1]\!] \cap [\![s_2]\!]) \subseteq post_t[\![s_1]\!] \cap post_t[\![s_2]\!] \subseteq [\![s'_1]\!] \cap [\![s'_2]\!] = [\![(s'_1, s'_2)]\!]$, thus *consecution* is preserved. Finally, let $s_m = (s_1, s_2)$ and $s_n = (s'_1, s'_2)$ for nodes m and n such that $m \rhd n$. As $[\![s_1]\!] \subseteq [\![s'_1]\!]$ and $[\![s_2]\!] \subseteq [\![s'_2]\!]$, clearly $[\![(s_1, s_2)]\!] = [\![s_1]\!] \cap [\![s_2]\!] \subseteq [\![s'_1]\!] \cap [\![s'_2]\!] = [\![(s'_1, s'_2)]\!]$, thus *coverage* is preserved.

Assume that t is feasible from n. Then t is both data- and clock-feasible from n by Remark 4. Thus $\text{DISABLE}_D(n, t) =$ false and $\text{DISABLE}_C(n, t) =$ false by contract, from which $\text{DISABLE}_\times(n, t) =$ false follows directly. Assume that t is not feasible from n. Then t is either not data- or not clock-feasible from n by Remark 4. Assume t is not data-feasible from n. Thus $\text{DISABLE}_D(n, t) =$ true and t becomes disabled from $[\![s_1]\!]$ by contract. As a consequence, $\text{DISABLE}_\times(n, t) =$ true, and t becomes disabled from $[\![s_n]\!] = [\![(s_1, s_2)]\!] = [\![s_1]\!] \cap [\![s_2]\!]$. The other case follows symmetrically. □

To simplify exposition, we are going to treat the labeling of nodes by abstract states ψ as a lens (in simple terms, a pair consisting of a "getter" and a "setter") that can be used to deeply manipulate the structure of a given label. Thus later in the text, when we refer to s_n, we are going to mean the corresponding component of a direct product based on the context.

4.4 Implementation

The notions described in this chapter have been implemented in the Theta framework. In order to make exposition for this and upcoming chapters simpler we made some simplifications exploiting the fact that the formalism in our case is fixed to timed automata. Although the mapping between concepts introduced in this and the previous chapter is mostly straightforward, for clarity we summarize our implicit assumptions in Table 4.1.

Table 4.1: Implementation in the Theta framework

Theta concept	**Implementation**
Abstract domain	The set of abstract states $\mathcal{S}$ and the preorder $\sqsubseteq$. In all our Theta domains for timed automata, the top-level abstract domain is the *location domain* that tracks the current location, and wraps all other domains over clock and discrete variables. In our exposition this domain is implicitly encoded in the notion of unwinding.
Precision	Our approach is underapproximation-driven, and thus does not rely on explicit representation of precision. The precision is thus the "unit precision", i.e. a single element set.
Action	A transition $t \in T$.
Init function	For the location domain it is $p \mapsto \{\ell_0\}$. For the domains described here, it is $p \mapsto \{\mathsf{init}\}$.
Transfer function	For the location domain it is the obvious function implied by T. For the domains described here, it is $p \mapsto t \mapsto s \mapsto \mathsf{post}_t(s)$.
Action function	Defined over the location domain; it is $(\ell, \cdot) \mapsto \{t \in T \mid t = (\ell, \cdot, \cdot, \cdot)\}$.
Abstract reachability tree	The ART, with the caveat that the notion of unwinding as described here implicitly encodes the location domain.
Abstractor	Procedure EXPLORE.
Refiner	Procedures COVER and DISABLE.

4.5 Conclusions

In this chapter, we presented *an algorithmic framework* for the lazy abstraction based location reachability checking of timed automata with discrete variables. We formalized the combination of abstractions and proved its properties. This framework allowed the straightforward implementation of efficient model checkers using configurable combined strategies, as described in Chapter 5 and Chapter 6. The different abstraction refinement strategies discussed in those chapters is summarized in Table 4.2.

Table 4.2: Summary of refinement strategies

	Lazy Zone Interpolation Forward	Lazy Zone Interpolation Backward	Lazy Valuation Interpolation Forward	Lazy Valuation Interpolation Backward
$\mathbb{D}$	$\mathbb{D}_{\mathcal{ZI}}$		$\mathbb{D}_{\mathcal{EI}}$	
COVER	$\text{COVER}_{\mathcal{ZI}}$		$\text{COVER}_{\mathcal{EI}}$	
DISABLE	$\text{DISABLE}_{\mathcal{ZI}}$		$\text{DISABLE}_{\mathcal{EI}}$	
Propagation	BLOCK_{FW}	BLOCK_{BW}	$\text{REFINE}_{\text{FW}}$	$\text{REFINE}_{\text{BW}}$
Interpolation	$\text{INTERPOLATE}_{\mathcal{Z}}$		$\text{INTERPOLATE}_{\mathcal{E}}$	

Future Work. According to the algorithm described in this chapter, refinement is triggered upon encountering a disabled transition. An interesting direction would be to experiment with counterexample-guided refinement for both the abstraction of discrete and continuous variables. Moreover, there are several possibilities for fine-tuning the proposed algorithm. For example, the algorithm as described applies an aggressive covering strategy, as it tries all possible nodes for coverage before expanding a node. The investigation of more sophisticated covering strategies (e.g. forced covering as in [McM06]) might yield better scaling with respect to execution time. Additionally, by memoizing abstract states, the memory footprint of the algorithm may be significantly reduced.

4.5.1 Thesis Summary

This concludes Thesis 1.2 of this dissertation. We summarize it as follows.

> **Thesis 1.2** *A uniform formalization of abstraction refinement strategies for timed automata.* I proposed and proved correct a formal algorithmic framework that enables the uniform formalization and combined use of various abstract domains and abstraction refinement strategies for the location reachability checking of timed automata.

Chapter 5

Lazy Reachability Checking for Timed Automata using Interpolants

The reachability problem of timed automata [AD94] deals with the question whether a given error location is reachable from an initial state along the transitions of the automaton. The standard solution of this problem involves performing a forward exploration in the so-called zone-graph induced by the automaton [DT98]. There, each abstract state is a zone, a special set of concrete states that can be represented as the solution set for a set of clock constraints.

To ensure performance and termination, model checkers for timed automata usually apply some sort of generalization of zones based on maximal lower- and upper bounds [Beh+04] (LU-bounds) appearing in the guards of the automaton. This can be performed directly by extrapolation [Beh+04] parameterized by bounds obtained by static analysis [Beh+03]. Alternatively, bounds can be propagated lazily for all transitions [Her+11] or along an infeasible path [HSW13], which, combined with an efficient method for inclusion checking [HSW12] with respect to a non-convex abstraction induced by the bounds, results in an efficient method for reachability checking of timed automata. This latter approach is a form of lazy abstraction, a variant of counterexample-guided abstraction refinement [Cla+03] (CEGAR), where – instead of eagerly computing abstractions using an abstract post-image operator, a typically expensive operation – abstraction is computed on-the-fly and locally in the state space along a single execution path where more precision is necessary.

In this chapter, we propose a similar lazy algorithm for reachability checking of timed automata. However, instead of propagating the bounds appearing in guards, the algorithm considers the guards themselves: if the abstraction is too coarse to exclude an infeasible path, a zone representing the guards of a disabled transition is propagated backwards using pre-image computation. Based on the pre-image, we compute a zone strong enough to block the disabled transition in form of an interpolant [McM03]. In a similar fashion, we use interpolation to effectively prune the search space by enforcing coverage of a newly discovered state with an already visited state when possible. We propose two refinement strategies in this framework. Both methods are a combination of forward search, backward search and zone interpolation, and can be considered as a generalization of zone interpolation to sequences of transitions of a timed automaton.

We compared the proposed interpolation based method and the non-convex LU-abstraction based method [HSW13] on the usual benchmark models for timed automata. Results show that our method performs similarly to the highly sophisticated algorithm of [HSW13], and in cases can even generate a smaller state space. Moreover, it turned out that for some models the proposed refinement strategies are less sensitive to search order, thus are more robust against bad decisions during search.

5.1 Related Work

Lazy abstraction [Hen+02] is an approach widely used for model checking, and in particular for model checking software. It consists of building an abstract reachability graph on-the fly, representing an abstraction of the system, and refining a part of the tree in case a spurious counterexample is found. Lazy abstraction with interpolants [McM06] (also known as Impact) and lazy annotation [McM10] are both lazy abstraction techniques for software where refinement is performed using interpolant generation.

For timed automata, a lazy abstraction approach based on non-convex LU-abstraction [Beh+04] and on-the-fly propagation of bounds has been proposed [HSW13]. A significant difference of this algorithm compared to usual lazy abstraction algorithms is that it builds an abstract reachability graph that preserves exact reachability information (a so-called adaptive simulation graph). As a consequence it is able to apply refinement as soon as the abstraction admits a transition disabled in the concrete system. In our work, we apply the same approach, but for a different abstract domain, with different refinement strategies.

The work closest to ours is difference bound constraint abstraction [WJ15]. The refinement method presented there and our refinement strategy we refer to as the binary (BWITP) strategy are highly analogous, and both are very similar to lazy annotation. However, our refinement strategy that we refer to as the sequence (FWITP) strategy is different in concept. Moreover, in [WJ15], abstractions are sets of difference constraints, and refinement rules are defined on a case-by-case basis for guards, resets and delay. In our work, we represent abstractions as canonical difference bound matrices, and define abstraction refinement in more general terms, as a combination of symbolic forward and backward search and zone interpolation. This formulation enables a simple generalization of our approach to automata with diagonal constraints in guards [BLR05] and to updatable timed automata [Bou04], as well as to the application of backward exploration. Moreover, by representing abstractions as canonical difference bound matrices, known zone-based abstraction methods can be considered orthogonal to our approach.

A more recent result related to our work appeared in [RSM19]. There, abstraction refinement is performed using zone interpolation as well, but interpolants are computed to be minimal at a cost $\mathcal{O}(|C|^4)$, instead of the $\mathcal{O}(|C|^3)$ cost of non-minimal interpolants.

5.2 Zones and DBMs

A *zone* $Z \in \mathcal{Z}$ is the set of solutions of a clock constraint $\varphi \in Constr_C$, that is $Z = \{\eta \in \mathcal{V}(C) \mid \eta \models \varphi\}$. If Z and Z' are zones and $t \in T$, then $\emptyset$, and $\mathcal{V}(C)$, and Z_0, and $Z \cap Z'$, and $post_t^C(Z)$ and $pre_t^C(Z')$ are also zones. In the context of zones, we will denote $\emptyset$ by $\bot$ and $\mathcal{V}(C)$ by $\top$. Zones are not closed under complementation, but the complement of any zone is the union of finitely many zones. For a zone Z, we are going to denote a minimal set of such zones by $\neg Z$.

Zones can be efficiently represented by difference bound matrices [Dil90]. A *bound* is either ∞, or a finite bound of the form $(m, \prec)$ where $m \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. Difference bounds can be totally ordered by "strength", that is, $(m, \prec) < \infty$ and $(m_1, \prec_1) < (m_2, \prec_2)$ for $m_1 < m_2$ and $(m, <) < (m, \leq)$. Moreover the sum of two bounds is defined as $b + \infty = \infty$ and $(m_1, \leq) + (m_2, \leq) = (m_1 + m_2, \leq)$ and $(m_1, <) + (m_2, \prec) = (m_1 + m_2, <)$.

A *difference bound matrix* (DBM) over $X = \{x_0, x_1, \ldots, x_n\}$ is a square matrix M of bounds of order $n + 1$ where an element $M_{ij} = (m, \prec)$ represents the clock constraint $x_i - x_j \prec m$. We denote by $[\![M]\!]$ the zone induced by the conjunction of constraints stored in M. We say that M is *consistent* iff $[\![M]\!] \neq \bot$. The following is a simple sufficient and necessary condition for a DBM to be inconsistent.

Proposition 8. *A DBM M is inconsistent iff there exists a negative cycle in M, that is, a set of pairs of indexes $\{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\}$ such that $M_{i_1,i_2} + \ldots + M_{i_{k-1},i_k} + M_{i_k,i_1} < (0, \leq)$ [Dil90].*

For a consistent DBM M, we say it is *canonical* iff constraints in it cannot be strengthened without losing solutions, formally, iff $M_{i,i} = (0, \leq)$ for all $0 \leq i \leq n$ and $M_{i,j} \leq M_{i,k} + M_{k,j}$ for all $0 \leq i, j, k \leq n$. For convenience, we will also consider the inconsistent DBM M with the single finite bound $M_{0,0} = (0, <)$ canonical. Up to the ordering of clocks, the canonical form is unique.

The zone operations described above, as well as set inclusion $\subseteq$ over zones, can be efficiently implemented in terms of canonical DBMs [BY04]. Therefore, we will refer to a canonical DBM M (syntax) and the zone $[\![M]\!]$ it represents (semantics) interchangeably throughout the dissertation.

Moreover, for two DBMs M_1 and M_2, we will denote by $\min(M_1, M_2)$ the (not necessarily canonical) DBM M where $M_{i,j} = \min(M_{1,ij}, M_{2,ij})$. It can be easily shown that $[\![\min(M_1, M_2)]\!] = [\![M_1]\!] \cap [\![M_2]\!]$, as well as set inclusion $\subseteq$ over zones, can be efficiently implemented in terms of canonical DBMs [BY04]. Therefore, we will refer to a canonical DBM M (syntax) and the zone $[\![M]\!]$ it represents (semantics) interchangeably throughout the dissertation.

5.3 Abstraction for Clock Variables

First, we address abstraction refinement over clock variables.

5.3.1 Zone Abstraction

Most model checkers for timed automata rely on zones for abstracting clock valuations. We define zone abstraction in our framework as follows.

> **Definition 5.1 (Zone abstraction).** *We define zone abstraction as the abstract domain $\mathbb{D}_{\mathcal{Z}} = (\mathcal{Z}, \subseteq, Z_0, post^C, (\!|\cdot|\!))$.* ▪

Note that in the absence of discrete variables, Definition 5.1 corresponds to the usual definition of zone abstraction.

Proposition 9. *$\mathbb{D}_{\mathcal{Z}}$ is sound.*

We define $\text{COVER}_{\mathcal{Z}}$ as a no-op, thus its total correctness is trivial. Moreover, we define $\text{DISABLE}_{\mathcal{Z}}$ as $\text{DISABLE}_{\mathcal{Z}}(n, t)$ iff $\mathsf{post}_t(Z) \sqsubseteq \bot$ for $Z = s_n$.

Proposition 10. *$\text{DISABLE}_{\mathcal{Z}}$ is totally correct: $\text{DISABLE}_{\mathcal{Z}}(n, t)$ terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* false *iff t is clock-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof. Termination of the procedure is trivial. Well-labeledness and feasibility follow from the fact that the procedure has no side effects. Let π be the path induced by n. Notice that $Z = post^C_\pi(Z_0)$. Assume $post^C_t(Z) \neq \bot$. Then by definition, t is clock-feasible from n, and the procedure returns false. Now assume $post^C_t(Z) = \bot$. Then by definition, t is not clock-feasible from n. But t is also disabled from $(\!|Z|\!)$, and the procedure returns true. □

5.3.2 Lazy Zone Abstraction

To obtain a coarser abstraction, we extend zone abstraction with interpolation as follows.

> **Definition 5.2 (Lazy zone abstraction).** *Let* $\mathbb{D}_{\mathcal{ZI}} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ *be the abstract domain over* $\mathbb{D}_{\mathcal{Z}}$ *with*
> - $\mathcal{S} = \mathcal{Z} \times \mathcal{Z}$,
> - $(Z, W) \sqsubseteq (Z', W')$ *iff* $W \sqsubseteq W'$,
> - $\mathsf{init} = (\mathsf{init}, \top)$,
> - $\mathsf{post}_t(Z, W) = (\mathsf{post}_t(Z), \top)$, *and*
> - $[\![(Z, W)]\!] = [\![W]\!]$. ▪

Proposition 11. $\mathbb{D}_{\mathcal{ZI}}$ *is sound.*

Given an abstract state (Z, W), the purpose of Z is to encode an exact set of reachable valuations, whereas the purpose of W is to represent a safe overapproximation of Z. This potentially enables better coverage between nodes, thus faster convergence, compared to the purely zone-based setting. In order to efficiently maintain this relationship however, we have to define procedure COVER$_{\mathcal{ZI}}$ and DISABLE$_{\mathcal{ZI}}$ accordingly. To maintain well-labeledness, these procedures rely on a procedure BLOCK that performs abstraction refinement by safely adjusting labels of nodes.

Algorithm 4 Lazy zone abstraction

procedure COVER$_{\mathcal{ZI}}$(n, n')
  let $(Z, \cdot) = s_n$
  let $(\cdot, W') = s_{n'}$
  if $Z \subseteq W'$ **then**
    for all $B \in \neg W'$ **do**
      BLOCK(n, B)

function DISABLE$_{\mathcal{ZI}}$(n, t)
  let $(Z, W) = s_n$
  let $Z' = post_t^C(Z)$
  if $Z' = \bot$ **then**
    BLOCK$(n, pre_t^C(\top))$
    return true
  else
    return false

invariant $\mathcal{G}$ is well-labeled and feasible
define $(Z, W) = s_n$
require $Z \cap B \subseteq \bot$
ensure $W \cap B \subseteq \bot$
procedure BLOCK(n, B)

In COVER$_{\mathcal{ZI}}$, as $Z \subseteq W'$ and $B \cap W' \subseteq \bot$, clearly $Z \cap B \subseteq \bot$, thus calling BLOCK(n, B) is safe. Other than that, total correctness of COVER$_{\mathcal{ZI}}$ follows trivially from total correctness of BLOCK (see later). To show the correctness of DISABLE$_{\mathcal{ZI}}$, we state the following simple lemma that establishes a connection between pre^C and $post^C$.

Lemma 4. $Z \cap pre_t^C(Z') \subseteq \bot \Leftrightarrow post_t^C(Z) \cap Z' \subseteq \bot$

Proposition 12. *DISABLE*$_{\mathcal{ZI}}$ *is totally correct: DISABLE*$_{\mathcal{ZI}}(n, t)$ *terminates and preserves well-labeledness and feasibility of* $\mathcal{G}$*; moreover, it returns* false *iff* t *is clock-feasible from* n*, and ensures that* t *is disabled from* $[\![s_n]\!]$ *otherwise.*

Proof. Termination of the procedure is trivial. Well-labeledness and feasibility follow from the total correctness of BLOCK. Let π be the path induced by n. Notice that $Z = post_\pi^C(Z_0)$. Assume $post_t^C(Z) \neq \bot$. Then by definition, t is clock-feasible from n, and the procedure returns false. Now assume $post_t^C(Z) = \bot$. Then by definition, t is not clock-feasible from n. By Lemma 4, we get $Z \cap pre_t^C(\top) \subseteq \bot$. Thus BLOCK$(n, pre_t^C(\top))$ can be called, and as a result, $W \cap pre_t^C(\top) \subseteq \bot$. By Lemma 4, we get $post_t^C(W) = \bot$. Thus t becomes disabled from $(\!|W|\!)$, and the procedure returns true. □

5.3.3 Interpolation for Zones

The proposed refinement strategies for zone abstraction, and in particular, the different implementations of BLOCK are based on interpolation, defined over zones expressed in terms of canonical DBMs.

> **Definition 5.3 (Zone interpolant).** *Given zones A and B such that $A \cap B \subseteq \bot$, a zone interpolant is a zone I such that $A \subseteq I$ and $I \cap B \subseteq \bot$ and I is defined over the clocks that appear in both A and B.* ▪

This definition of a zone interpolant is analogous to the definition of an interpolant in the usual sense [McM03]. As zones correspond to formulas in $\mathcal{DL}(\mathbb{Q})$, a theory that admits interpolation [CGS08], an interpolant always exists for a pair of disjoint zones. Algorithm 5 is a direct adaptation of the graph-based algorithm of [CGS08] for DBMs. For simplicity, we assume that A and B are defined over the same set of clocks with the same ordering, and are both canonical (naturally, these restrictions can be lifted).

Algorithm 5 Interpolation for canonical DBMs

require $A \cap B \subseteq \bot$
ensure I is a zone interpolant for A and B
function INTERPOLATE$_\mathcal{Z}(A, B)$ **returns** I
  if $A \subseteq \bot$ **then**
    return $\bot$
  else if $B \subseteq \bot$ **then**
    return $\top$
  else
    let $M = \min(A, B)$
    let $C = \{(i_1, i_2), \dots, (i_{k-1}, i_k), (i_k, i_1)\}$ be a negative cycle in M
    let $C_A = \{(i, j) \in C \mid A_{i,j} = M_{i,j}\}$
    let $I_{i,j} = \begin{cases} (0, \leq) & \text{if } i = j \\ A_{i,j} & \text{if } (i,j) \in C_A \\ \infty & \text{otherwise} \end{cases}$
    let $I = [I_{i,j}]$
    return I

After checking the trivial cases, the algorithm searches for a negative cycle in $\min(A, B)$ to witness its inconsistency. This can be done e.g. by running a variant of the Floyd-Warshall algorithm.The interpolant I is then the DBM induced by the constraints in the negative cycle that come from A. It is easy to verify that I is indeed an interpolant.

Proposition 13. *Function* INTERPOLATE$_\mathcal{Z}$ *is totally correct: if* $A \cap B \subseteq \bot$*, then* INTERPOLATE$_\mathcal{Z}(A, B)$ *terminates and ensures* $A \subseteq I$ *and* $I \cap B \subseteq \bot$*. Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$*.*

Proof. Function INTERPOLATE$_\mathcal{Z}$ has no side effect, it thus trivially maintains feasibility and well-labeledness. In the trivial cases, I is clearly an interpolant. Assume $A \neq \bot$ and $B \neq \bot$. As $A \cap B \subseteq \bot$ by contract, there exists a negative cycle C in $\min(A, B)$ by Proposition 8. As A is canonical, we can assume that no two edges are subsequent in C_A, thus the DBM I induced by C_A is clearly canonical. The properties of an interpolant directly follow from the definitions of C_A and I. □

5.3.4 Abstraction Refinement for Lazy Zone Abstraction

To maintain well-labeledness, procedures COVER and DISABLE rely on a procedure BLOCK that performs abstraction refinement by safely adjusting labels of nodes. Algorithm 6 describes two methods for abstraction refinement based on interpolation for zones. Both methods are based on pre- and post-image computation, and can be considered as a generalization of zone interpolation to sequences of transitions of a timed automaton. The main difference between the two strategies is that BLOCK$_\text{FW}$ (which we refer to as the "forward" zone interpolation strategy) propagates the interpolant forward using $post^C$; whereas BLOCK$_\text{BW}$ (which we refer to as the "backward" zone interpolation strategy) propagates "bad" zones, obtained as the complement of the interpolant, backward using pre^C.

Algorithm 6 Refinement strategies for lazy zone abstraction

ensure $W \subseteq I$
ensure $I \cap B \subseteq \bot$
function BLOCK$_\text{FW}(n, B)$ **returns** I
    if $W \cap B \subseteq \bot$ **then**
        return W
    else
        if $(m, n) \in E$ for some m **then**
            let $t = t_{(m,n)}$
            let $B' = pre^C_t(B)$
            let $A' =$ BLOCK$_\text{FW}(m, B')$
            let $A = post^C_t(A')$
        else
            let $A = Z$
        let $I =$ INTERPOLATE$_\mathcal{Z}(A, B)$
        UPDATE$(n, (Z, W \cap I))$
        return I

procedure BLOCK$_\text{BW}(n, B)$
    if $W \cap B \subseteq \bot$ **then**
        return
    else
        let $I =$ INTERPOLATE$_\mathcal{Z}(Z, B)$
        if $(m, n) \in E$ for some m **then**
            let $t = t_{(m,n)}$
            for all $B' \in \neg I$ **do**
                let $B'' = pre^C_t(B')$
                BLOCK$_\text{BW}(m, B'')$
        UPDATE$(n, (Z, W \cap I))$

In order to make proofs of correctness for the two refinement strategies more concise, we state the following simple lemmas.

Lemma 5. $post^C_t(Z) \subseteq Z' \Rightarrow post_t(\!|Z|\!) \subseteq (\!|Z'|\!)$

Lemma 6. $(\!|Z \cap Z'|\!) = (\!|Z|\!) \cap (\!|Z'|\!)$

Proposition 14. *BLOCK_{FW} is totally correct: if $Z \cap B \subseteq \bot$, then $\text{BLOCK}_{\text{FW}}(n, B)$ terminates and ensures $W \subseteq I$ and $I \cap B \subseteq \bot$ and $W \cap B \subseteq \bot$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract (Algorithm 4), $Z \cap B \subseteq \bot$ is ensured. Moreover, notice that $W \cap B \subseteq \bot$ follows from $W \subseteq I$ and $I \cap B \subseteq \bot$, thus it is sufficient to establish the latter two claims.

If $W \cap B \subseteq \bot$, then $I = W$, so $W \subseteq I$ and $I \cap B \subseteq \bot$ are trivially established. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, if n is the root, then $A = Z$. Thus $\text{INTERPOLATE}_{\mathcal{Z}}(A, B)$ can be called, and the resulting interpolant I is such that $Z \subseteq I$ and $I \cap B \subseteq \bot$. As in this case $Z = Z_0$, clearly $\Sigma_0 \subseteq (\!| I |\!)$. Thus $\Sigma_0 \subseteq (\!| W \cap I |\!)$ by *initiation* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$.

Otherwise, there exists a transition $t = t_{m,n}$ for some node m. Since $Z = post_t^C(Z')$ and $B' = pre_t^C(B)$, we have $Z' \cap B' \subseteq \bot$ for $(Z', W') = s_m$ by Lemma 4. Thus $\text{BLOCK}_{\text{FW}}(m, B')$ can be called, and as a result, A' is such that $W' \subseteq A'$ and $A' \cap B' \subseteq \bot$ by contract. As $A = post_t^C(A')$, we obtain $A \cap B \subseteq \bot$ by Lemma 4. Thus $\text{INTERPOLATE}_{\mathcal{Z}}(A, B)$ can be called, and the resulting interpolant I is such that $A \subseteq I$ and $I \cap B \subseteq \bot$. By the monotonicity of images in $\subseteq$, we have $post_t^C(W') \subseteq A$. Hence $post_t^C(W') \subseteq I$, from which $post_t(\!| W' |\!) \subseteq (\!| I |\!)$ follows by Lemma 5. Thus $post_t(\!| W' |\!) \subseteq (\!| W \cap I |\!)$ by *consecution* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$. □

Proposition 15. *BLOCK_{BW} is totally correct: if $Z \cap B \subseteq \bot$, then $\text{BLOCK}_{\text{BW}}(n, B)$ terminates and ensures $W \cap B \subseteq \bot$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $Z \cap B \subseteq \bot$ is ensured.

If $W \cap B \subseteq \bot$, then the contract is trivially satisfied. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, $\text{INTERPOLATE}_{\mathcal{Z}}(Z, B)$ can be called, and the resulting interpolant I is such that $Z \subseteq I$ and $I \cap B \subseteq \bot$. We show that at the end of the procedure, the claim $W \subseteq I$, and thus $W \cap B \subseteq \bot$ holds.

Assume n is the root node. In this case $Z = Z_0$, thus clearly $\Sigma_0 \subseteq (\!| I |\!)$. Thus $\Sigma_0 \subseteq (\!| W \cap I |\!)$ by *initiation* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$.

Now assume there exists a transition $t = t_{m,n}$ for some node m with $(Z', W') = s_m$. Let $B' \in \neg I$, and $B'' = pre_t^C(B')$. Clearly, $Z \cap B' \subseteq \bot$. As $Z = post_t^C(Z')$, we obtain $Z' \cap B'' \subseteq \bot$ by Lemma 4. Thus $\text{BLOCK}_{\text{BW}}(m, B'')$ can be called, which ensures $W' \cap B'' \subseteq \bot$ by contract. Thus $post_t^C(W') \cap B' \subseteq \bot$ by Lemma 4. Hence $post_t^C(W') \subseteq I$, from which $post_t(\!| W' |\!) \subseteq (\!| I |\!)$ follows by Lemma 5. Thus $post_t(\!| W' |\!) \subseteq (\!| W \cap I |\!)$ by *consecution* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$. □

We would like to point out that for refinement with BLOCK_{FW}, syntactically, it is sufficient to store a single zone at each node, thus obtaining a major optimization in memory consumption. In particular, it is sufficient to store Z at leaves, and store W at non-leaf nodes. This is due to the fact that while running the algorithm, Z is only necessary when EXPAND is called, and when the interpolant is computed for the initial node, in this later situation Z being obvious. On the other hand, W is only necessary when calling cover, where covering nodes are always non-leaf. Moreover, it is always safe to treat W as $\bot$ for leafs.

5.4 Evaluation

We implemented a prototype version of our algorithm and refinement strategies in the open source model checking framework Theta [c10]. Our tool performs location reachability checking on models given in a reasonable language subset[1] of the Uppaal 4.0 XTA format.

To enable comparison to the state-of-the-art, we implemented in our framework a variant of the lazy abstraction method of [HSW13] based on LU-bounds as an alternative refinement strategy for clock variables (by defining the domain, cover and disable accordingly). The main difference in our implementation compared to [HSW13] is that when performing abstraction refinement, bounds are propagated from all guards on an infeasible path, and not just from ones that contribute to the infeasibility. Because of this, refinement in the resulting algorithm is extremely cheap, but as the comparison of our data with that of [HSW13] suggests, for the models examined in both papers, the algorithm is similarly as space- and time-efficient as the original one.

The algorithms are evaluated for both breadth-first and depth-first search orders of ART expansion. By combining all the possible alternatives, this results in 6 distinct algorithm configurations:

- as search order, breadth-first (BFS) or depth-first (DFS) search,
- for refinement over clock variables, forward (FWITP) or backward (BWITP) zone interpolation, or lazy $\mathfrak{a}_{\preccurlyeq LU}$ abstraction (LU).

Each algorithm configuration is encoded as a string containing two characters, specifically the first character of the name of each selected parameter. So for example, the configuration with BFS as search order, LU as refinement strategy over clock variables is going to be encoded as BL.

As inputs we considered 51 timed automata models in total, which we divided to three distinct categories. For each model, the number of clock variables / number of discrete variables is given in parentheses.

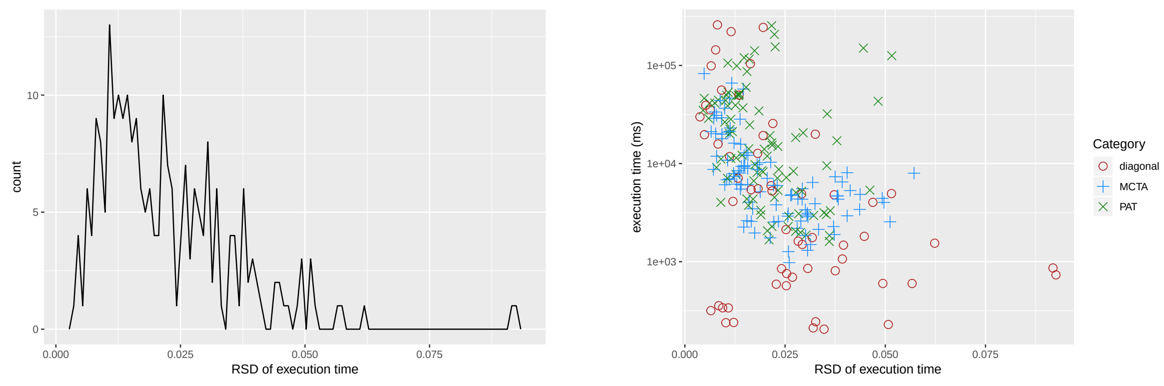
- Category PAT: classic timed automata models from the Pat benchmark set[2].
  - critical n with $n \in \{3, 4\}$ (n/1): Critical Region with n processes.
  - csma n with $n \in \{9, 10, 11, 12\}$ (n/1): CSMA/CD protocol with n processes.
  - fddi n with $n \in \{50, 70, 90, 110\}$ ($3n + 1$/1): FDDI token ring with n processes.
  - fischer n with $n \in \{7, 8, 9, 10\}$ (n/1): Fischer's mutual exclusion protocol with n processes.
  - lynch n with $n \in \{7, 8, 9\}$ (n/2): Lynch-Shavit protocol with n processes.
- Category MCTA: model containing a significant number of discrete variables (relative to the number of clock variables). Most of the models come from the Mcta benchmark set[3], while some of them come from the Uppaal benchmark set[4].
  - bocdp (3/26), bocdpf (3/26): models of the Bang & Olufsen Collision Detection Protocol obtained from the Uppaal benchmark set.
  - brp (7/7): a model of the Bounded Retransmission Protocol.
  - c1 (3/12), c2 (3/14), c3 (3/15), c4 (3/17): models of a real-time mutual exclusion protocol obtained from the Mcta benchmark set.

[1] Not supporting procedures and composite types other than arrays of synchronization channels.
[2] https://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html
[3] http://gki.informatik.uni-freiburg.de/tools/mcta/benchmarks.html
[4] https://www.it.uu.se/research/group/darts/uppaal/benchmarks

  - e1 (3/41), m1 (4/11), m2 (4/13), m3 (4/13), m4 (4/15), n1 (7/11), n2 (7/13), n3 (7/13), n4 (7/15): industrial cases studies obtained from the MCTA benchmark set.
- Fischer's protocol with diagonal constraints, based on [Rey07]
  - diag n with $n \in \{3, 4, 5, 6, 7, 8\}$ ($2n/1$): the original model, containing diagonal constraints.
  - split n with $n \in \{3, 4, 5, 6, 7, 8\}$ ($2n/n+1$): diagonal-free model obtained from diag n by eliminating diagonal constraints by introducing additional discrete variables and transitions, following the idea described in [Bér+98].
  - opt n with $n \in \{3, 4, 5, 6, 7, 8\}$ ($2n/n+1$): diagonal-free model obtained from split n by (manually) removing some guards, updates and transitions about which it can statically be established that they do not influence the set of reachable locations.

We performed our measurements on a machine running Windows 10 with a 2.6GHz dual core CPU and 8GB of RAM. We evaluated the algorithm configurations for both execution time and the number of nodes in the resulting ART. The timeout (denoted by "−" in the tables) was set to 300 seconds. The execution time shown in the following tables is the average of 10 runs, obtained from 12 deterministic runs by removing the slowest and the fastest one. For each model, the value belonging to the single best configuration, if any, is typeset in **bold**. Besides the tables shown in this chapter, tables containing all our measurement data can be found in Appendix A. Moreover, the complete set of raw measurement data, along with all input models and instructions to reproduce our experiments, are also available in a supplementary material [*s*14].

Performing location reachability checking on the models, Figure 5.1(a) shows the frequency with which different relative standard deviation (RSD) values of execution time occur. It can bee seen from the plot that higher RSD values ($> 5\%$) are relatively rare among the measurements. Moreover, Figure 5.1(b) shows how the RSD of execution time relates to the average execution time for each model and configuration (in this type of figures, each point represents the average result for a given model and configuration). Aside from a few outliers among the PAT models, it can be stated that higher RSD values belong to small average execution times, as expected. Thus it is justifiable to base the comparison of configurations on the average value.



(a) Frequency

(b) Compared to execution time (ms)

Figure 5.1: Relative standard deviation of execution time

5.4.1 Diagonal-Free Models

The detailed results for the PAT models are shown in 5.1. On these models, configurations BL and DL usually perform best in terms of execution time. When considering the size of the state space however, there is a small variability between configurations. Moreover, we point out that our results for configurations BL and DL are consistent with the results presented in [HSW13]. Detailed results for the MCTA models are shown in 5.2. Here, configurations DF gives the fastest execution on most models.

For category PAT, with respect to execution time, Fischer and Lynch provide the worst cases for our algorithm. The reason for the higher execution time despite the same number of generated nodes is that for these two models, the more costly refinement was not counterweighed by the smaller number of refinements performed, as opposed to CSMA, where the interpolation-based algorithms performed (as our logs showed) significantly less refinement steps. For FDDI, the three algorithms performed the same small number of refinement steps each, which explains the slight relative overhead of the interpolation-based algorithms. However, the three algorithms scale in the same way.

A more favorable case for our algorithm is provided by the model Critical. For this model, the interpolation-based algorithms were able to generate a 40% smaller ART. Among the two interpolation strategies, forward interpolation was somewhat more efficient in both execution time and the size of the generated ART.

Figure 5.2 shows that with respect to execution time, for the given models, all algorithms scale similarly in the number of processes of the model.

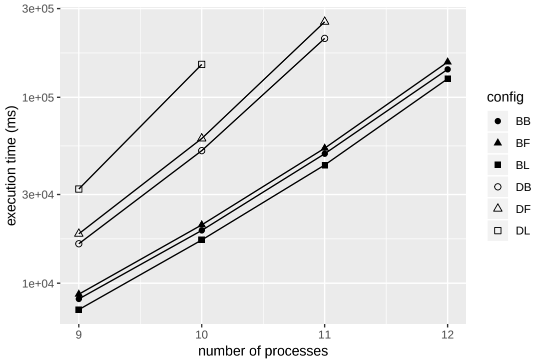
We also performed pairwise comparisons on the different algorithm configurations for each defining parameter. As can be seen on Figure 5.3, on the selected benchmark set, having all other configuration parameters fixed, clock refinement strategies FWITP and BWITP do not significantly differ in performance. On both benchmarks, FWITP slightly outperforms BWITP in the size of the generated state space. Moreover, for the MCTA models, FWITP, while for the PAT models, BWITP performs slightly better in terms of execution time (note the logarithmic scale on the axes). An explanation for this is that in general, FWITP tends to perform less refinement steps (as refinement is performed in a single iteration), whereas BWITP performs refinement steps more cheaply (as no post-image computation is involved). In our experiments, the two algorithms performed roughly the same number of refinement steps for the PAT models (probably due to discovering the same or similar simple invariants), in which case BWITP has an advantage. In the case of MCTA models however, in general, the number of refinement steps performed was in favor of FWITP.

Clock refinements LU and FWITP are compared on Figure 5.4. With respect to execution time, LU performs better in category PAT, whereas FWITP performs better in category MCTA. However, with respect to +the size of the state space, FWITP outperforms LU.

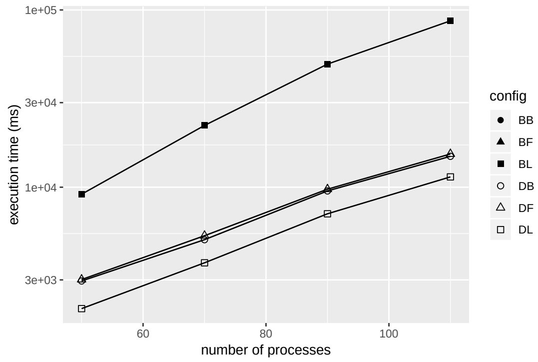
Figure 5.5 compares the impact of the two search orders on performance. With respect to execution time, DFS generally outperforms BFS on the MCTA models, whereas on the PAT models, the performance of the two search orders is balanced. When considering the size of the state space, the tendency is similar.

5.4.2 Models with Diagonal Guards

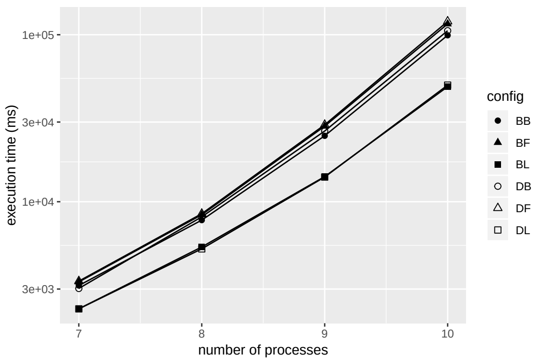
We also evaluated how the different configurations are able to handle models with diagonal constraints. As our benchmark, we used the diagonal version of Fischer's mutual exclusion algorithm, as presented in [Rey07]. We considered two approaches:



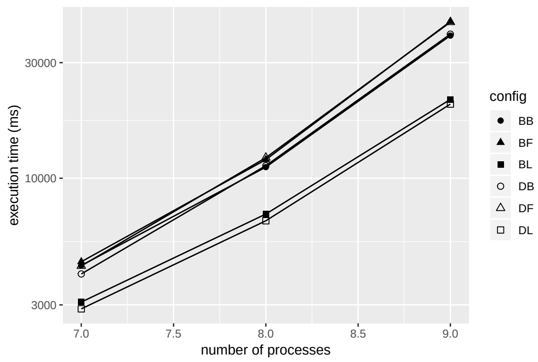
(a) CSMA



(b) FDDI

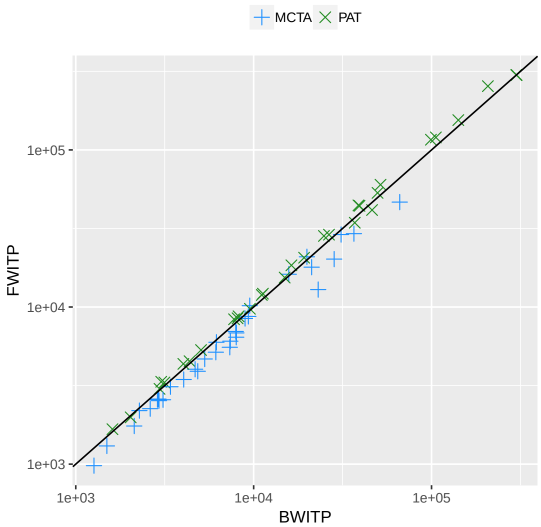


(c) Fischer

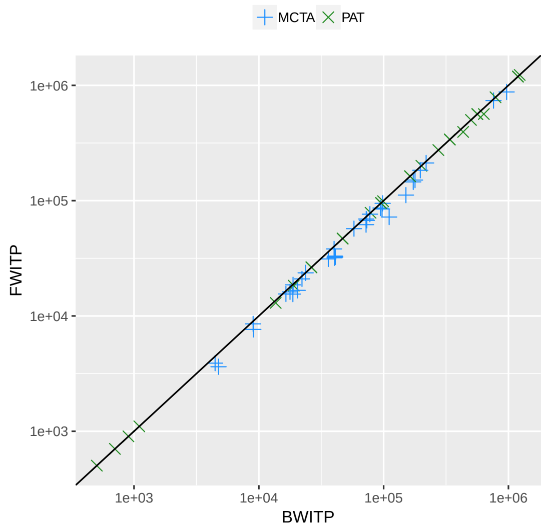


(d) Lynch

Figure 5.2: Scaling of execution time (ms) with number of processes

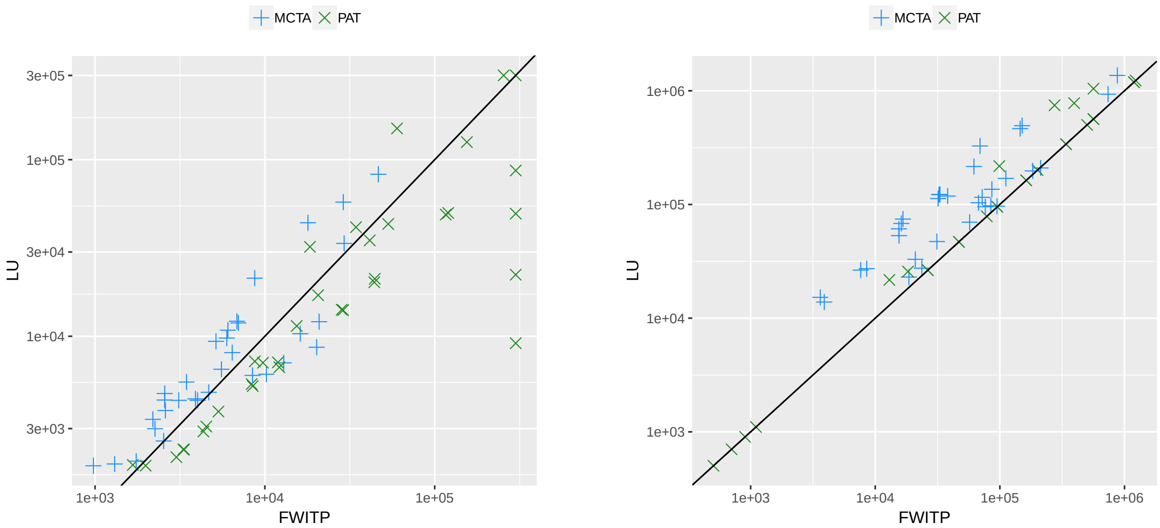


(a) Execution time (ms)



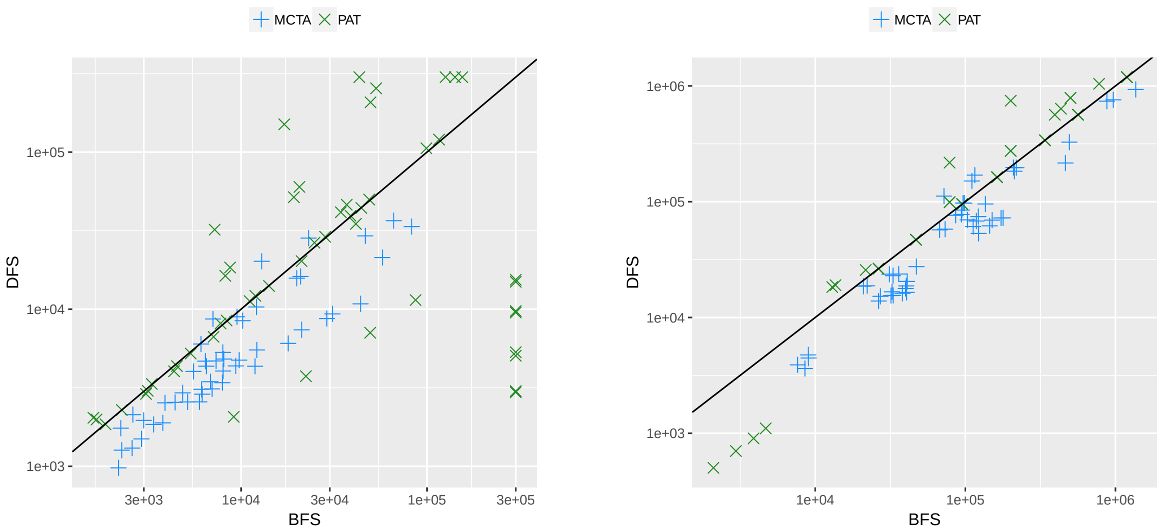
(b) Number of nodes

Figure 5.3: Clock refinement: FWITP vs. BWITP



(a) Execution time (ms)   (b) Number of nodes

Figure 5.4: Clock refinement: LU vs. FWITP



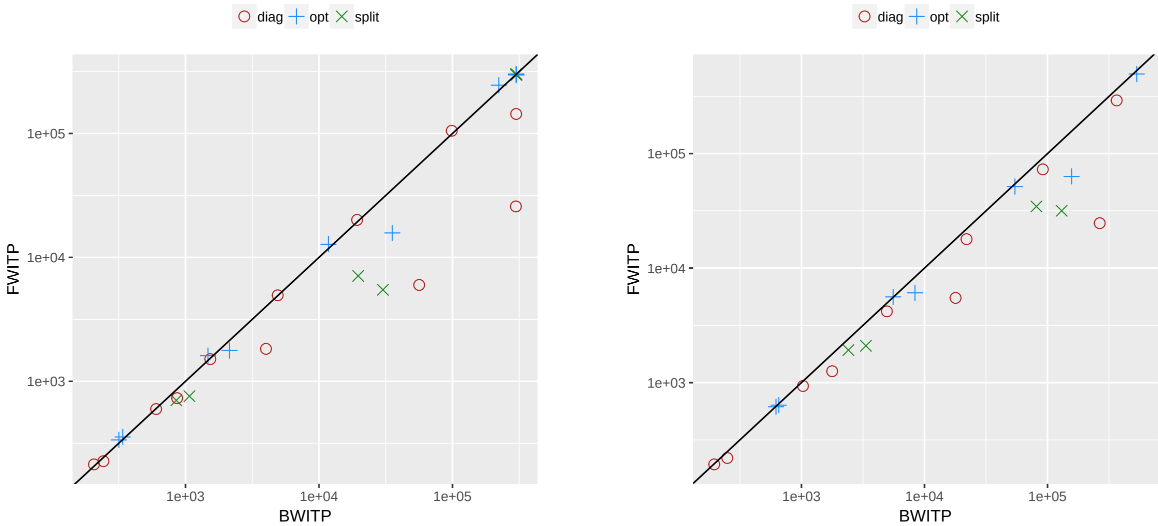
(a) Execution time (ms)   (b) Number of nodes

Figure 5.5: Search order: DFS vs. BFS

1. Eager elimination of difference constraints by introducing new discrete variables (models split n and manually optimized versions opt n).
2. Applying abstraction refinement to the model with diagonal constraints directly (models diag n).

5.3 shows our detailed measurement data for all three types of models.

In case of models diag n, clock refinement strategy LU is not applicable. The other four configurations, using FWITP for the handling of clocks, perform well regardless of search strategy, with BF being the fastest. In fact, in case of this particular model, not eliminating diagonal constraints, and using zone interpolation seems to be the best of the examined approaches.



(a) Execution time (ms)

(b) Number of nodes

Figure 5.6: Clock refinement: FWITP vs. BWITP

As Figure 5.6, shows, there is a significant difference in the performance of the two interpolation strategies, with FWITP having the better performance.

Finally, we point out that in case a model with diagonal constraints is analyzed by applying zone interpolation on its own (e.g. without zone splitting [BY04]), then termination is not guaranteed. In particular, during our experiments, we found that the algorithm diverges on the well-known example presented in [Bou03].

5.5 Conclusions

In this chapter we proposed a lazy reachability checking algorithm for timed automata based on interpolation for zones. Moreover, we proposed two refinement strategies, both a combination of forward search, backward search and interpolation. We demonstrated with experiments that - even without the use of extrapolation - the method is competitive with sophisticated non-convex abstractions in both execution time and memory consumption.

Future Work. As the method we proposed computes abstractions in terms of zones, it is straightforward to combine it with existing zone-based abstractions for timed automata. In particular, we believe that a combination with $\mathfrak{a}_{\preccurlyeq LU}$ would potentially yield a more efficient method with no considerable overhead, as backward propagation of LU-bounds is much cheaper than the propagation of interpolants. In this setting, interpolation can be considered as a further reduction on top of $\mathfrak{a}_{\preccurlyeq LU}$ abstraction.

An interesting application of our approach would be to apply it to further expressive variants of timed automata, e.g. to updatable timed automata [Bou04] with updates of the form $x_i \coloneqq c$ or $x_i \coloneqq x_i + c$ (shift) or $x_i \coloneqq x_j$ (copy) or, more generally, even $x_i \coloneqq x_j + c$. As all these operations yield zones both for forward and backward computation, with a generalization of pre^C and $post^C$, the approach becomes directly applicable. Naturally, due to general undecidability and the lack of a suitable extrapolation operator, termination can not be guaranteed in some of these cases [Bou04].

We note that by switching the role of pre^C and $post^C$ in the algorithm, a variant can be obtained that performs backward exploration in a lazy manner. Such an algorithm might result in an interesting method for simple timed automata with a restricted use of integer operations. Moreover, we note that although our current implementation is based on DBMs, the adaptation of the method to e.g. minimal constraint systems is straightforward, and is possibly more efficient.

5.5.1 Thesis Summary

This concludes Thesis 2 of this dissertation. We summarize it as follows.

> **Thesis 2** *Lazy reachability checking for timed automata using interpolants.* I proposed a solution for the location reachability problem of timed automata based on the following steps.
>
> - I defined interpolation for zones, and gave an algorithm for computing a zone interpolant from two inconsistent zones, represented as canonical difference bound matrices.
> - Based on pre- and post-image computation for timed automata in the zone abstract domain, I generalized the notion of zone interpolation to sequences of interpolants, this way enabling its use for abstraction refinement-based location reachability checking of timed automata.
> - I proposed forward and backward zone interpolation as approaches to lazy abstraction refinement.
> - I experimentally evaluated the performance of the proposed abstraction refinement strategies, and showed that these compare favorably to known methods based on efficient lazy non-convex abstractions.

Table 5.1: Detailed results for Pat models

a Execution time (s)

Model	BB	BF	BL	DB	DF	DL
critical 3	**1.6**	1.7	1.9	2.0	2.0	1.8
critical 4	37.0	**34.4**	41.4	46.3	41.4	34.9
csma 9	8.2	8.7	**7.2**	16.3	18.4	32.1
csma 10	19.2	20.6	**17.1**	51.6	60.0	150.3
csma 11	49.7	53.2	**43.2**	207.4	254.7	–
csma 12	141.4	154.8	**125.8**	–	–	–
fddi 50	–	–	9.1	3.0	3.0	**2.1**
fddi 70	–	–	22.3	5.1	5.3	**3.7**
fddi 90	–	–	49.5	9.5	9.7	**7.1**
fddi 110	–	–	86.8	14.9	15.4	**11.4**
fischer 7	3.1	3.3	2.3	3.0	3.3	**2.3**
fischer 8	7.8	8.4	5.4	8.1	8.5	**5.2**
fischer 9	24.8	28.3	14.1	26.5	28.9	**14.1**
fischer 10	99.2	116.1	**48.9**	105.7	120.1	49.9
lynch 7	4.4	4.5	3.1	4.0	4.3	**2.9**
lynch 8	11.1	11.9	7.1	11.3	12.2	**6.7**
lynch 9	38.8	44.4	21.2	39.3	44.1	**20.2**

b Number of nodes

Model	BB	BF	BL	DB	DF	DL
critical 3	13641	**12981**	21699	19036	18310	25697
critical 4	433787	**394525**	777784	635308	564014	1043487
csma 9	78552	78552	78552	98989	98989	217656
csma 10	200649	200649	200649	274759	274759	745149
csma 11	501432	501432	501432	787898	787898	–
csma 12	1230757	1230757	1230757	–	–	–
fddi 50	–	–	2098	503	503	503
fddi 70	–	–	2961	703	703	703
fddi 90	–	–	3881	903	903	903
fddi 110	–	–	4678	1103	1103	1103
fischer 7	26405	26405	26405	26405	26405	26405
fischer 8	95353	95353	95353	95353	95353	95353
fischer 9	339211	339211	339211	339211	339211	339211
fischer 10	1191211	1191211	1191211	1191211	1191211	1191211
lynch 7	46915	46915	46915	46915	46915	46915
lynch 8	162801	162801	162801	162801	162801	162801
lynch 9	563491	563491	563491	563491	563491	563491

Table 5.2: Detailed results for Mcta models

a Execution time (s)

Model	BB	BF	BL	DB	DF	DL
bocdp	9.5	10.2	6.1	9.0	8.5	**6.0**
bocdpf	19.9	20.9	12.1	15.8	16.2	**10.3**
brp	23.1	12.9	**7.1**	28.4	20.2	8.7
c1	2.6	2.3	3.0	2.1	**1.7**	2.0
c2	7.3	5.5	6.5	4.7	**4.0**	4.3
c3	8.0	6.4	8.1	5.3	**4.7**	4.8
c4	66.2	46.6	82.7	36.6	**29.3**	33.6
e1	4.8	3.9	4.4	2.9	**2.5**	2.6
m1	2.3	2.2	3.4	1.3	**1.0**	1.8
m2	6.1	5.2	9.4	3.1	**2.6**	4.4
m3	6.2	6.0	9.8	2.9	**2.6**	4.7
m4	21.2	17.9	43.9	7.4	**6.1**	10.8
n1	2.9	2.6	3.8	1.5	**1.3**	1.9
n2	7.9	7.0	11.9	3.4	**3.1**	4.3
n3	8.0	6.8	12.2	4.0	**3.5**	5.5
n4	31.0	28.9	57.5	9.3	**8.7**	21.3

b Number of nodes

Model	BB	BF	BL	DB	DF	DL
bocdp	98314	94801	96460	97125	**84643**	97462
bocdpf	218745	212225	209430	196782	**183402**	197234
brp	110600	**72117**	115675	150970	111705	169672
c1	22157	20967	32963	18802	**18614**	22968
c2	73326	67433	103476	57896	**57170**	69760
c3	94286	86285	136015	77698	**76335**	95548
c4	968171	876266	1365289	758739	**737964**	932334
e1	35989	31247	47199	23729	**23657**	27513
m1	8998	8541	27216	4753	**3625**	15233
m2	40413	31932	112634	18737	**15471**	60995
m3	40054	38128	118485	17797	**16189**	68091
m4	172868	145378	464477	72302	**61915**	215984
n1	9030	7645	26467	4466	**3898**	13869
n2	40640	33054	122680	16477	**15514**	53212
n3	40983	32493	122178	20484	**16677**	74393
n4	178362	150864	493530	72527	**69308**	326938

Table 5.3: Detailed results for diagonal models

a Execution time (s)

Model	BB	BF	BL	DB	DF	DL
diag 3	**0.2**	0.2	–	0.2	0.2	–
diag 4	**0.6**	0.6	–	0.9	0.7	–
diag 5	1.5	**1.5**	–	4.0	1.8	–
diag 6	**4.9**	4.9	–	56.1	6.0	–
diag 7	**19.3**	19.9	–	–	25.7	–
diag 8	**99.2**	104.1	–	–	144.2	–
split 3	0.8	0.7	0.6	1.1	0.8	**0.6**
split 4	19.7	7.1	5.5	30.0	5.4	**5.3**
split 5	–	–	**259.4**	–	–	–
split 6	–	–	–	–	–	–
split 7	–	–	–	–	–	–
split 8	–	–	–	–	–	–
opt 3	0.3	0.3	0.2	0.3	0.4	**0.2**
opt 4	1.5	1.6	0.9	2.1	1.8	**0.8**
opt 5	11.8	12.7	4.8	35.4	15.8	**4.1**
opt 6	221.3	244.4	49.9	–	–	**39.3**
opt 7	–	–	–	–	–	–
opt 8	–	–	–	–	–	–

b Number of nodes

Model	BB	BF	BL	DB	DF	DL
diag 3	199	**193**	–	246	220	–
diag 4	1045	**933**	–	1800	1262	–
diag 5	4926	**4181**	–	17929	5515	–
diag 6	21685	**17815**	–	264445	24772	–
diag 7	90252	**73137**	–	–	100147	–
diag 8	360233	**291593**	–	–	406392	–
split 3	2448	**1929**	3137	3277	2096	3322
split 4	79998	**34579**	68999	132835	31827	82939
split 5	–	–	**1572515**	–	–	–
split 6	–	–	–	–	–	–
split 7	–	–	–	–	–	–
split 8	–	–	–	–	–	–
opt 3	621	**619**	621	652	639	655
opt 4	**5534**	5591	5666	8234	6092	5837
opt 5	53714	51465	**51431**	155731	63504	54586
opt 6	525802	494997	**474498**	–	–	541533
opt 7	–	–	–	–	–	–
opt 8	–	–	–	–	–	–

Chapter 6

Lazy Reachability Checking for Timed Automata with Discrete Variables

In the context of timed automata, methods rarely address the problem of *abstraction for discrete data variables* that often appear in specifications for practical real-time systems, or do so by applying a fully SMT based approach, relying on the efficiency of underlying decision procedures for the abstraction of both continuous and discrete variables.

In our work, we address the location reachability problem of timed automata with discrete variables by proposing an abstraction method that can be used to *lazily control the visibility of discrete variables* occurring in such specifications: if the abstraction is too coarse to disable an infeasible transition, then we propagate the pre-image of the transition backward using weakest precondition computation, and use interpolation (defined for variable assignments) to extract a set of visible variables [Kur94; CGS04; Cha+02; Gru06] that are sufficient to block the transition from the abstract state. We use interpolation in a similar fashion to attempt to enforce coverage of a newly discovered state with an already visited state when possible, this way effectively pruning the search space. Our method does not rely on an interpolating SMT solver, and can be freely combined with zone-based forward search (eager or lazy) methods for efficient handling of clock variables.

We evaluated the proposed abstraction method by combining it with lazy refinement techniques for continuous variables. Results show that in terms of execution time our method performs similarly to lazy methods without abstraction of discrete variables, but generates a smaller (in cases significantly smaller) state space.

6.1 Related Work

Symbolic handling of integer variables for timed automata is often supported by unbounded fully symbolic SMT-based approaches. Symbolic backward search techniques like [CGR10] and [MPS11] are based on the computation and satisfiability checking of pre-images. In [Hoj+14], reachability checking for timed automata is addressed by solving Horn clauses. In the IC3-based [Bra11] technique of [KJN12b], the problem of discrete variables is not addressed directly, but the possibility of generalization over discrete variables is (to some extent) inherent in the technique. In [IW14], also based on IC3, generalization of counterexamples to induction is addressed for both discrete and clock variables by zone-based pre-image computation. The abstraction methods proposed in our work are *completely theory agnostic*, and do not rely on an SMT-solver.

In [DKL07], an abstraction refinement algorithm is proposed for timed automata that handles clock and discrete variables in a uniform way. There, given a set of visible variables, an abstracted timed automaton is derived from the original by removing all assignments to abstracted variables, and by replacing all constraints by the strongest constraint that is implied and that does not contain abstracted variables. In case the model checker finds an abstract counterexample, a linear test automaton is constructed for the path, which is then composed with the original system to check whether the counterexample is spurious. If the final location of the test automaton is unreachable, a set of relevant variables is extracted from the disabled transition that will be included in the next iteration of the abstraction refinement loop. In our work, we use a similar approach, but instead of building abstractions globally on the system level and then calling to a model checker for both model checking and counterexample analysis, we use a more integrated, lazy abstraction method, where the *abstraction is built on-the-fly, and refinement is performed locally in the state space* where more precision is necessary.

Interpolation for variable assignments was first described in [BL13]. There, the interpolant is computed for a prefix and a suffix of a constraint sequence, and an inductive sequence of interpolants is computed by propagating interpolants forward using the abstract post-image operator. In our work, we define interpolation for a variable assignment and a formula, and compute inductive sequences of interpolants by *propagating interpolants both forward and backward*, using post-image and weakest precondition computation, respectively. In our context, this enables us to consider a suffix of an infeasible path, instead of the whole path, for computing inductive sequences of interpolants.

Timed automata with diagonal constraints are exponentially more concise than diagonal-free timed automata [BC05]. In [Bér+98], a method has been proposed that eliminates diagonal constraints occurring in timed automata specifications, resulting in an (in general) exponential blowup in the size of the automaton. An extrapolation method has been proposed in [BY04] that handles diagonal constraints on-the-fly. A refinement-based approach has been described in [Bou04] that does not remove all diagonal constraints systematically. Instead, it performs forward model checking using the standard extrapolation operator used for diagonal-free timed automata, which might admit false negatives. In case a counterexample is found, it is analyzed for feasibility. If the counterexample is spurious, a set of diagonal constraints is selected and eliminated from the model, resulting in a new model, which is then fed back to the model checker. An implementation of the algorithm is described in [Rey07]. In [GMS18], the *LU*-abstraction based simulation relation of [Beh+04] is extended to models with diagonal constraints. The corresponding simulation test, which generalizes the inclusion test defined in [HSW12] for the diagonal-free setting, is shown to be NP-complete, and is implemented in terms of SMT solving. In our work, we examine two methods for analyzing timed automata with diagonal constraints. The first is based on the eager elimination of diagonal constraints, however, as our algorithms support discrete variables, instead of introducing new locations, we *introduce a new discrete variable per constraint*. In case abstraction refinement is used for these variables [*c*11], a method is obtained that considers constraints as needed, similarly to [Bou04]. However, instead of building a new model and running the model checker from scratch, this method is lazy, and performs abstraction refinement *locally in the state space where more precision is necessary*. The second approach is based on zone interpolation, which supports diagonal constraints, as well as other extensions [*c*9], automatically. Thus in this case, elimination of diagonal constraints is not necessary. Unfortunately, this method is not complete in itself, as without a suitable abstraction function, it does not guarantee termination on all models. A more recent, complete algorithm for the problem appeared in [GMS19] that is based on a novel simulation relation for timed automata with diagonal constraints. For the model on which both methods have been evaluated, the two algorithms exhibit similar performance. An improved version of this approach, focusing on updatable timed automata, appeared in [GMS20].

We provide an algorithmic framework in which we uniformly formalize, prove correct and evaluate our abstraction refinement strategies and their combinations. Moreover, besides a refinement strategy that propagates interpolants backward, we introduce a novel strategy that performs abstraction refinement by forward propagation of interpolants. Furthermore, we present an empirical evaluation of the algorithm configurations that the framework offers on a benchmark containing 51 timed automata models. In particular, we examine how the different configurations perform on models containing diagonal constraints.

6.2 Abstraction and Refinement for Discrete Variables

In the following, we describe strategies for the handling of discrete variables that appear in timed automata specifications.

6.2.1 Explicit Tracking of Variables

The most straightforward way for the handling discrete variables is to explicitly track their value.

> **Definition 6.1 (Explicit domain).** *Let $\mathcal{E} = \mathcal{V}(D)$. We define the abstraction that tracks discrete variables explicitly as the abstract domain $\mathbb{D}_{\mathcal{E}} = \left(\mathcal{E}, =, \nu_0, post^D, (\!|\cdot|\!)\right)$.* ▪

Proposition 16. *$\mathbb{D}_{\mathcal{E}}$ is sound.*

Similarly to zone abstraction, we define $\textsc{cover}_{\mathcal{E}}$ to be a no-op, thus its total correctness is trivial. Moreover, let $\textsc{disable}_{\mathcal{E}}(n, t) \mathrel{\hat{=}} (\mathsf{post}_t(\nu) \sqsubseteq \bot)$ where $\nu = s_n$.

Proposition 17. *$\textsc{disable}_{\mathcal{E}}$ is totally correct: $\textsc{disable}_{\mathcal{E}}(n, t)$ terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* false *iff t is data-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof. Termination of the procedure is trivial. Well-labeledness and feasibility follow from the fact that the procedure has no side effects. Let π be the path induced by n. Notice that $\nu = post^D_\pi(\nu_0)$. Assume $post^D_t(\nu) \neq \bot$. Then by definition, t is data-feasible from n, and the procedure returns false. Now assume $post^D_t(\nu) = \bot$. Then by definition, t is not data-feasible from n. But t is also disabled from $(\!|\nu|\!)$, and the procedure returns true. □

6.2.2 Visible Variables Abstraction

Instead of explicitly tracking in all states the values for all variables, by tracking in each state only those that play a role in unreachability of a given location along a path through the state, and "hiding" all the others, the size of the explored state space can be significantly reduced. In the following, we describe such an abstract domain, together with the corresponding refinement strategies.

> **Definition 6.2 (Visible variables domain).** *Let $\mathbb{D}_{\mathcal{E}\mathcal{I}} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ be the abstract domain over $\mathbb{D}_{\mathcal{E}}$ with*
> - *$\mathcal{S} = \mathcal{V}(D) \times \mathcal{P}(D)$,*
> - *$(\nu, Q) \sqsubseteq (\nu', Q')$ iff $\nu \preceq \nu'{\restriction}_{Q'}$ and $Q' \subseteq Q$ (thus $\sqsubseteq$ is a preorder),*

- $\mathsf{init} = (\mathsf{init}, \emptyset)$,
- $\mathsf{post}_t(\nu, Q) = (\mathsf{post}_t(\nu), \emptyset)$, *and*
- $[\![(\nu, Q)]\!] = (\!|\nu{\restriction_Q}|\!)$. ▪

Proposition 18. *$\mathbb{D}_{\mathcal{EI}}$ is sound.*

Algorithm 7 describes the corresponding refinement methods. Both $\textsc{cover}_{\mathcal{EI}}$ and $\textsc{disable}_{\mathcal{EI}}$ rely on a procedure $\textsc{refine}$ for abstraction refinement. Moreover, $\textsc{disable}_{\mathcal{EI}}$ depends on a weakest precondition operator, defined by the following property.

Definition 6.3 (Weakest discrete precondition). *Let $wp^D_t(\varphi)$ be the formula such that $\nu \models wp^D_t(\varphi)$ iff $post^D_t(\nu) \models \varphi$ for all ν and φ, with respect to t.* ▪

Algorithm 7 Visible variables abstraction

procedure $\textsc{cover}_{\mathcal{EI}}(n, n')$
  let $(\nu, \cdot) = s_n$
  let $(\nu', Q') = s_{n'}$
  if $\nu \preceq \nu'{\restriction_{Q'}}$ **then**
    $\textsc{refine}(n, \mathsf{form}(\nu'{\restriction_{Q'}}))$

function $\textsc{disable}_{\mathcal{EI}}(n, t)$
  let $(\nu, \cdot) = s_n$
  let $\nu' = post^D_t(\nu)$
  if $\nu' = \bot$ **then**
    $\textsc{refine}(n, wp^D_t(\bot))$
    return true
  else
    return false

invariant $\mathcal{G}$ is well-labeled and feasible
define $(\nu, Q) = s_n$
require $\nu \models \varphi$
ensure $\nu{\restriction_Q} \models \varphi$
procedure $\textsc{refine}(n, \varphi)$

In $\textsc{cover}_{\mathcal{EI}}$, as $\nu \preceq \nu'{\restriction_{Q'}}$, we have $\nu \models \mathsf{form}(\nu'{\restriction_{Q'}})$ by Lemma 2, thus calling $\textsc{refine}(n, \mathsf{form}(\nu'{\restriction_{Q'}}))$ is safe. Other than that, total correctness of $\textsc{cover}_{\mathcal{EI}}$ follows trivially from total correctness of $\textsc{refine}$ (see later).

Proposition 19. *$\textsc{disable}_{\mathcal{EI}}$ is totally correct: $\textsc{disable}_{\mathcal{EI}}(n, t)$ terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns false iff t is data-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof. Termination of the procedure is trivial. Well-labeledness and feasibility follow from the total correctness of $\textsc{refine}$. Let π be the path induced by n. Notice that $\nu = post^D_\pi(\nu_0)$. Assume $post^D_t(\nu) \neq \bot$. Then by definition, t is data-feasible from n, and the procedure returns false. Now assume $post^D_t(\nu) = \bot$. Then by definition, t is not data-feasible from n. As $post^D_t(\nu) \models \bot$, by Definition 6.3, we get $\nu \models wp^D_t(\bot)$. Thus $\textsc{refine}(n, wp^D_t(\bot))$ can be called, and as a result, $\nu{\restriction_Q} \models wp^D_t(\bot)$. By Definition 6.3, we get $post^D_t(\nu{\restriction_Q}) \models \bot$, thus clearly $post^D_t(\nu{\restriction_Q}) = \bot$. Thus t becomes disabled from $(\!|\nu{\restriction_Q}|\!)$, and the procedure returns true. □

6.2.3 Interpolation for Valuations

The proposed refinement strategies for discrete variables, and in particular, different implementations of REFINE are based on the notion of a valuation interpolant, defined over a valuation and a formula.

> **Definition 6.4 (Valuation interpolant).** *Given a valuation σ and a formula φ such that $\sigma \models \varphi$, a valuation interpolant is a valuation σ' such that $\sigma \preceq \sigma'$ and $\sigma' \models \varphi$ and $\mathsf{def}(\sigma') \subseteq \mathsf{def}(\sigma) \cap \mathsf{vars}(\varphi)$.* ▪

Algorithm 8 Interpolation for valuations

invariant $\mathcal{G}$ is well-labeled and feasible
require $\sigma \models \varphi$
ensure $\sigma\lceil_I$ is an interpolant for σ and φ
function INTERPOLATE$_{\mathcal{E}}(\sigma, \varphi)$ **returns** I
    let $X = \mathsf{def}(\sigma) \cap \mathsf{vars}(\varphi)$
    $I \leftarrow X$
    for all $x \in X$ **do**
        let $I' = I \setminus \{x\}$
        if $\sigma\lceil_{I'} \models \varphi$ **then**
            $I \leftarrow I'$
    return I

Proposition 20. *Function INTERPOLATE$_{\mathcal{E}}$ is totally correct: if $\sigma \models \varphi$, then INTERPOLATE$_{\mathcal{E}}(\sigma, \varphi)$ terminates and ensures $\sigma\lceil_I \models \varphi$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

Proof. Function INTERPOLATE$_{\mathcal{E}}$ has no side effect, it thus trivially maintains feasibility and well-labeledness. Moreover, it is easy to see that it satisfies its contract, as the postcondition is an invariant for the loop. □

Next, we show how valuation interpolants can be used for hiding variables that are irrelevant with respect to the reachability of a given location along a path.

6.2.4 Abstraction Refinement for Visible Variables Abstraction

Algorithm 9 outlines two strategies for abstraction refinement over the visible variables abstract domain. Symmetrically to the variants of BLOCK, procedure REFINE$_{\mathrm{FW}}$ (which we refer to as the "forward" valuation interpolation strategy) propagates interpolants forward using $post^D$; whereas procedure REFINE$_{\mathrm{BW}}$ (which we refer to as the "backward" valuation interpolation strategy) propagates interpolants backward using wp^D along the path to be refined.

To make our formal description more concise, we state the following simple lemmas.

Lemma 7. $\alpha \preceq \beta \Rightarrow post_t^D(\alpha) \preceq post_t^D(\beta)$

Lemma 8. $post_t^D(\nu) \preceq \nu' \Rightarrow post_t(\!|\nu|\!) \subseteq (\!|\nu'|\!)$

Lemma 9. $(\!|\nu\lceil_{A \cup B}|\!) = (\!|\nu\lceil_A|\!) \cap (\!|\nu\lceil_B|\!)$

Algorithm 9 Refinement strategies for visible variables abstraction

ensure $I \subseteq Q$
ensure $\nu\restriction_I \models \varphi$
function $\text{REFINE}_{\text{FW}}(n, \varphi)$ **returns** $\nu\restriction_I$
    if $\nu\restriction_Q \models \varphi$ **then**
        return $\nu\restriction_Q$
    else
        if $(m, n) \in E$ for some m **then**
            let $t = t_{(m,n)}$
            let $\varphi' = wp_t^D(\varphi)$
            let $\alpha' = \text{REFINE}_{\text{FW}}(m, \varphi')$
            let $\alpha = post_t^D(\alpha')$
        else
            let $\alpha = \nu$
        let $I = \text{INTERPOLATE}_{\mathcal{E}}(\alpha, \varphi)$
        $\text{UPDATE}(n, (\nu, Q \cup I))$
        return $\nu\restriction_I$
procedure $\text{REFINE}_{\text{BW}}(n, \varphi)$
    if $\nu\restriction_Q \models \varphi$ **then**
        return
    else
        let $I = \text{INTERPOLATE}_{\mathcal{E}}(\nu, \varphi)$
        if $(m, n) \in E$ for some m **then**
            let $t = t_{(m,n)}$
            let $\varphi' = wp_t^D(\mathsf{form}(\nu\restriction_I))$
            $\text{REFINE}_{\text{BW}}(m, \varphi')$
        $\text{UPDATE}(n, (\nu, Q \cup I))$

Proposition 21. *$\text{REFINE}_{\text{FW}}$ is totally correct: if $\nu \models \varphi$, then $\text{REFINE}_{\text{FW}}(n, \varphi)$ terminates and ensures $I \subseteq Q$ and $\nu\restriction_I \models \varphi$ and $\nu\restriction_Q \models \varphi$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $\nu \models \varphi$ is ensured. Moreover, notice that $\nu\restriction_Q \models \varphi$ follows from $I \subseteq Q$ and $\nu\restriction_I \models \varphi$ by Lemma 1, thus it is sufficient to establish the latter two claims.

If $\nu\restriction_Q \models \varphi$, then $I = Q$, so $I \subseteq Q$ and $\nu\restriction_I \models \varphi$ are trivially established. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, if n is the root, then $\alpha = \nu$. Thus $\text{INTERPOLATE}_{\mathcal{E}}(\alpha, \varphi)$ can be called, and the resulting interpolant I is such that $\nu\restriction_I \models \varphi$. As in this case $\nu = \nu_0$, clearly $\Sigma_0 \subseteq (\!|\nu\restriction_I|\!)$. Thus $\Sigma_0 \subseteq (\!|\nu\restriction_{Q \cup I}|\!)$ by *initiation* and Lemma 9. Therefore, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$.

Otherwise, there exists a transition $t = t_{m,n}$ for some node m. Since $\nu = post_t^D(\nu')$ and $\varphi' = wp_t^D(\varphi)$, we have $\nu' \models \varphi'$ for $(\nu', Q') = s_m$ by Definition 6.3. Thus $\text{REFINE}_{\text{FW}}(m, \varphi')$ can be called, and as a result, α' is such that $\alpha' = \nu'\restriction_{I'}$ and $I' \subseteq Q'$ and $\alpha' \models \varphi'$ by contract for some I'. As $\alpha = post_t^D(\alpha')$, we obtain $\alpha \models \varphi$ by Definition 6.3. Thus $\text{INTERPOLATE}_{\mathcal{E}}(\alpha, \varphi)$ can be called, and the resulting interpolant I is such that $\alpha\restriction_I \models \varphi$. Clearly $\nu' \preceq \alpha'$, thus $\nu \preceq \alpha$ by Lemma 7. Therefore, $\nu\restriction_I = \alpha\restriction_I$, as $I \subseteq \mathsf{def}(\alpha)$. From this, $\nu\restriction_I \models \varphi$ follows directly. Moreover, as $\nu'\restriction_{Q'} \preceq \nu'\restriction_{I'}'$, by Lemma 7, we have $post_t^D(\nu'\restriction_{Q'}) \preceq \alpha$. Hence $post_t^D(\nu'\restriction_{Q'}) \preceq \nu\restriction_I$, from which $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_I|\!)$ follows by Lemma 8. Thus $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_{Q \cup I}|\!)$ by *consecution* and Lemma 9. Therefore, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$. ☐

Proposition 22. *$\text{REFINE}_{\text{BW}}$ is totally correct: if $\nu \models \varphi$, then $\text{REFINE}_{\text{BW}}(n, \varphi)$ terminates and ensures $\nu\restriction_Q \models \varphi$. Moreover, it preserves well-labeledness and feasibility of $\mathcal{G}$.*

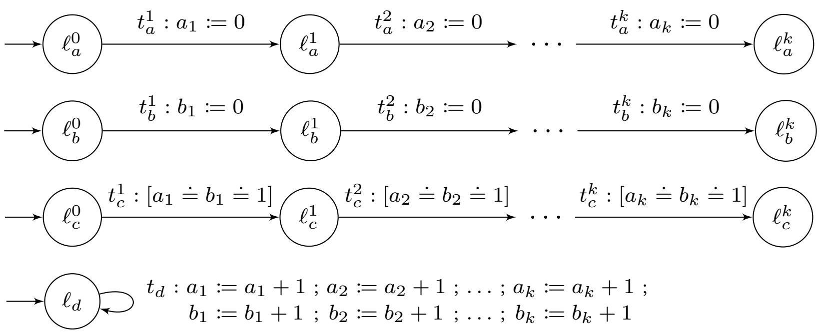


Figure 6.1: Automaton $\mathcal{A}_k$

Proof. Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $\nu \models \varphi$ is ensured.

If $\nu\lceil_Q \models \varphi$, then the contract is trivially satisfied. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise $\textsc{interpolate}_{\mathcal{E}}(\nu, \varphi)$ can be called, and the resulting interpolant I is such that $\nu\lceil_I \models \varphi$. We show that at the end of the procedure, the claim $I \subseteq Q$, and thus by Lemma 1 also $\nu\lceil_Q \models \varphi$ holds.

Assume n is the root node. In this case $\nu = \nu_0$, thus clearly $\Sigma_0 \subseteq (\!|\nu\lceil_I|\!)$. Thus $\Sigma_0 \subseteq (\!|\nu\lceil_{Q \cup I}|\!)$ follows by *initiation* and Lemma 9. As a consequence, $\textsc{update}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$.

Now assume there exists a transition $t = t_{m,n}$ for some node m with $(\nu', Q') = s_m$. Clearly, $\nu \preceq \nu\lceil_I$, thus $\nu \models \mathsf{form}(\nu\lceil_I)$ by Lemma 2. As $\nu = post_t^D(\nu')$ and $\varphi' = wp_t^D(\mathsf{form}(\nu\lceil_I))$ we obtain $\nu' \models \varphi'$ by Definition 6.3. Thus $\textsc{refine}_{\textsc{bw}}(m, \varphi')$ can be called, which ensures $\nu'\lceil_{Q'} \models \varphi'$ by contract. Thus $post_t^D(\nu'\lceil_{Q'}) \models \mathsf{form}(\nu\lceil_I)$ by Definition 6.3. Hence $post_t^D(\nu'\lceil_{Q'}) \preceq \nu\lceil_I$ by Lemma 2, from which $post_t(\!|\nu'\lceil_{Q'}|\!) \subseteq (\!|\nu\lceil_I|\!)$ follows by Lemma 8. Thus $post_t(\!|\nu'\lceil_{Q'}|\!) \subseteq (\!|\nu\lceil_{Q \cup I}|\!)$ by *consecution* and Lemma 9. As a consequence, $\textsc{update}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$. □

6.3 Example

In this section, we give an example that demonstrates how the algorithm described above lazily controls the visibility of discrete variables of the system during construction of the abstraction. We are going to consider $\textsc{refine}_{\textsc{fw}}$.

Figure 6.1 shows automaton $\mathcal{A}_k$, a modified version of the examples given in [LNZ04; HSW13] where clock variables are replaced by discrete variables and a component is added that nondeterministically increments all variables. The resulting automaton is the parallel composition of four components, and has $2k$ discrete variables, namely $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$.

As an example, we are going to consider $\mathcal{A}_1$, the simplest version of the automaton. For simplicity, we are going to omit the indexes in names whenever possible. Figure 6.2 shows part of the ART produced by the algorithm. Here, normal edges represent edges of the unwinding (elements of the relation E), dashed edges represent covering edges (elements of the relation $\rhd$), and dotted edges represent edges of the unwinding that lead to subtrees omitted from the figure. For each node, the set of visible variables is shown.

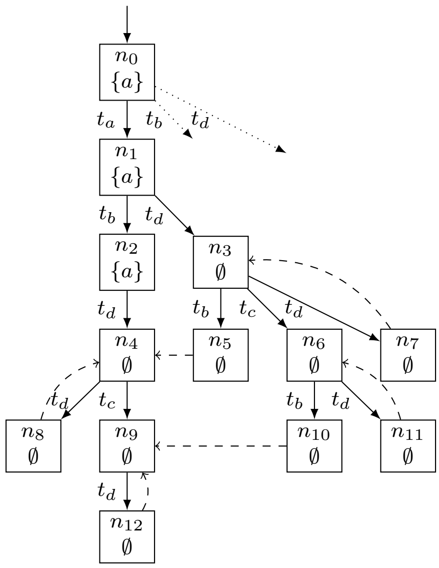


Figure 6.2: ART of $\mathcal{A}_1$

Let $s_{n_i} = s_i = (\nu_i, Q_i)$ and $\ell_{n_i} = \ell_i$ for each node n_i. The algorithm starts by instantiating the root node n_0 with $Q_0 = \emptyset$. As transition t_c is not data-feasible from n_0, but also not yet disabled from $(\!|\nu_0 \upharpoonright_{Q_0}|\!) = \top$, the set of visible variables Q_0 has to be refined. Hence during refinement, a will be included in the set of visible variables, ensuring $\nu_0 \upharpoonright_{Q_0} = \{a \hookleftarrow 0\} \models (a \not\equiv 1 \lor b \not\equiv 1) = wp^D_{t_c}(\bot)$. For the same reason, a will become visible when expanding n_1 and n_2. For any other node n_i however, t_c is either not an outgoing transition of location ℓ_i, or is enabled from $(\!|\nu_i|\!)$, thus no refinement will be triggered during expansion, resulting in the coarse abstraction $Q_i = \emptyset$. This enables coverage between nodes that assign different concrete values to the variables. For example, covering edges (n_5, n_4) and (n_{10}, n_9) are only possible because b is not visible in either nodes (as $\nu_4 = \nu_9 = \{a \hookleftarrow 1, b \hookleftarrow 1\}$ and $\nu_5 = \nu_{10} = \{a \hookleftarrow 1, b \hookleftarrow 0\}$). Even more importantly, the algorithm is able to quickly cover nodes that result from the second firing of t_d along a path, thus the resulting ART remains finite. Even if the number of times t_d can be taken is bounded by some number N, an algorithm that handles discrete variables explicitly would generate a significantly larger state space depending on N. Similarly, as k increases, the advantage of the abstraction based method compared to the explicit handling of variables becomes increasingly notable.

6.4 Evaluation

To evaluate our refinement strategies, we considered the same 51 timed automata models as inputs as in Chapter 5. We performed our measurements on a machine running Windows 10 with a 2.6GHz dual core CPU and 8GB of RAM. We evaluated the algorithm configurations for both execution time and the number of nodes in the resulting ART. By combining all the possible alternatives, this results in 18 distinct algorithm configurations.

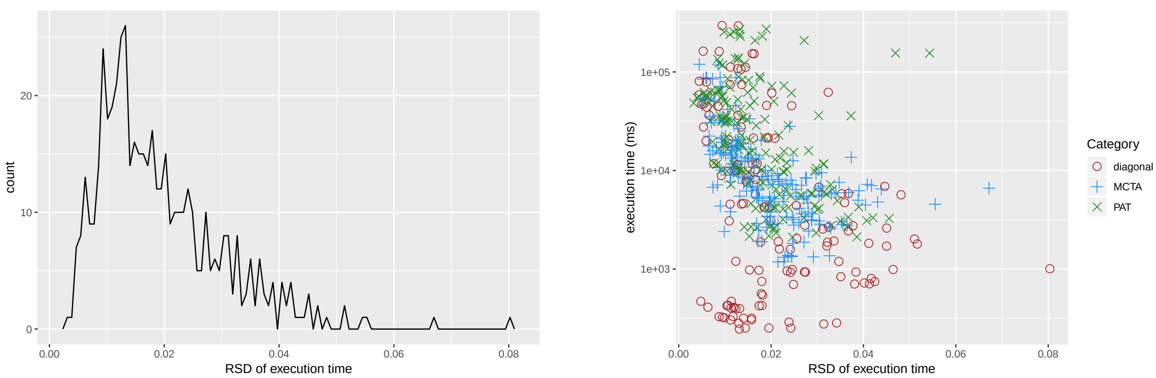
- as search order, breadth-first (BFS) or depth-first (DFS) search,
- for clock variables, forward (FWITP) or backward (BWITP) zone interpolation, or lazy $\mathfrak{a}_{\preccurlyeq LU}$ abstraction (LU),

- for discrete variables, forward (FWITP) or backward (BWITP) valuation interpolation, or no refinement (NONE).

Each algorithm configuration is encoded as a string containing three characters, specifically the first character of the name of each selected parameter. So for example, the configuration with BFS as search order, LU as refinement strategy for clock variables, and NONE as refinement strategy for discrete variables, is going to be encoded as BLN. The timeout (denoted by "–" in the tables) was set to 300 seconds. The execution time shown in the following tables is the average of 10 runs, obtained from 12 deterministic runs by removing the slowest and the fastest one. For each model, the value belonging to the single best configuration, if any, is typeset in **bold**. For comparison, the results for the best configuration without discrete refinement (· · N) are presented as well. Besides the tables shown in this chapter, tables containing all our measurement data can be found in Appendix A. Moreover, the complete set of raw measurement data, along with all input models and instructions to reproduce our experiments, are also available in a supplementary material [*s*14].

For the configurations that handle discrete variables explicitly (· · N), we partitioned the set of nodes of the ART based on the value of the data valuation, this way saving the $\mathcal{O}(n)$ cost of checking inclusion for valuations. This optimization also significantly reduces the number of nodes for which coverage is checked and attempted during CLOSE. Apart from this and the difference in refinement strategies, the implementation of the configurations is shared.

Performing location reachability checking on the models, Figure 6.3(a) shows the frequency with which different relative standard deviation (RSD) values of execution time occur. It can be seen from the plot that higher RSD values ($> 5\%$) are relatively rare among the measurements. Moreover, Figure 6.3(b) shows how the RSD of execution time relates to the average execution time for each model and configuration (in this type of figures, each point represents the average result for a given model and configuration). Aside from a few outliers among the PAT models, it can be stated that higher RSD values belong to small average execution times, as expected. Thus it is justifiable to base the comparison of configurations on the average value.

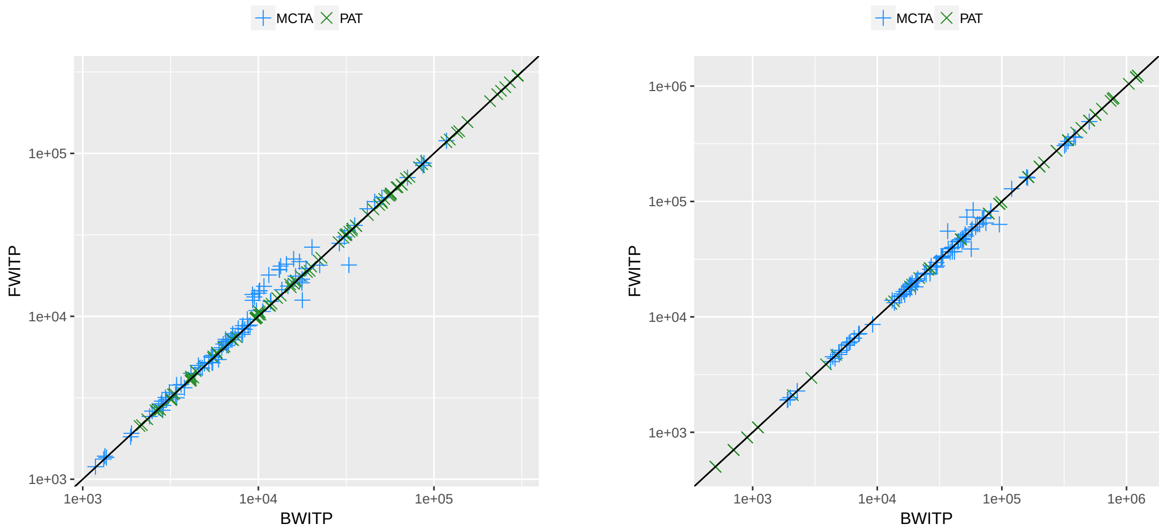


(a) Frequency (b) Compared to execution time (ms)

Figure 6.3: Relative standard deviation of execution time

6.4.1 Diagonal-Free Models

Figure 6.4 shows that on the selected benchmark set, having all other configuration parameters fixed, discrete refinement strategies FWITP and BWITP do not significantly differ in performance. Here, BWITP tends to perform better in terms of execution time. Therefore, we are going to omit detailed results discrete refinement FWITP for the rest of the section.



(a) Execution time (ms) (b) Number of nodes

Figure 6.4: Discrete refinement: FWITP vs. BWITP

The detailed results for the PAT models are shown in 6.1. As these models do not contain many discrete variables, performing refinement over discrete variables does not have a positive effect on performance, as expected. It can be observed however that the overhead of refinement is not significant. Detailed results for the MCTA models are shown in 6.2. Here, configurations DFN or DFB give the fastest execution on most models. Moreover, configuration DFB generates the least number of nodes in almost all cases, which highlights the advantages of our new interpolation based algorithm presented first in [*c*11].

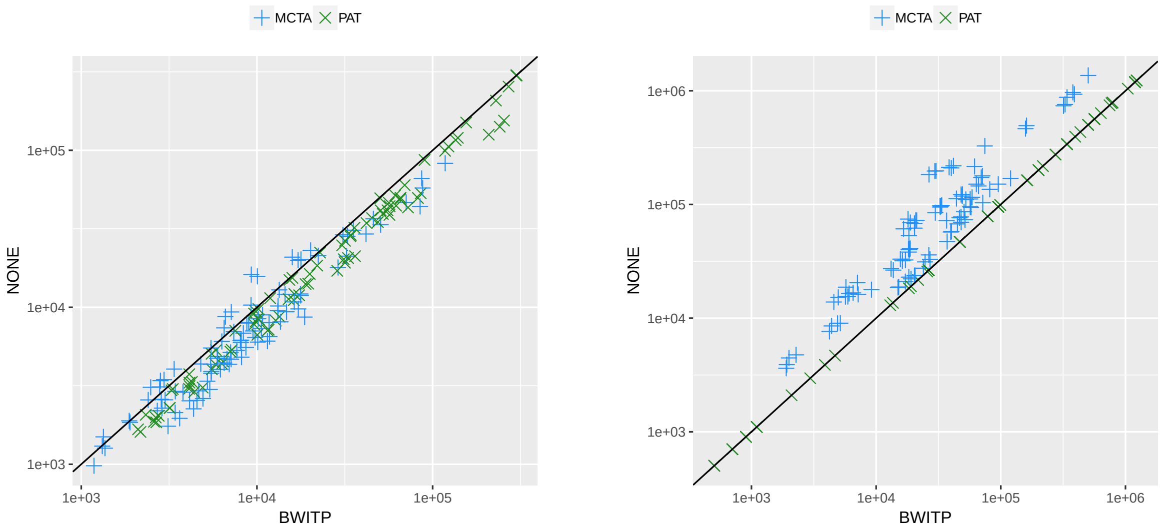
Figure 6.5 shows the pairwise comparison of interpolation-based and explicit handling of discrete variables. On the MCTA models, BWITP is always able to generate an — in some cases, significantly — smaller state space. Unsurprisingly, the same reduction effect is not present on PAT models, where there are only one or two discrete variables. Despite the significant reduction in state space, on the models considered, aside from a couple of cases, BWITP is somewhat slower. Beside the obvious overhead of running abstraction refinement, this can be explained with the optimization of coverage checking applied in the explicit case, as described above.

6.4.2 Models with Diagonal Guards

Analogously as in Chapter 5, we evaluated how the different configurations are able to handle models with diagonal constraints. 6.3 shows our detailed measurement data for all three types of models.

In case of models diag n, as the number of discrete variables is low, using zone interpolation without discrete refinement is still the fastest of the examined approaches.

Models split n, where diagonal constraints are eliminated, enable the comparison of our approach with state-of-the-art approaches presented in [Rey07; GMS18]. We point out that our results for configuration BL are consistent with the results presented in [GMS18]. In these models, by using valua-



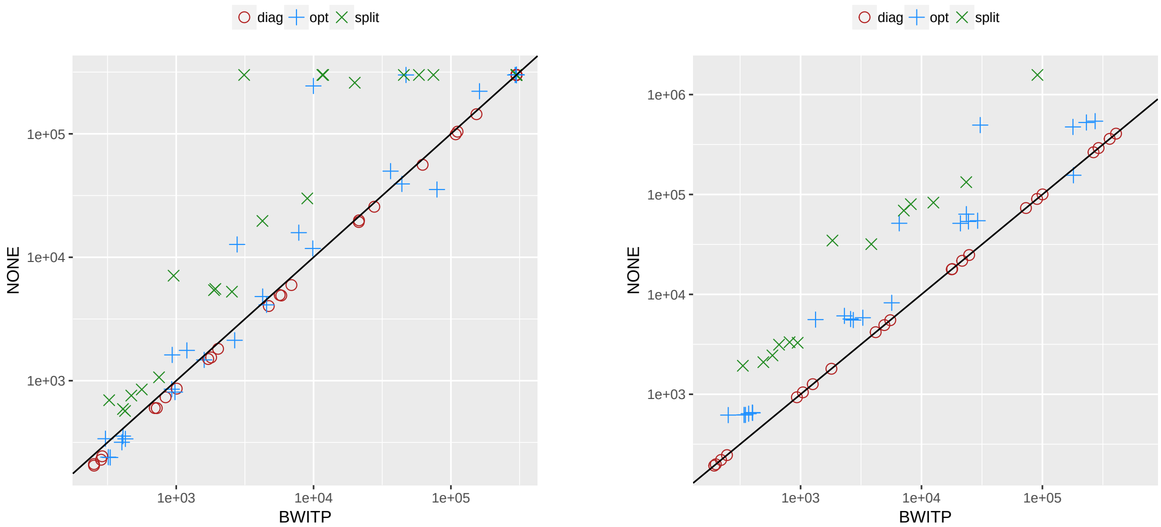
(a) Execution time (ms)

(b) Number of nodes

Figure 6.5: Discrete refinement: NONE vs. BWITP

tion interpolation, both execution times and the size of the state space can be significantly reduced. In particular, configuration BFB significantly outperforms all the other configurations.

In general, all configurations benefited greatly from the manual optimization that we applied for models opt n. However, using valuation interpolation still significantly improves performance for all configurations (Figure 6.6). Moreover, configuration BFB is still by far the most successful configuration. This also highlights the beneficial effects of combining abstraction refinement strategies for clock and discrete variables, in line with our results in [c11].



(a) Execution time (ms)

(b) Number of nodes

Figure 6.6: Discrete refinement: NONE vs. BWITP

6.5 Conclusions

In this chapter, we proposed a lazy algorithm for the location reachability problem of timed automata with discrete variables. The method is based on controlling the visibility of discrete variables by using interpolation for valuations of variables. We demonstrated with experiments that our abstraction and refinement strategy, combined with lazy methods for the abstraction of continuous clock variables, can achieve significant reduction in the size of the generated state space during search, typically with low or no overhead in execution time, and in cases even with an additional speedup.

Future Work. A interesting direction would be to experiment with different abstract domains (e.g. intervals, octahedra, or polyhedra), and investigate alternative refinement strategies for the discrete variables of timed systems. Furthermore, although we evaluated our abstraction method in the context of timed systems, the technique itself can be applied in a more general context, e.g. for model checking imperative programs.

6.5.1 Thesis Summary

This concludes Thesis 3 of this dissertation. We summarize it as follows.

> **Thesis 3** *Lazy reachability checking for timed automata with discrete variables.* I proposed a solution for the location reachability problem of timed automata with discrete variables based on the following steps.
>
> - I defined interpolation between a valuation and a formula, and gave an algorithm for computing valuation interpolants.
> - Based on weakest precondition computation for transitions of timed automata, I generalized the notion of valuation interpolation to sequences of interpolants, this way enabling its use for abstraction refinement-based location reachability checking.
> - I proposed forward and backward valuation interpolation as approaches to lazy abstraction refinement.
> - I experimentally evaluated the performance of the proposed abstraction refinement strategies, and showed that these are suitable to significantly reduce the number of states generated during state space exploration of timed automata models with many discrete variables.

Table 6.1: Detailed results for PAT models

a Execution time (s)

Model	BestN	Time	BBB	BFB	BLB	DBB	DFB	DLB
critical 3	BBN	**1.6**	2.2	2.1	2.6	2.8	2.7	2.7
critical 4	BFN	**34.4**	45.2	42.1	55.4	56.4	50.6	48.8
csma 9	BLN	**7.2**	12.8	13.4	11.7	20.0	22.0	35.9
csma 10	BLN	**17.1**	31.7	33.0	28.7	61.2	69.3	155.2
csma 11	BLN	**43.2**	82.4	85.6	72.4	229.3	270.6	–
csma 12	BLN	**125.8**	241.0	254.7	208.7	–	–	–
fddi 50	DLN	**2.1**	–	–	9.6	3.3	3.3	2.3
fddi 70	DLN	**3.7**	–	–	22.9	5.5	5.8	4.1
fddi 90	DLN	**7.1**	–	–	50.3	9.7	10.2	7.5
fddi 110	DLN	**11.4**	–	–	90.0	15.3	15.9	11.9
fischer 7	DLN	**2.3**	4.1	4.1	3.2	4.1	4.3	3.2
fischer 8	DLN	**5.2**	9.7	10.3	7.2	9.8	10.1	7.1
fischer 9	DLN	**14.1**	30.5	34.1	19.6	31.7	34.2	18.9
fischer 10	BLN	**48.9**	117.8	135.3	65.4	123.1	139.1	65.7
lynch 7	DLN	**2.9**	6.3	6.4	4.9	5.5	5.8	4.4
lynch 8	DLN	**6.7**	16.2	17.4	11.5	15.1	16.3	10.1
lynch 9	DLN	**20.2**	56.8	62.0	36.2	52.1	56.9	31.2

b Number of nodes

Model	BestN	Nodes	BBB	BFB	BLB	DBB	DFB	DLB
critical 3	BFN	12981	13641	12981	21699	19036	18310	25697
critical 4	BFN	394525	434393	395188	772221	635308	564014	1043487
csma 9	BBN	78552	78552	78552	78552	98989	98989	217656
csma 10	BBN	200649	200649	200649	200649	274759	274759	745149
csma 11	BBN	501432	501432	501432	501432	787898	787898	–
csma 12	BBN	1230757	1230757	1230757	1230757	–	–	–
fddi 50	DBN	503	–	–	2098	503	503	503
fddi 70	DBN	703	–	–	2961	703	703	703
fddi 90	DBN	903	–	–	3881	903	903	903
fddi 110	DBN	1103	–	–	4678	1103	1103	1103
fischer 7	BBN	26405	26405	26405	26405	26405	26405	26405
fischer 8	BBN	95353	95353	95353	95353	95353	95353	95353
fischer 9	BBN	339211	339211	339211	339211	339211	339211	339211
fischer 10	BBN	1191211	1191211	1191211	1191211	1191211	1191211	1191211
lynch 7	BBN	46915	46915	46915	46915	46915	46915	46915
lynch 8	BBN	162801	162801	162801	162801	162801	162801	162801
lynch 9	BBN	563491	563491	563491	563491	563491	563491	563491

Table 6.2: Detailed results for Mcta models

a Execution time (s)

Model	BestN	Time	BBB	BFB	BLB	DBB	DFB	DLB
bocdp	DLN	**6.0**	13.1	13.2	11.5	10.8	10.2	10.1
bocdpf	DLN	10.3	17.1	15.9	14.5	10.1	9.3	**9.3**
brp	BLN	**7.1**	20.2	13.4	9.5	32.8	17.8	18.7
c1	DFN	**1.7**	4.9	4.4	5.4	3.4	3.1	3.6
c2	DFN	**4.0**	10.6	8.7	11.8	6.8	6.2	7.0
c3	DFN	**4.7**	11.7	9.8	13.6	7.7	7.1	8.2
c4	DFN	**29.3**	86.6	70.7	117.8	46.0	41.7	50.6
e1	DFN	**2.5**	6.0	5.5	6.5	4.7	4.1	4.6
m1	DFN	**1.0**	2.9	2.7	5.2	1.4	1.2	1.9
m2	DFN	2.6	8.1	7.1	14.7	2.5	**2.4**	4.8
m3	DFN	**2.6**	8.1	8.1	17.2	3.8	3.0	5.9
m4	DFN	**6.1**	32.4	28.9	84.8	6.5	6.3	16.3
n1	DFN	**1.3**	3.4	2.9	5.5	1.3	1.3	1.9
n2	DFN	3.1	8.8	7.4	17.7	2.8	**2.8**	5.4
n3	DFN	3.5	9.0	8.4	17.7	3.4	**3.0**	5.5
n4	DFN	8.7	35.4	30.9	87.7	7.1	**6.6**	22.3

b Number of nodes

Model	BestN	Nodes	BBB	BFB	BLB	DBB	DFB	DLB
bocdp	DFN	84643	33591	32639	33030	32537	**29846**	33341
bocdpf	DFN	183402	41707	38492	40083	29557	**26544**	30230
brp	BFN	72117	52410	**36761**	58825	95439	56786	119826
c1	DFN	18614	19041	17156	27058	15174	**14973**	18292
c2	DFN	57170	51588	44906	71657	40179	**39644**	48069
c3	DFN	76335	57676	50713	81524	47911	**46593**	56833
c4	DFN	737964	378267	339560	502423	327474	**318480**	389018
e1	DFN	23657	26461	24677	37105	20520	**20299**	23931
m1	DFN	3625	4907	4394	13171	2279	**1901**	4970
m2	DFN	15471	18182	16246	44095	5723	**5673**	16603
m3	DFN	16189	18447	18369	49032	9181	**7181**	20291
m4	DFN	61915	69661	66255	157864	20787	**20335**	61606
n1	DFN	3898	5163	4222	13731	2000	**1921**	4579
n2	DFN	15514	18628	15648	49197	6070	**5933**	18315
n3	DFN	16677	18779	17177	48007	7083	**6536**	18031
n4	DFN	69308	71159	63674	160825	21150	**18798**	74430

Table 6.3: Detailed results for diagonal models

a Execution time (s)

Model	BestN	Time	BBB	BFB	BLB	DBB	DFB	DLB
diag 3	BBN	**0.2**	0.3	0.3	–	0.3	0.3	–
diag 4	BBN	**0.6**	0.7	0.7	–	1.0	0.8	–
diag 5	BFN	**1.5**	1.8	1.7	–	4.7	2.0	–
diag 6	BBN	**4.9**	5.8	5.7	–	62.2	6.9	–
diag 7	BBN	**19.3**	21.3	21.4	–	–	27.7	–
diag 8	BBN	**99.2**	108.3	111.8	–	–	153.6	–
split 3	DLN	0.6	0.6	**0.3**	0.4	0.7	0.5	0.4
split 4	DLN	5.3	4.2	**1.0**	1.9	9.0	1.9	2.5
split 5	BLN	259.4	74.6	**3.1**	19.9	–	11.8	45.4
split 6	–	–	–	**11.6**	–	–	–	–
split 7	–	–	–	**58.5**	–	–	–	–
split 8	–	–	–	–	–	–	–	–
opt 3	DLN	**0.2**	0.4	0.3	0.3	0.4	0.4	0.3
opt 4	DLN	**0.8**	1.6	0.9	0.9	2.7	1.2	1.0
opt 5	DLN	4.1	9.9	**2.8**	4.3	79.1	7.8	4.5
opt 6	DLN	39.3	161.5	**10.0**	36.4	–	–	43.9
opt 7	–	–	–	**47.1**	–	–	–	–
opt 8	–	–	–	**293.5**	–	–	–	–

b Number of nodes

Model	BestN	Nodes	BBB	BFB	BLB	DBB	DFB	DLB
diag 3	BFN	193	199	193	–	246	220	–
diag 4	BFN	933	1045	933	–	1800	1262	–
diag 5	BFN	4181	4926	4181	–	17929	5515	–
diag 6	BFN	17815	21685	17815	–	264445	24772	–
diag 7	BFN	73137	90252	73137	–	–	100147	–
diag 8	BFN	291593	360233	291593	–	–	406392	–
split 3	BFN	1929	585	**333**	664	946	492	811
split 4	DFN	31827	8163	**1833**	7144	23459	3847	12527
split 5	BLN	1572515	121370	**9388**	90877	–	27135	207627
split 6	–	–	–	**45566**	–	–	–	–
split 7	–	–	–	**211828**	–	–	–	–
split 8	–	–	–	–	–	–	–	–
opt 3	BFN	619	341	**252**	350	401	372	399
opt 4	BBN	5534	2726	**1330**	2591	5674	2305	3268
opt 5	BLN	51431	24455	**6550**	20987	180464	23529	29124
opt 6	BLN	474498	230929	**30634**	178954	–	–	272734
opt 7	–	–	–	**137788**	–	–	–	–
opt 8	–	–	–	**601970**	–	–	–	–

Chapter 7

K-Induction Based Liveness Checking of Real-Time Systems

The formal proof of correctness of the behavior of safety critical systems is a challenging task as these systems are often fault-tolerant, real-time distributed systems with time-dependent data processing. We faced this problem in checking the correctness of an industrial protocol, the ProSigma SCAN protocol developed by one of our industrial partners, that is responsible for safe transmission of the status of field modules to a control center. We addressed the verification problem by formal modeling and model checking. Our first attempts using several classic modeling formalisms and model checking tools (e.g. timed automata [AD94]) revealed difficulties. First, the use and processing of time-stamps (that was included in the protocol) was either not allowed, or resulted in an infinite state space that could not be handled. Accordingly, we turned towards formalisms that support induction based proofs, and in particular, the technique of k-induction [SSS00; BC00; MRS03; ES03]. However, k-induction based techniques supported only the verification of safety properties (invariants). This way we decided to extend the capabilities of these techniques to support the checking of liveness properties. Second, the formalism that supported k-induction required quite low-level transition systems that were not easy to construct and understand by engineers. Accordingly, we decided to provide a higher-level formalism (so-called calendar systems) that is more easy to use, and can be automatically mapped to the underlying lower level formalism. This formalism proved to be advantageous to find modeling problems by static analysis, and identify invariants that are often required in k-induction based proofs.

In this chapter we introduce the framework that supports these achievements. After briefly describing k-induction in Section 7.1, the new results are presented. The adapted formalism is introduced in Section 7.2. The extensions of k-induction our model checking approach is based on are discussed in Section 7.3. The tool support we provided is summarized in Section 7.4. Finally, we present the validation of our approach by verifying and industrial protocol that motivated our research in Section 7.5. Reference to related work appear in the relevant sections.

7.1 k-Induction

To prove an invariant property P over a transition system S, one typically applies induction over the transition relation.

$$\frac{\begin{array}{lll} s \models P & \text{for all } s \in I & \text{(base case)} \\ s \models P \text{ then } s' \models P & \text{for all } s, s' \in S \text{ with } s \to s' & \text{(ind. hyp.)} \end{array}}{\begin{array}{ll} s \models P & \text{for all } s \in \mathit{Reach}(\mathcal{S}) \end{array}}$$

A more general approach is k-induction, which progresses as follows.

$$\frac{\begin{array}{ll} s_i \models P \text{ for all } 0 \leq i \leq n & \text{for any initial trace } s_0 s_1 \cdots s_n \text{ of length } n < k \\ s_i \models P \text{ for all } 0 \leq i < k \text{ then } s_k \models P & \text{for any trace } s_0 s_1 \cdots s_k \text{ of length } k \end{array}}{\begin{array}{ll} s_i \models P & \text{for all } s_i \in \mathit{Reach}(\mathcal{S}) \end{array}}$$

Given an auxiliary invariant (or lemma) L, one can strengthen the induction hypothesis. In certain cases this enables proving the property by restricting evaluation of the induction step to L-states. The resulting proof scheme is as follows.

$$\frac{\begin{array}{ll} s_i \models P \text{ for all } 0 \leq i \leq n & \text{for any initial trace } s_0 s_1 \cdots s_n \text{ of length } n < k \\ s_i \models P \text{ and } s_i \models L \text{ for all } 0 \leq i < k \text{ then } s_k \models P & \text{for any trace } s_0 s_1 \cdots s_k \text{ of length } k \\ s_i \models L & \text{for all } s_i \in \mathit{Reach}(\mathcal{S}) \end{array}}{\begin{array}{ll} s_i \models P & \text{for all } s_i \in \mathit{Reach}(\mathcal{S}) \end{array}}$$

Naturally, the above method also generalizes to a set $\{L_0, L_1, \ldots, L_n\}$ of lemmas as well.

7.2 Calendar Systems

Inspired by the paper [DS04], we adapted for our purposes the formalism of *calendar automata*, as it supports the modeling of time-dependent behavior, the use of time-stamps, and k-induction based model checking. Calendar automata is a formalism for describing timed systems as transition systems. Its main idea is based on that of discrete event simulation: instead of clocks of timed automata (that store the time elapsed since a past event), it uses variables to store events scheduled to occur at a point of time in the future. Although this way time progresses to infinity, resulting in an infinite state space, the formalism is easy to handle with induction.

Time progress is modeled as follows. A calendar automaton has a set of timeouts that stores local events and an event calendar for messages the automata schedule for each other. A discrete transition may update timeouts to future values or dispatch messages to the calendar, again, scheduled to occur in the future. Such transitions must also consume a current message from the calendar or update a current timeout to prevent instantaneous loops. Time progress transitions are enabled if no current events are available, that is, if the time is lower than any point in time when an event is scheduled to occur. If so, they update time to the time value of the next event. Provided this behavior, the time value of events may never be lower than the current time, and maximal time progress is guaranteed.

To increase model checking performance, we applied two modifications.

- To shorten paths in the state space we adapted the method of merging discrete and time progress transitions introduced in [Pik05]. By doing so, the induction depth needed to verify properties is significantly decreased.
- To eliminate the need for updating momentarily irrelevant timeouts to future values, we modified time progress semantics so that only a valid subset of timeouts is taken into account by determining time value for the next step. This is performed by enabling the possibility for transitions to explicitly validate and invalidate timeouts, thus marking the set of timeouts that are taken into account. This way a great deal of nondeterminism and deadlocks are eliminated, thus improving the performance of the verification.

On top of the modified semantics we developed a higher level formalism, the *calendar system* formalism that makes modeling easier, yet is still suitable for describing a broad range of systems. The next paragraphs describe its syntax and semantics in detail.

Let $\Delta = \{[a,b],(b,c],[b,c),(b,c) \mid 0 \leq a \leq b < c$ and $a,b,c \in \mathbb{N}\}$ and $A^? = A \cup \{\mathsf{none}\}$ and $A^! = A \setminus \{\mathsf{none}\}$. Moreover, for a pair $p = (a,b)$, let $\mathsf{fst}(p) = a$ and $\mathsf{snd}(p) = b$.

Definition 7.1 (Syntax). *A calendar system is a tuple* $(L, T, M, \rightarrow, \ell_0, T_0)$ *where*

- L *is a finite set of locations,*
- T *is a finite set of timeouts,*
- M *is a finite set of messages,*
- $\rightarrow \subseteq L \times \mathit{Event} \times \mathit{Action}_M \times \mathcal{P}(\mathit{Action}_T) \times L$ *is the transition relation, where* $\mathit{Event} = T \cup M$ *is the set of* triggering events, $\mathit{Action}_M = (M \times \Delta)^?$ *is the set of* message sending actions *and* $\mathit{Action}_T = T \times \Delta^?$ *is a set of* timeout setting actions,
- $\ell_0 \in L$ *is the initial location, and finally,*
- $T_0 : T \rightarrow \Delta^?$ *is a function that assigns timeouts their initial value.*

A state of a calendar system is a pair (ℓ, σ) with $\ell \in L$ and σ a function with domain $T \cup \{C, \tau\}$ such that $\sigma(\tau) \in \mathbb{R}_{\geq 0}$ tracks the current time, $\sigma(x) \in \mathbb{R}^?_{\geq 0}$ tracks the current value of a timeout $x \in T$, and with multiset $\sigma(C)$, called the *event calendar*, where for an element $(m,t) \in \sigma(C)$, number $t \in \mathbb{R}_{\geq 0}$ is the point in time message $m \in M$ is scheduled to occur. Initial states are of the form (ℓ_0, σ_0) where $\sigma_0(\tau) = 0$ and $\sigma_0(C) = \emptyset$ and for all $x \in T$ we have $\sigma_0(x) \in T_0(x)$ if $T_0(x) \in \Delta$ and $\sigma_0(x) = \mathsf{none}$ otherwise. Moreover, for each transition $\ell \xrightarrow{e,\mu,S} \ell'$ of the calendar system, there is a transition $(\ell, \sigma) \xrightarrow{e} (\ell', \sigma')$ in the transition system defining its semantics such that $\sigma'(\tau) = \min(\sigma(T)^! \cup (\mathsf{snd} \circ \sigma)(C))$, and the following conditions hold.

- For all $x \in T$, exactly one of the following rules applies for the next value of timeout x.

$$\frac{(x,\delta) \in S \qquad \delta = \mathsf{none}}{\sigma'(x) = \mathsf{none}} \ \text{invalidate } x$$

$$\frac{(x,\delta) \in S \qquad \delta \in \Delta \qquad d \in \delta}{\sigma'(x) = \sigma(x) + d} \ \text{set } x$$

$$\frac{\forall \delta.(x,\delta) \notin S}{\sigma'(x) = \sigma(x)} \ \text{skip } x$$

- Exactly one of the following rules applies for the next value of the calendar C.

$$\frac{e \in T \qquad \sigma(e) = \sigma(\tau) \qquad \mu = \mathsf{none}}{\sigma'(C) = \sigma(C)} \ e \text{ over / send none}$$

$$\frac{e \in T \qquad \sigma(e) = \sigma(\tau) \qquad \mu = (m,\delta) \qquad d \in \delta}{\sigma'(C) = \sigma(C) \cup \{(m, \sigma(t) + d)\}} \ e \text{ over / send } m$$

$$\frac{e \in M \qquad (e, \sigma(t)) \in \sigma(C) \qquad \mu = \mathsf{none}}{\sigma'(C) = \sigma(C) \setminus \{(e, \sigma(t))\}} \ e \text{ received / send none}$$

$$\frac{e \in M \qquad (e, \sigma(t)) \in \sigma(C) \qquad \mu = (m, \delta) \qquad d \in \delta}{\sigma'(C) = \sigma(C) \setminus \{(e, \sigma(t))\} \cup \{(m, \sigma(t) + d)\}} \; e \text{ received / send } m$$

For modeling purposes, it is convenient to describe systems compositionally. For that we also defined the composition of calendar systems, which is the interleaving of two systems.

7.3 Model Checking of Calendar Systems

A useful structural feature of calendar automata is that time never exceeds any time value of scheduled events [DS04]. Our formalism preserves this property with respect to values of currently valid timeouts and calendar events. Other invariants like the minimal and maximal value of events relative to time at a given control location or possible elements of the set of valid timeouts at a given control location can be automatically determined by processing a graph induced by the calendar system. In the following we present our achievements in the verification of calendar systems.

7.3.1 Finding Counterexamples for ω-Regular Properties

As described before, model checking of an ω-regular property can be solved by searching for lassos in the product system of the original system and the automaton representing the negated property. Since calendar systems have a dense-time semantics with a monotonically increasing time variable (thus resulting in a continuous, infinite state space), in order to find lassos in the semantics, one needs a suitable bisimulation over states of the product system. Our solution was to partition states by the time value of their scheduled events relative to current time. Formally, two states (ℓ_1, σ_1) and (ℓ_2, σ_2) are considered equivalent iff $\ell_1 = \ell_2$ and

- for all timeouts $x \in T$, we have $\sigma_1(x) = \text{none}$ iff $\sigma_2(x) = \text{none}$
- for all timeouts $x \in T$, if $\sigma_1(x) \neq \text{none}$ and $\sigma_2(x) \neq \text{none}$, then $\sigma_1(x) - \sigma_1(\tau) = \sigma_2(x) - \sigma_2(\tau)$
- there exists a bijection $\pi : \sigma_1(C) \to \sigma_2(C)$ such that for all $c \in \sigma_1(C)$ with $c = (m_1, t_1)$ and $\pi(c) = (m_2, t_2)$, we have $m_1 = m_2$ and $t_1 - \sigma_1(\tau) = t_2 - \sigma_2(\tau)$

Although the quotient state space that can be produced with this bisimulation is still not finite, it contains lasso-shaped runs that can be recognized on the fly. This can be done by a synchronous observer of the system that nondeterministically saves the current state and compares each following state to that saved state [BAS02; SB06]. If the two are equal with regard to the bisimulation relation, they are the intersection of an (abstract) lasso-shaped run.

However, this bisimulation is not necessarily coarse enough to find each such trace of the calendar system, so the method is only capable of finding counterexamples. The problem is a manifestation of the one presented in [KJN12a] for timed automata, and a witness for this statement, as depicted in Figure 7.1, can be constructed analogously to the example presented there.

7.3.2 Proving ω-Regular Properties Using k-Induction

Proving ω-regular properties (including liveness properties) of calendar systems with k-induction can also be attempted by constructing the product system. To prove that the number of accepting states occurring in every run of the product system is finite, one can try to find an upper bound l for the number of accepting states of a run. If such number exists, the property must hold. This method, known as k-liveness, is complete for finite systems: if the property holds then there is an upper bound [CS12].

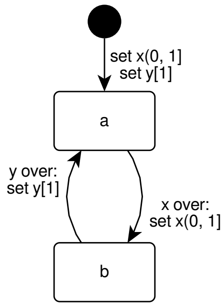


Figure 7.1: A calendar system with no simple loop

Suppose the capacity of the calendar is restricted to some finite number. Although the semantics of such a calendar system is not finite, there exists a finite system that is bisimilar to it. This statement can be proven by giving such a bisimulation relation. Since calendar systems are very similar to timed automata (as by scheduling events only intervals bounded by natural numbers are allowed), region equivalence [AD94] can be applied for this purpose. The only considerable difference is that instead of clock values the time value of events relative to current time would be taken into account, and a clock in the unbounded clock region would correspond to an invalid timeout. Formally, two states (ℓ_1, σ_1) and (ℓ_2, σ_2) are considered equivalent iff $\ell_1 = \ell_2$ and

- for all timeouts $x \in T$, we have $\sigma_1(x) = \mathsf{none}$ iff $\sigma_2(x) = \mathsf{none}$
- for all timeouts $x, y \in T$ such that $\sigma_1(x) \neq \mathsf{none}$ and $\sigma_1(y) \neq \mathsf{none}$ and $\sigma_2(x) \neq \mathsf{none}$ and $\sigma_2(y) \neq \mathsf{none}$, we have
  - $\lfloor \sigma_1(x) - \sigma_1(\tau) \rfloor = \lfloor \sigma_2(x) - \sigma_2(\tau) \rfloor$
  - $\{\sigma_1(x) - \sigma_1(\tau)\} = 0$ iff $\{\sigma_2(x) - \sigma_2(\tau)\} = 0$
  - $\{\sigma_1(x) - \sigma_1(\tau)\} \leq \{\sigma_1(y) - \sigma_1(\tau)\}$ iff $\{\sigma_2(x) - \sigma_2(\tau)\} \leq \{\sigma_2(y) - \sigma_2(\tau)\}$
- there exists a bijection $\pi : \sigma_1(C) \to \sigma_2(C)$ such that for all $c, c' \in \sigma_1(C)$ with $c = (m_1, t_1)$ and $c' = (m_1', t_1')$ and $\pi(c) = (m_2, t_2)$ and $\pi(c') = (m_2', t_2')$, we have
  - $m_1 = m_2$
  - $\lfloor t_1 - \sigma_1(\tau) \rfloor = \lfloor t_2 - \sigma_2(\tau) \rfloor$
  - $\{t_1 - \sigma_1(\tau)\} = 0$ iff $\{t_2 - \sigma_2(\tau)\} = 0$
  - $\{t_1 - \sigma_1(\tau)\} \leq \{t_1' - \sigma_1(\tau)\}$ iff $\{t_2 - \sigma_2(\tau)\} \leq \{t_2' - \sigma_2(\tau)\}$
- moreover, for all $x \in T$ such that $\sigma_1(x) \neq \mathsf{none}$ and $\sigma_2(x) \neq \mathsf{none}$ and $c \in \sigma_1(C)$ such that $c = (m_1, t_1)$ and $\pi(c) = (m_2, t_2)$, we have
  - $\{t_1 - \sigma_1(\tau)\} \leq \{\sigma_1(x) - \sigma_1(\tau)\}$ iff $\{t_2 - \sigma_2(\tau)\} \leq \{\sigma_2(x) - \sigma_2(\tau)\}$
  - $\{\sigma_1(x) - \sigma_1(\tau)\} \leq \{t_1 - \sigma_1(\tau)\}$ iff $\{\sigma_2(x) - \sigma_2(\tau)\} \leq \{t_2 - \sigma_2(\tau)\}$

As a consequence, under the above assumption, for a calendar system for that a ω-regular property holds, there exists a suitable upper bound l, namely any upper bound of its finite counterpart. As conclusion, our method can be considered complete just like in the finite case. (Naturally, as usual for k-induction, verification might require additional lemmas to succeed.)

The existence of such an upper bound can easily be stated as an invariant property over a modified system: one must expand the system with a synchronous observer that counts the accepting states during the run. The property is then that the value of this counter is not greater than the upper bound. The formulated invariant property then can be checked with k-induction.

For successful verification, in our framework we support the model checker with the following settings:

- We add a supporting lemma that the value of this counter is positive (otherwise counterexamples to induction of arbitrary length could be constructed, starting from an adequately small negative counter value).
- By a straightforward interval analysis of the calendar system model, we provide simple invariants that describe the possible minimal and maximal values for timeouts at given locations of the system, and for the dispatch time of messages, this way sorting out a significant number of unreachable states.
- We set the induction depth k to be at least equal to the upper bound l, or else no counterexample during the base case can be found, since the length of paths would be too small for the number of accepting states to exceed the bound. Moreover, if the bound is greater than the induction depth, then no path in the state space will contradict the lemma over the counter values during the induction step, serving as a possible counterexample (if not sorted out by other lemmas), thus enforcing the increasing of the induction depth.

7.4 Tool Support

For efficient verification of calendar systems, we developed a toolchain that supports the aforementioned modeling and verification steps. Our implementation is based on the Eclipse Modeling Framework (EMF) and related technologies. It includes a domain specific language (DSL) that enables the modular description of calendar systems and the formulation of their requirements. Its metamodel, shown in Figure 7.2, is constructed in EMF and is augmented with a graphical concrete syntax that enables marking control locations and transitions between them, labeled with events (receiving a message or that a timeout is over) and actions (sending a message or setting a timeout). The static analysis of models focused on recognizing possible design flaws like incomplete or nondeterministic transition description or unreachable control locations. To support k-induction verification we implemented the means for deriving the kind of invariants described in Section 7.3. Invariants are detected as fix-points of recursive graph patterns that can be matched over the models using the incremental graph pattern matcher EMF IncQuery [Ber+10].

For model checking, we implemented a code generator that automatically provides the mapping to the lower level artifacts that are used for model checking in the SAL environment [MOS03]:

- The modular description of a transition system that corresponds to the formal semantics of the calendar system given in the instance model.
- The description of Büchi automata belonging to the requirements, that can be synchronously composed with the system to provide the product system.
- The tools for finding counterexamples: an observer for the bisimulation and an observer for finding loops.
- The tools for proving properties: the counter module for proving properties and the derived invariant properties.

7.5 Case Study

Using the methods and tool described here, we managed to formally verify liveness properties of a communication protocol from an industrial SCADA system. During our work, we examined the part of the protocol that establishes connections between modules and transmission of their states. We created a model of the fault-free system as a product of two calendar systems that represent the two participants – the so-called field and control sides – that attempt to build a connection. The

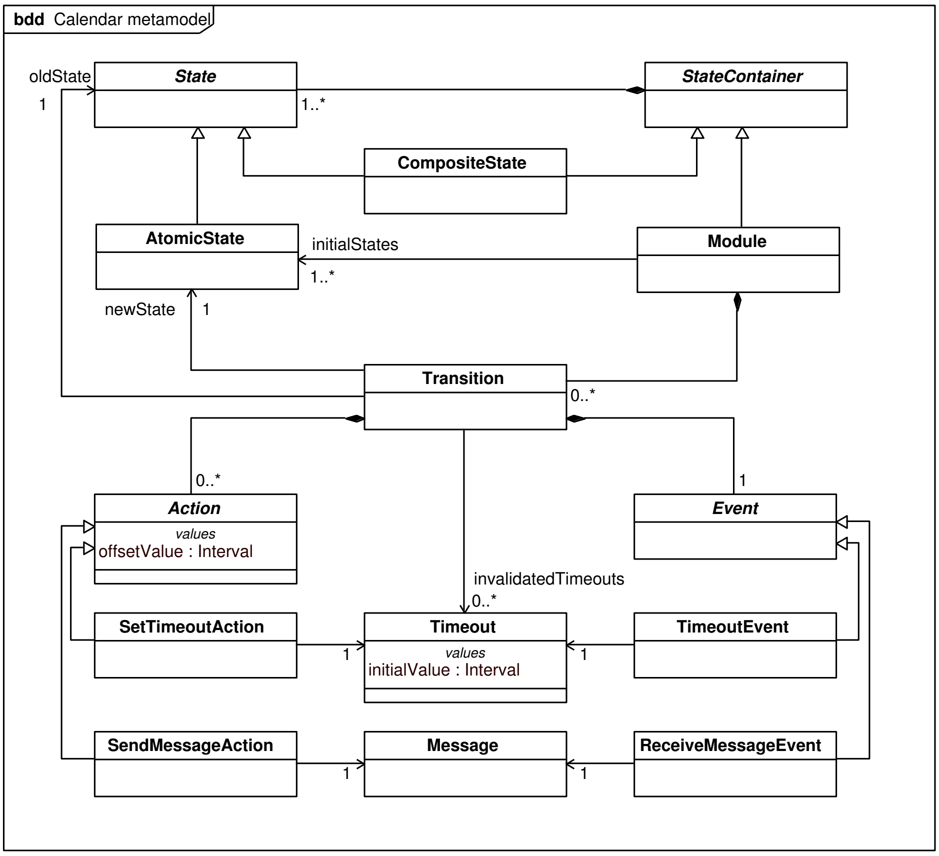


Figure 7.2: EMF metamodel of the calendar system DSL

models are presented in a graphical syntax in Figure 7.3. We fixed the timing parameters at value $\{TPropMin \hookleftarrow 0, TPropMax \hookleftarrow 1, TSync \hookleftarrow 3, TRtMax \hookleftarrow 6\}$.

The formal model of the system enables the formal specification of requirements. We are going to consider the protocol correct if eventually both sides of the connection reach state *Connected*, and stay in that state for the future. This can be formalized in LTL as $\varphi = \mathsf{FG}c$, where $c = FieldLG.Connected \land ControlLG.Connected$ is the proposition expressing that the system is connected. The negation of this formula is $\neg\varphi = \mathsf{GF}\neg c$, for which the corresponding Büchi automaton is depicted in Figure 7.4.

7.5.1 Discovering Invariants

As mentioned earlier, auxiliary invariants are often crucial for successful k-induction. From the calendar system model of the modules in the protocol, our tooling automatically extracted the invariants summarized in Table 7.1. Besides the invariant conditions, the table contains the required induction depth and the time required to prove the property. The invariants have been proved relative to the following lemmas that are invariant for any calendar system.

- $0 \leq \tau$
- $x \neq \mathsf{none} \rightarrow \tau \leq x$ for all $x \in T$
- $\tau \leq t$ for all $(m, t) \in C$

Relative to these lemmas, each invariant is inductive, that is, $k = 1$. Additionally, we include the invariant $q_0 \rightarrow c$ for the Büchi automaton.

Table 7.1: Automatically extracted invariants of the calendar system model

Invariant	k	Time (s)
$FieldLG.Reset \rightarrow FieldLG.ToReset = \text{none}$	1	< 1
$FieldLG.Connecting \lor FieldLG.Connected \rightarrow$ $0 \leq FieldLG.ToReset - \tau \leq TRtMax$	1	< 1
$0 \leq FieldLG.ToSync - \tau \leq TSync$	1	< 1
$ControlLG.Reset \rightarrow ControlLG.ToReset = \text{none}$	1	< 1
$ControlLG.Connecting \lor ControlLG.Connected \rightarrow$ $0 \leq ControlLG.ToReset - \tau \leq TRtMax$	1	< 1

7.5.2 Proving Correctness using Abstraction

Using the abstraction technique described in [DS04], we proved further lemmas over the system. Using this technique, it was possible to provide lemmas for the proof that are not invariant properties over the original system. This can be achieved by extending the system with monitor components that prescribe that whenever some given proposition Φ_i holds in the current state, then some proposition Ψ_i is to hold in the next state. Semantically, each such monitor is a finite state machine for the regular safety property $\mathsf{G}(\Phi_i \rightarrow \mathsf{X}\Psi_i)$, over which we can simply formulate an invariant that the property holds. Naturally, this idea generalizes to any regular safety property, and even to general ω-regular properties if we use Büchi automata as monitors and the k-liveness method for counting occurrences of accepting states.

To prove the system correct, we defined the abstraction depicted in Figure 7.5. Each state of the abstraction model induces a lemma, as summarized in Table 7.2, that can be proved using the method described in Section 7.3.2. As any such lemma is a regular safety property, the Büchi automaton for its negation can be chosen so that it effectively encodes a minimal deterministic finite automaton that recognizes bad prefixes. In this case, the upper bound l can be chosen to 0. By proving the abstraction properties one by one and using them as lemmas, the property $\varphi = \mathsf{FG}c$ can be easily proved.

7.5.3 Extending the System with an Error Model

As the modeled system operates in a safety critical environment, it is necessary to evaluate its correctness under fault assumptions. Thus we extended the model of the system with a simple fault model, shown in Figure 7.6, that admits the loss of a single message.

The analysis then revealed the counterexample loop depicted in Figure 7.7. The counterexample shows that in the model, given a certain ordering of events, even the loss of a single message can cause the modules to get stuck in an unconnected state, and prevent the connection to be established. To make the analysis more efficient and the counterexample easier to comprehend, we performed the bounded model checking on a discrete time model. The result is summarized in the first row of Table 7.3.

Table 7.2: Properties describing an abstraction model

Property	k	l	Time (s)
$\mathsf{G}(A_{11} \to \mathsf{X}A_{21})$	18	0	8.93
$\mathsf{G}(A_{21} \to \mathsf{X}A_{22})$	18	0	5.66
$\mathsf{G}(A_{22} \to \mathsf{X}A_{32})$	19	0	4.28
$\mathsf{G}(A_{32} \to \mathsf{X}(A_{32} \vee c))$	6	0	1.26
$\mathsf{G}(c \to \mathsf{X}c)$	7	0	1.39
$\mathsf{G}(\neg A_{12})$	16	0	3.02
$\mathsf{G}(\neg A_{13})$	8	0	1.48
$\mathsf{G}(\neg A_{23})$	9	0	1.58
$\mathsf{G}(\neg A_{31})$	6	0	1.25
$\mathsf{FG}c$	6	4	1.24

To try to fix this problem, we extended the model of $FieldLG$ so that it responds to a received OBJ1 with OBJ2 in state $Connecting$. This modification eliminated the counterexample found earlier.

The proof of the system was then elaborated as follows. Let $f = FaultModel.One\_left$. As under the assumption $\mathsf{G}f$, the newly added transition never fires, and thus the earlier correctness result applies, it is sufficient to prove the property $\mathsf{G}(\neg f \to \neg c \to \mathsf{FG}c)$. The Büchi automaton corresponding to the negation of this formula is depicted in Figure 7.8. To enable verification, we provided the invariant that the counter for l has its initial value iff the Büchi automaton is in state q_0. We then successfully proved the property, with the result summarized in the second row of Table 7.3.

This result can be further generalized. We can show that the system tolerates any finite number of message losses by proving that the model (without the fault model) satisfies the property starting from *any* state as initial state. (A similar approach is presented in Chapter 8.) To prove this, we modified the model and the generated SAL code by removing any constraints on the set of initial states. The result of the analysis is shown in the third row of Table 7.3

Table 7.3: Results of the analysis

Property	k	l	Time (s)
$\mathsf{FG}c$ (counterexample)	18	–	4.93
$\mathsf{G}(\neg f \to \neg c \to \mathsf{FG}c)$	32	29	49.36
$\mathsf{FG}c$ (from any state, without fault model)	32	29	78.47

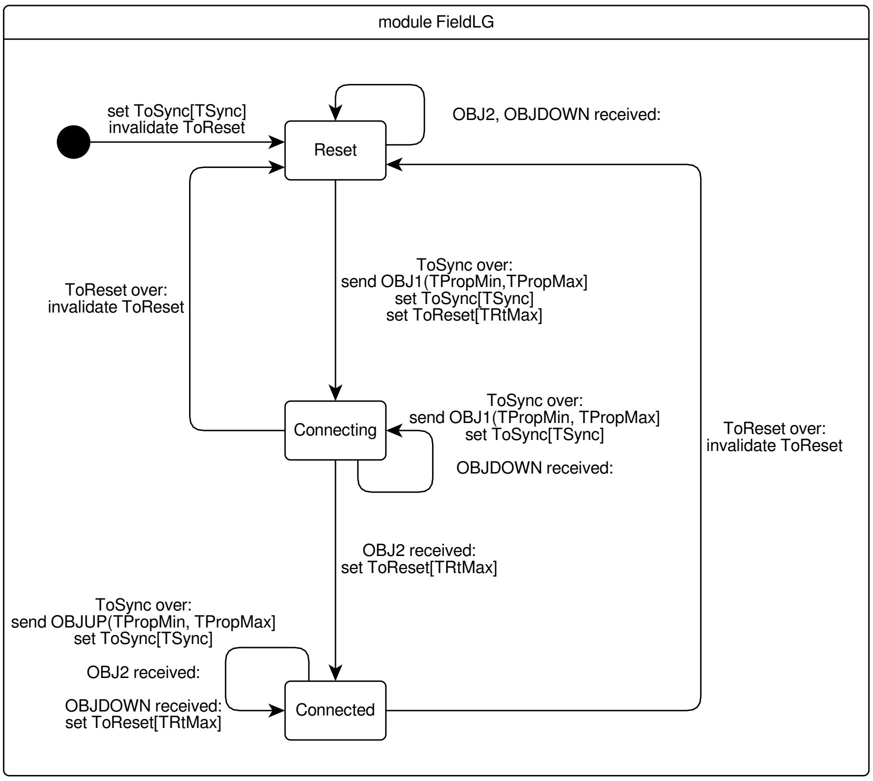
7.6 Conclusions

In this chapter, we proposed (1) the extension of calendar automata to provide the calendar system formalism that allows convenient modeling of the core protocols of communicating real-time systems, (2) the extension of k-induction based techniques to support the verification of both safety and liveness properties of calendar systems, and (3) the tool support to perform static analysis, derivation of invariants and artifacts required for k-induction based automated verification. The framework proved to be useful to find problems in industrial protocols.

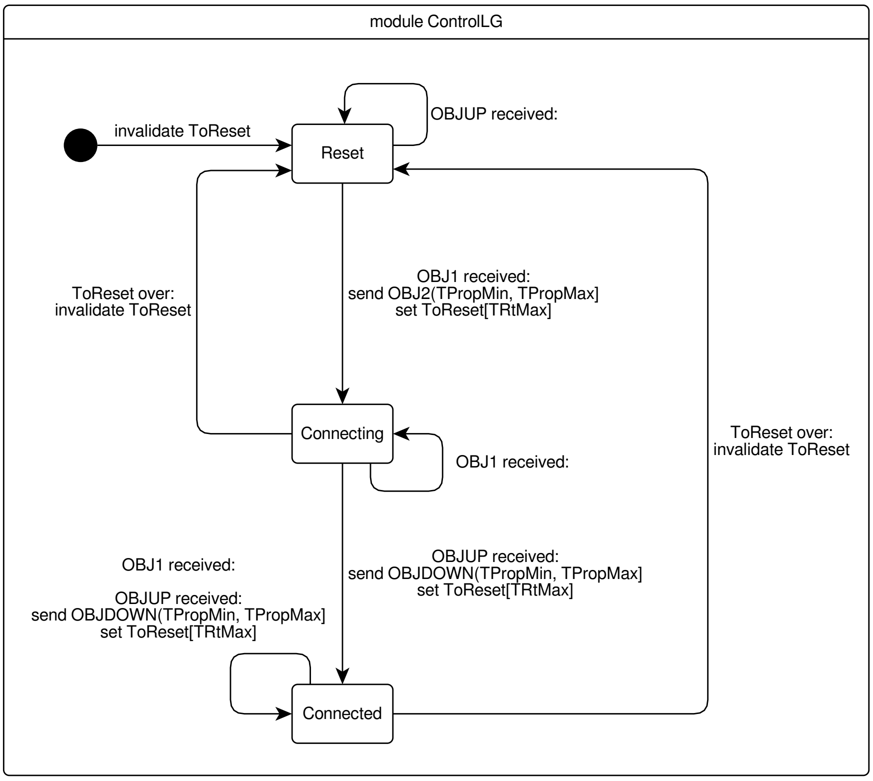
7.6.1 Thesis Summary

This concludes Thesis 4.1 of this dissertation. We summarize it as follows.

> **Thesis 4.1** *K-induction based liveness checking of real-time systems.* I proposed the calendar system formalism that allows convenient modeling of the core protocols of communicating real-time systems. By a series of transformation steps, I extended k-induction based model checking to support the verification of both safety and liveness properties of calendar systems. Moreover, I provided a tool-supported solution for the derivation of lemmas required for successful k-induction based automated verification.



(a) Field LG



(b) Control LG

Figure 7.3: Calendar system models of the protocol

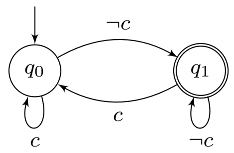


Figure 7.4: Büchi automaton for $\mathsf{GF}\neg c$

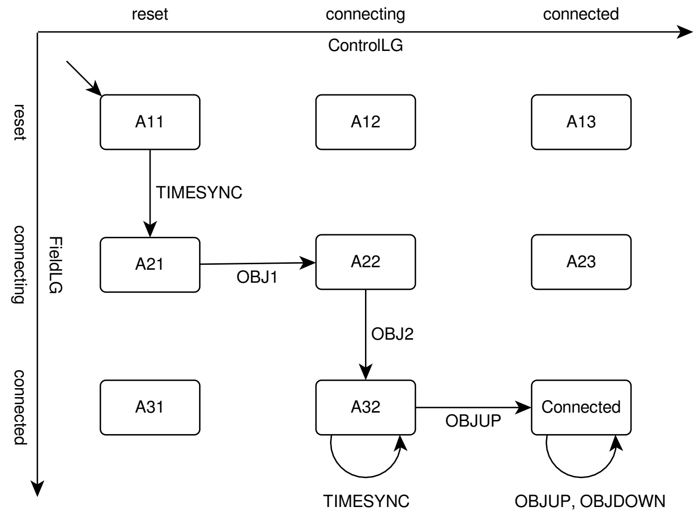


Figure 7.5: Abstraction model for proving correctness

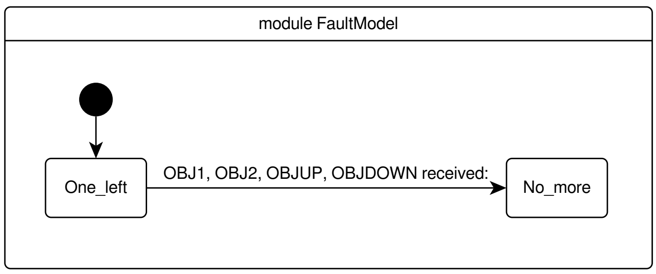


Figure 7.6: Fault model

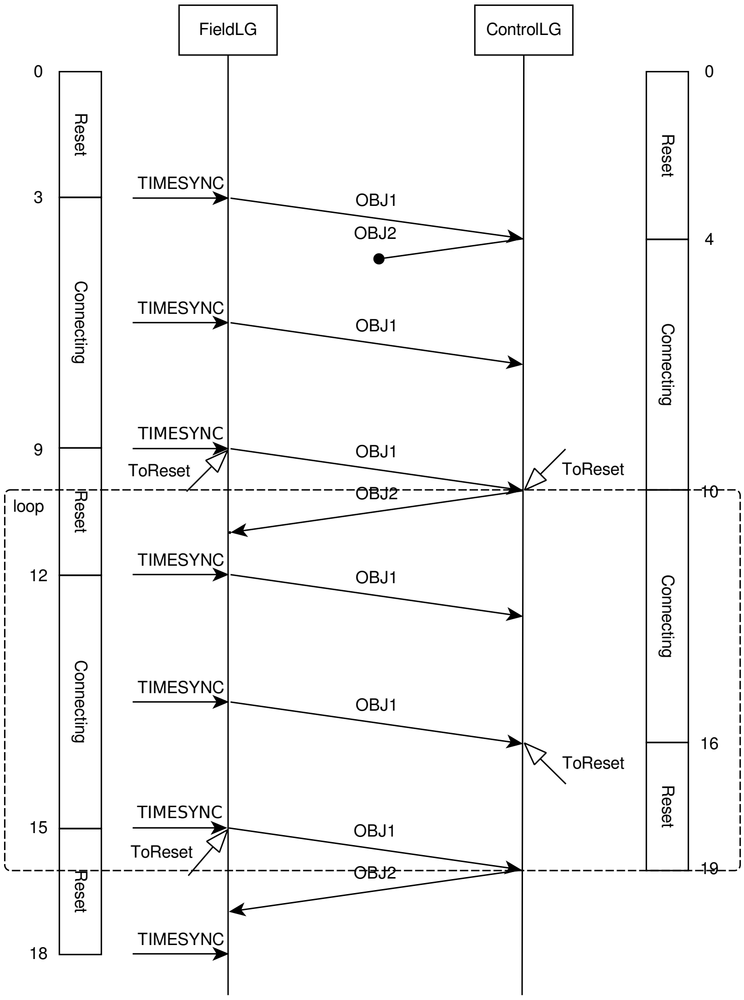


Figure 7.7: Counterexample for the property

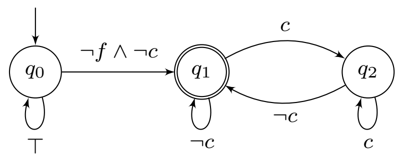


Figure 7.8: Büchi automaton for $\neg \mathsf{G}(\neg f \rightarrow \neg c \rightarrow \mathsf{FG}c)$

Chapter 8

A Decomposition Method for Liveness Checking of Hierarchical Real-Time Protocols

Even for models of simple safety critical systems, model checking might be intractable due to the inherent distributed and timed characteristic of such systems. In particular, the verification of distributed systems often leads to the well-known phenomena of state space explosion which is a major obstacle for successful model checking. Real-time systems require methods being able to handle timed behaviors expressed with real-valued clock variables and their relations, further increasing the complexity of verification. Due to the above mentioned reasons, model checking techniques are often unable to verify complex real-time systems in a fully automatic manner. Decomposition can serve as a solution: safety critical systems, especially protocols used in such systems, are mainly composed hierarchically, where different layers of functions rely on each other. Experts can exploit this layered structure to decompose the verification problem to smaller and tractable ones. In addition, the specified properties in real-life systems are typically complex in the sense that they are usually combinations of reachability and liveness queries. On the basis of the expected behavior of the system and the structure of the property specification, experts can decompose the specification and give simpler verification problems to the model checker.

In this chapter, this decomposition approach is presented formally and demonstrated by the verification of a distributed safety critical protocol, whose main functionality is to guarantee reliable communication between components in a distributed SCADA (Supervisory Control and Data Acquisition) system. The protocol is hierarchically layered in the sense that it implements two functionalities: master election and the allocation of communication identifiers, where the latter functionality is based on the former one, i.e. performed by an elected master. The requirement for the protocol is to provide this functionality even after the occurrence of a finite number of transient faults. This requirement is formalized in linear temporal logic and a decomposition scheme is introduced in order to make verification feasible. The main goal of our work is to show how the structure of the system and the specification can be exploited to provide efficient verification. This decomposition approach is a generic scheme that can be followed in similar systems where the functions can be decomposed and a similar combination of reachability and liveness properties shall be verified.

8.1 Verification Approach

In general, the verification process of a fault tolerant system consists of many modeling and model checking steps. First, the system has to be verified leaving any fault assumptions out of consideration, thus the formal model of the fault-free system has to be developed. After the successful verification of the fault-free system, to verify fault tolerance, possible faults and their effects on the system have to be taken into account. Hence fault models have to be defined, that composed with the model of the fault-free system represent the behavior of the system under the given fault assumptions.

Since the verification of all possible faults and their combinations is often infeasible, at this point the verification engineer may restrict the range of investigated faults to selected ones. However, omitting any relevant fault or combination of faults can lead to verification results that cannot be justified with respect to the behavior of the real system. In this section we introduce a different approach, which is based on the following assumptions and restrictions:

- We assume that the system under consideration is a distributed protocol with a layered hierarchy of services, where correctness of higher level functions is based on the correctness of lower level functions. Our goal is to check the correctness of such systems under the occurrence of finitely many faults.
- Permanent and crash faults are not modeled since the focus is the verification of resilience, i.e. resuming the correct behavior of the system after transient faults. Permanent and crash faults are easier to detect than transient faults and need redundancy to provide fault tolerance.
- The effects of transient faults are modeled on a logical level as disturbances in the behavior of the related components in the form of additional transitions (called fault transitions) between states of the fault-free model. With regard to the common fault classification (crash, omission, timing, computation and Byzantine faults) we have the following considerations. Crash faults are not modeled as mentioned above. Omissions are covered by fault transitions that step over the omitted processing steps (including message sending or message processing). The effects of delayed messages and corrupt messages are covered by the combination of fault transitions that cause the loss of the original message and creation of a faulty one. Similarly, data corruption is covered by fault transitions that alter the state variables. Control flow errors among states, including the restart of the component, are also covered by fault transitions. Regarding Byzantine faults, those faults are covered whose effects can be modeled in terms of transitions between the states of the fault-free model.
- The resilience of the system is expressed as a persistence property: the effects caused by a transient fault shall be tolerated in such a way that after the occurrence of a fault (and the related disturbance), the behavior will eventually resume the correct one (this way almost all states along a path will belong to a correct behavior).

As presented in the following sections, the second assumption allows a systematic verification of faults, without requiring separate (manual) modeling of each fault. The third assumption enables in certain cases the use of a decomposition approach that divides the verification task into smaller and simpler ones.

In the following the used notations are introduced then the proof strategy for the efficient verification of fault models is detailed. Finally, the decomposition of persistence properties into simpler properties is given.

8.1.1 Notation

We introduce the following notations for two different restrictions of a transition system with respect to a propositional formula. Let $\mathcal{S} = (S, A, T, I)$. Then $\mathcal{S}_\varphi = (S, A, T, S\restriction_\varphi)$ and $\mathcal{S}^\varphi = (S\restriction_\varphi, A, T\restriction_\varphi, S\restriction_\varphi)$. Here, we define $S\restriction_\varphi = \{s \in S \mid s \models \varphi\}$ and $T\restriction_\varphi = T \cap (S\restriction_\varphi \times S\restriction_\varphi)$. For example, $\mathcal{S}_\top = (S, A, T, S)$, that is, $\mathcal{S}$ with all states considered as potential initial states. It is easy to see that $(\mathcal{S}^\psi)_\varphi = (\mathcal{S}_\varphi)^\psi$, thus in this case the brackets can be omitted. Moreover, $(\mathcal{S}^\varphi)^\psi = \mathcal{S}^{\varphi\wedge\psi} = \mathcal{S}^{\psi\wedge\varphi} = (\mathcal{S}^\psi)^\varphi$ and $(\mathcal{S}_\varphi)_\psi = \mathcal{S}_\psi$.

8.1.2 Modeling Transient Faults

A transient fault of a system is considered to change the state of a component from one state to an other. Such a fault is for example the restart of a component (which brings the component to an initial state) or the loss of a message in the channel. In the following the concept of a transient fault is formalized and we show how this formalization can be exploited during formal verification.

Let $\mathcal{S} = (S, A, T, I)$ be a transition system. We model a fault in $\mathcal{S}$ as a set of transitions $F \subseteq S \times A' \times S$, where a fault transition $(s, \alpha, s') \in F$ models the effects of the occurrence of the fault in state s. In other words, we consider transient faults that can be expressed in terms of a nondeterministic change of state in the fault-free system. Naturally, the range of faults that can be modeled this way depends on the formulation of the system.

Given $\mathcal{S}$ and F, we can define a transition system $\mathcal{S}_F$ that models the system with a finite number of possible occurrences of transient fault(s) F as $\mathcal{S}_F = (S_F, A_F, T_F, I_F)$ where

- $S_F = S \times \mathbb{N}$. Given a state (s, n), number n is the number of transient faults that can still occur in the system.
- $A_F = A \cup A'$.
- $I_F = I \times \mathbb{N}$. Initially, any finite number of faults are allowed to occur.
- T_F is the smallest relation defined by the following rules:

$$\frac{(s, \alpha, s') \in T \qquad n \in \mathbb{N}}{(s, n) \xrightarrow{\alpha} (s', n)} \text{ normal transition}$$

$$\frac{(s, \alpha, s') \in F \qquad n \in \mathbb{N}}{(s, n+1) \xrightarrow{\alpha} (s', n)} \text{ fault transition}$$

To verify that a system $\mathcal{S}$ satisfies a persistence property $\mathsf{FG}\varphi$ even if a transient fault defined by F can occur finitely many times, the following direct approach can be applied:

1. Construct $\mathcal{S}_F$ from $\mathcal{S}$ and F.
2. Check $\mathcal{S}_F \models \mathsf{FG}\varphi$.

However, the fact that the system $\mathcal{S}_F$ satisfies a persistence property $\mathsf{FG}\varphi$ often originates from the stronger property that $\mathcal{S}$ stabilizes to φ-states starting from *any of its states*. Using the above notation, this can be expressed by the following rule.

$$\frac{\mathcal{S}_\top \models \mathsf{FG}\varphi}{\mathcal{S}_F \models \mathsf{FG}\varphi} \text{ fault abstraction}$$

It is easy to see that this approach is sound, that is, if the antecedent hold, then the consequent also holds.

Proof. We prove the stronger property that $\tau \models \mathsf{FG}\varphi$ for all $\tau \in \mathit{Traces}(\mathcal{S}_F)$. Assume $\mathcal{S}_\top \models \mathsf{FG}\varphi$ and let $\tau = (s_0, n_0)(s_1, n_1)(s_2, n_2)\ldots$ be an trace of $\mathcal{S}_F$. We apply induction on n_0. If $n_0 = 0$, then τ is an initial trace of $\mathcal{S}_\top$, thus the statement holds. Now assume $n_0 > 0$. If for all $i > 0$ we have $(s_{i-1}, \alpha_i, s_i) \in T$ for some $\alpha_i \in A$, the same applies as in the base case. So assume there is a state (s_{i-1}, n_{i-1}) with a minimal i such that $(s_{i-1}, \alpha_i, s_i) \in F$ for some $\alpha_i \in A'$. Since $n_i < n_{i-1}$, by the induction hypothesis, $\tau^i \models \mathsf{FG}\varphi$, thus $\tau \models \mathsf{FG}\varphi$. □

Since the rule is sound for any F, it allows the verification of fault tolerance without the need of explicitly modeling faults.

8.1.3 Decomposition of Persistence Properties

The resilience of the system is expressed as a persistence property $\mathsf{FG}\varphi$. The verification of such properties is a complex task as the model checker has to handle all traces and check if they contain fair cycles (with fairness constraint $\neg\varphi$) as counterexamples. In the following, we describe two rules that in certain cases – in our case, the layered structure of protocol functionalities – enable the simplification of the model checking problem of such properties. We omit soundness proofs due to their simplicity.

The first rule describes the decomposition of a persistence property according to the expected behavior of the system. Without loss of generality, we can assume that the persistence condition is of the form $\varphi \wedge \psi$. Here, both φ and ψ define some configuration of the system that is expected to eventually persist. If the persistence of the system with respect to ψ depends on its persistence with respect to φ, the following rule can be applied to simplify the model checking problem.

$$\frac{\mathcal{S} \models \mathsf{FG}\varphi \qquad \mathcal{S}^\varphi \models \mathsf{FG}\psi}{\mathcal{S} \models \mathsf{FG}(\varphi \wedge \psi)}\ \text{FG-detachment}$$

Here, all states of $\mathcal{S}^\varphi$ are φ-states. The main advantage of such a decomposition is that if φ and ψ refer to different variables of the system, then the subproblems can be simplified significantly by abstractions that depend on the property, such as cone of influence reduction [CGP99].

The second rule divides the model checking problem into two simpler problems.

$$\frac{\mathcal{S} \models \mathsf{F}\varphi \qquad \mathcal{S}_\varphi \models \mathsf{G}\varphi}{\mathcal{S} \models \mathsf{FG}\varphi}\ \text{G-detachment}$$

Here, the check of $\mathcal{S} \models \mathsf{F}\varphi$ is a query searching for a lasso shaped initial path of $(\neg\varphi)$-states (as counterexample). The check $\mathcal{S}_\varphi \models \mathsf{G}\varphi$ basically amounts to verify whether φ is inductive, which is a less expensive step.

8.2 Description of the Protocol

In this section, as the context and motivation of our work, the protocol and specified properties are introduced in details. The main purpose of the protocol is to ensure stable and fault tolerant communication between components of a distributed SCADA system. In the protocol, communication is performed in two layers: the lower layer serves for administration, while the upper layer transmits information between the components.

There are two types of components in the system: at most four communication units, called ETHs, and at most ten input-output units, called LIOs, that are connected via a CAN bus that serves as the

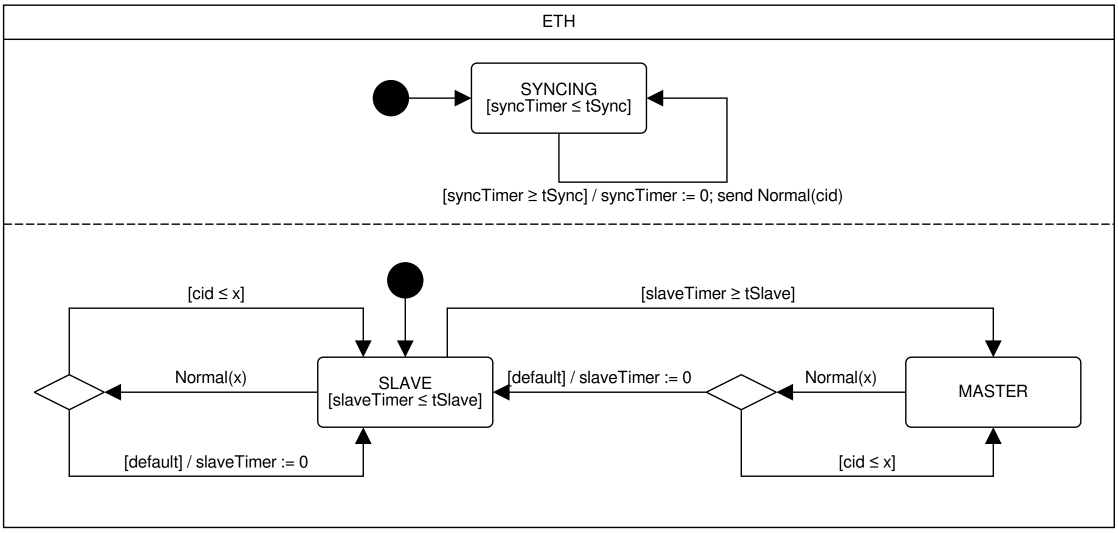


Figure 8.1: Master election

communication channel. Each component has a 29 bit physical address called *hwid* that is used in administrative messages to identify a specific component on the bus. However, components also get assigned a 4 bit logical address called *cid* that is used in the higher level communication protocols instead of *hwid* to save bandwidth. The *cid*s of *ETH*s are assigned statically from the range $[0 \ldots 3]$, while *LIO*s obtain their *cid* values dynamically from the range $[4 \ldots 13]$ from a distinguished *ETH* that is an elected master. *cid* values 14 and 15 are reserved for addressing multicast and broadcast messages, respectively.

The functionalities of the protocol can be summarized as follows:

- *Master election.* From the *ETH*s that communicate on the bus, the one with the lowest *cid* value must be elected as master.
- *Assignment of logical addresses.* The master *ETH* must ensure that all *LIO*s have a unique *cid*.

Since the system is used in a critical context, it must provide the above functionalities even in the presence of a finite number of predefined faults. Accordingly, the verification must be aimed at the checking of the correct functionality of the protocol in a fault-free case and also in the presence of these faults. As the protocol was designed using SysML models (with time extensions), we will refer to the relevant statechart models to present the operation of the protocol. These statecharts were used to derive the formal models that were the basis of verification using our fault modeling and decomposition approach.

8.2.1 Master Election

To ensure that *LIO*s obtain unique logical addresses, *cid*s can only be assigned by a distinguished *ETH* called master. The purpose of master election is to ensure that during the operation of the system, the *ETH* with the lowest *cid* is consistently considered as master by all *ETH*s that are up. A simple timed statechart model of master election is depicted in Figure 8.1.

The behavior of *ETH*s defined by the statechart can be summarized as follows. Note that *syncTimer* and *slaveTimer* are clock variables that are used to define time dependent behavior in the same way as clock variables are used in the common timed automata formalism: their values are constantly increasing by a uniform rate and can be checked in guard expressions and reset by actions.

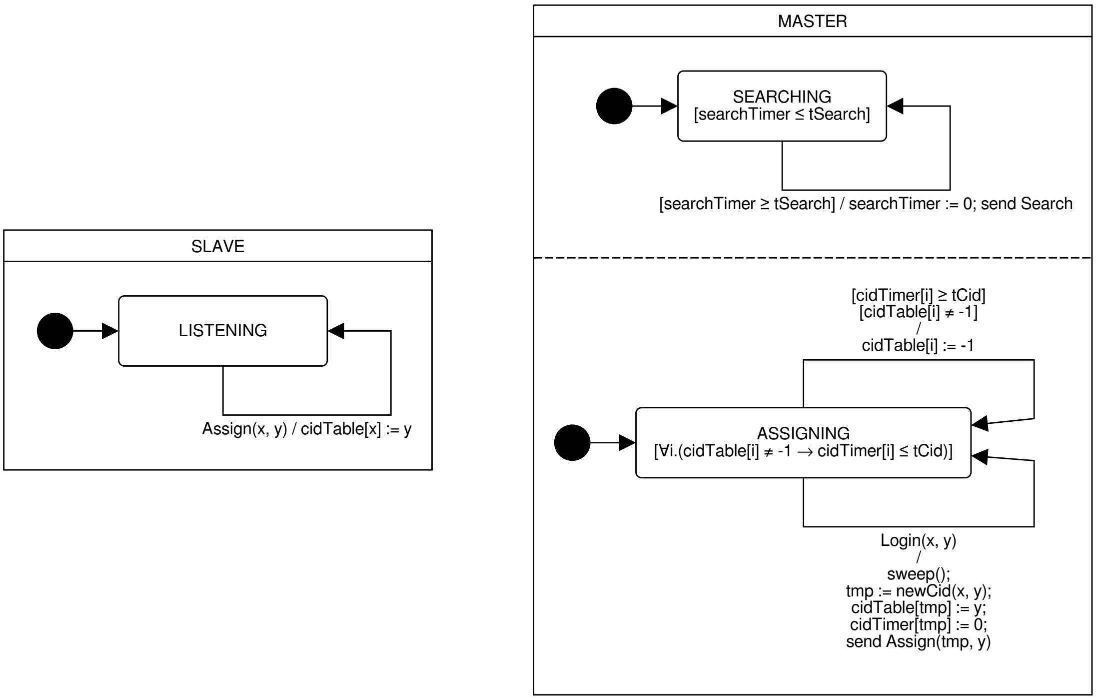


Figure 8.2: Assignment of logical addresses as (a) slave (b) master

Moreover, state invariants can be defined (written into the state symbol in square brackets) that may also refer to clock variables.

- A message $Normal(cid)$ is broadcasted at every $tSync$ time units with the cid of the ETH as payload. This message serves as a heartbeat between ETHs.
- Initially, the ETH is a slave. If for the last $tSlave$ time units the ETH has not received any $Normal$ messages with lower cid value than the ETH itself has, then the ETH becomes master.
- An ETH remains master as long as it does not receive a message $Normal$ with a cid lower than its own cid.

Summarizing the above, an ETH is master iff all heartbeats received in the last $tSlave$ time units are from ETHs with a cid not lower than its own – the reception of a message with a lower cid value immediately brings the ETH back to the slave role.

8.2.2 Assignment of Logical Addresses

To keep record of the cids of all LIOs, each ETH maintains an array $cidTable$ that is indexed with cids from range $[4...13]$ and contains $hwid$s as values. For a cid x from the above range, an ETH then assumes that $cidTable[x]$ is the $hwid$ of the LIO to whom x is assigned as cid. If $cidTable[x] = -1$, then x is assumed to be unassigned.

The assignment of cids is performed by the master ETH, while slaves only update their $cidTable$s based on received messages. The statechart model of the cid assignment is showed in Figure 8.2 for both masters and slaves. These models can be interpreted as refinement of the corresponding composite states (containing this way sub-machines) in the model of master election.

The role of a slave ETH is simply to keep track of assigned cid values by listening to $Assign$ messages sent by the master and updating its $cidTable$ based on them.

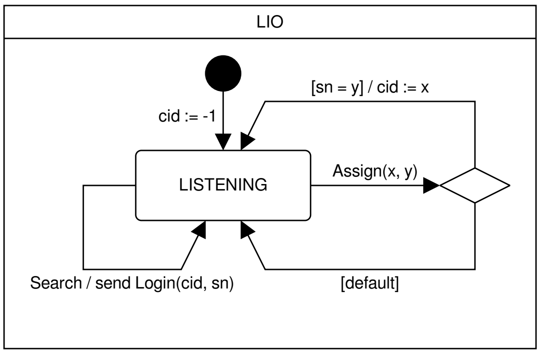


Figure 8.3: Behavior of LIOs

The behavior of a master ETH can be summarized as follows (note that here $searchTimer$ is a clock variable and $cidTimer$ is an array of clock variables).

- Every $tSync$ time units it broadcasts a message $Search$. As a response, each LIO is supposed to send a message $Login(x, y)$ where x is the current cid of the LIO (-1 if undefined) and $y \neq -1$ is its $hwid$.
- Upon receiving a message $Login(x, y)$, the following steps are performed.
  1. By calling a procedure SWEEP, any occurrence of a given $hwid$ other than the first is erased from $cidTable$. As it turned out during verification, this method is required for resilient operation of the protocol.
  2. Based on the entries in $cidTable$, a new cid is calculated by the function NEWCID so that the following conditions are met.
     - If y appears in $cidTable$ as a value at some index, then the index is returned as result. Since SWEEP ensures that each $hwid$ is unique in $cidTable$, the result is well defined.
     - Else if $x \neq -1$ and x is unassigned then it is returned as result.
     - Else the smallest unassigned cid is returned. The existence of such a cid is ensured by SWEEP.
  3. The $cidTable$ is updated, the corresponding timer in $cidTimer$ is reset and a message $Assign$ is sent with the new cid.
- Other than that, if a row of $cidTable$ corresponding to an assigned cid was not updated in the last $tCid$ time units, then the cid gets unassigned.

8.2.3 LIOs

The model of a LIO is shown on Figure 8.3.

- Initially, the LIO has no cid assigned ($cid = -1$).
- Upon receiving a message $Search$, the LIO replies with a message $Login(cid, hwid)$.
- Upon receiving a message $Assign(x, y)$, if $hwid = y$, then cid is updated to x. The message is ignored otherwise.

8.3 Verification of the Protocol

This section details the application of the approach presented in the previous section in the verification of the protocol. The formal, dense time model of the system was constructed as a network of timed automata, whose operational semantics can be expressed in terms of a transition system. The verification aims at proving the resilience of the system: even in the presence of transient faults, the components shall be able to communicate with each other. This requires that after a finite number of faults, the system will persistently have a unique master and all LIOs have a logical address assigned in a consistent way. Among others, this formulation admits the verification of correctness in the presence of the following transient faults:

- An ETH or LIO restarts.
- The content of an ETH's $cidTable$ changes.
- The cid of a LIO changes.
- The content or recipient of a message changes.
- A message is lost.
- A message is created.

8.3.1 Decomposing the Verification of the Protocol

To enable model checking, the statechart model containing the composite statecharts of all ETHs and LIOs is mapped to a network of timed automata. Signal events are handled by an automaton representing a bounded capacity communication channel that is able to store and delay the sent messages until their reception. The resulting formal model can be analyzed by the model checker Uppaal [Beh+06].

As the protocol has two functionalities (master election and assignment of communication IDs), the requirement of resilience is a composite property that includes the temporal correctness of these functionalities. Accordingly, resilience is formalized as a persistence property $\mathsf{FG}(\varphi \wedge \psi)$, where φ expresses that there is a unique master in the system, whereas ψ states that each LIO was assigned a unique logical address that corresponds to a row of the master's $cidTable$.

The following proof tree shows the decomposition of this top level requirement.

$$\dfrac{\dfrac{\mathcal{S}_\top \models \mathsf{FG}\varphi \qquad \mathcal{S}_\top^\varphi \models \mathsf{FG}\psi}{\mathcal{S}_\top \models \mathsf{FG}(\varphi \wedge \psi)}\ \text{FG-detachment}}{\mathcal{S}_F \models \mathsf{FG}(\varphi \wedge \psi)}\ \text{fault abstraction}$$

Instead of verifying the system model with different fault configurations, we employ the fault abstraction rule: this simulates that the verification starts after the occurrence of any finite number of transient faults, leaving the system in any state. The next reduction rule splits up the property according to the FG-detachment rule: in the protocol, master election is a precondition for the successful logical address assignment. By proving the subproperties we can infer the validity of the property itself. Now, the task is to prove two properties referring to different aspects of the system.

- $\mathcal{S}_\top \models \mathsf{FG}\varphi$ expresses that the system initialized in any state will have a master and the participants will not change their role.
- $\mathcal{S}_\top^\varphi \models \mathsf{FG}\psi$ expresses that the system initialized in any state will finally have consistent cid assignment, assuming there is a unique stable master.

In the following sections the proofs of these two properties are detailed.

8.3.2 Verification of Master Election

The verification of the master election protocol is reduced to the model checking of the $\mathsf{FG}\varphi$ temporal logic specification on system $\mathcal{S}_\top$. Now, the rule G-detachment can be applied, and thus the resulting model checking queries to be proven are $\mathcal{S}_\top \models \mathsf{F}\varphi$ and $\mathcal{S}_\varphi \models \mathsf{G}\varphi$.

As these resulting temporal logic formulas refer to only some aspects of the system, cone of influence reduction can be employed to construct transition system $\mathcal{S}_1$ from $\mathcal{S}_\top$. Behavior related to cid assignment is not relevant in the verification of master election: no interaction in master election is triggered or influenced by the administration of cid assignment. This enables the cone of influence reduction to fully reduce the model to the following elements, that are included in the model $\mathcal{S}_1$:

- Four ETHs (with behavior as in Figure 8.1).
- Communication channel.

The model $(\mathcal{S}_1)_\varphi$ is the same as $\mathcal{S}_1$, the only difference is that the initial states are those where the master has already been elected.

The property to be verified is φ, which refers to the situation of successful master election:

- ETH_0 is master.
- ETH_1, ETH_2 and ETH_3 are slave.

The formal proof tree that was applied in the verification of the master election protocol is the following:

$$\dfrac{\dfrac{\mathcal{S}_1 \models \mathsf{F}\varphi}{\mathcal{S}_\top \models \mathsf{F}\varphi} \qquad \dfrac{(\mathcal{S}_1)_\varphi \models \mathsf{G}\varphi}{\mathcal{S}_\varphi \models \mathsf{G}\varphi}}{\mathcal{S}_\top \models \mathsf{FG}\varphi}\ \text{G-detachment}$$

8.3.3 Verification of Logical Address Assignment

The verification of the logical address assignment protocol is reduced to the model checking of temporal logic specification $\mathsf{FG}\psi$ on system $\mathcal{S}_\top^\varphi$. Similar to the verification of the master election protocol, the rule G-detachment can be applied to decompose the problem into two parts. The resulting model checking queries to be proven are $\mathcal{S}_\top^\varphi \models \mathsf{F}\psi$ and $\mathcal{S}_\psi^\varphi \models \mathsf{G}\psi$.

In transition system $\mathcal{S}_\top^\varphi$, the master election procedure is assumed to have been successful, thus in the verification of cid assignment we can exploit that there will be no more changes in the roles of the ETHs. In addition, the resulting temporal logic formulas refer only to aspects of the system related to cid assignment. These advantages of the decomposition can be exploited and cone of influence reduction can be applied to construct transition system $\mathcal{S}_2$ from $\mathcal{S}_\top^\varphi$, where $\mathcal{S}_2$ contains:

- ETH_0 as master (with behavior as in Figure 8.2).
- Ten LIOs (Figure 8.3).
- Communication channel.

The property to be verified is ψ, which refers to the situation where the LIOs have unique cid values and it is consistent with the knowledge of the master:

- For each two rows of $ETH_0.cidTable$, if they contain an equal value, then both values are -1 (thus the assigned cid values in the table of the master are unique).
- The cids assigned to LIOs correspond to the values in $ETH_0.cidTable$.
- Each LIO has a cid different from -1.

The formal proof tree that was applied in the verification of the cid assignment protocol is the following:

$$\dfrac{\dfrac{\mathcal{S}_2 \models \mathsf{F}\psi}{\mathcal{S}_\top^\varphi \models \mathsf{F}\psi} \qquad \dfrac{(\mathcal{S}_2)_\psi \models \mathsf{G}\psi}{\mathcal{S}_\psi^\varphi \models \mathsf{G}\psi}}{\mathcal{S}_\top^\varphi \models \mathsf{FG}\psi}\ \text{G-detachment}$$

8.3.4 Result of the Verification

The verification problem was decomposed according to the proof rules detailed in the previous sections. Cone of influence reduction was applied to the formal models, which significantly reduced the size of the formal models. When the first version of the protocol design was verified, insufficiencies were revealed in the protocol: an oscillation between states could occur that prevented the proof of the liveness property regarding the successful cid assignment. After the required modification of the design (among others the inclusion of the procedure SWEEP the Uppaal model checker could then verify all the four tasks successfully within seconds. Without the proposed approach, namely the decomposition and abstraction steps, the verification could not succeed due to resource limitations, and because properties in Uppaal are restricted to a narrow subset of CTL.

8.4 Conclusions

In this chapter, we devised an approach which combines the decomposition of the temporal specification with abstraction. Fault abstraction is used to construct a single formal model that covers the effects of various transient faults that may disturb the operation of the protocol. This abstract model includes all behaviors of the system where a finite number of transient faults is allowed to occur. We proved the soundness of the approach. We introduced two decomposition rules for persistence properties in linear temporal logic which are tailored to the problem domain. When applying these rules, we exploited the composite structure of the system functionalities (behavior) to obtain simpler subtasks where the system could be simplified significantly by cone of influence reduction. By using the introduced approach, the verification of the protocol was successfully elaborated.

8.4.1 Thesis Summary

This concludes Thesis 4.2 of this dissertation. We summarize it as follows.

> **Thesis 4.2** *A decomposition method for liveness checking of hierarchical real-time protocols.* I proposed a generic decomposition scheme for the verification of real-time systems with a hierarchical structure in functionality. The method is applicable when a combination of safety and liveness properties shall be verified.

Chapter 9

Summary of the Research Results

We conclude by comparing the challenges formulated in Section 1.2 against the contributions described in this dissertation.

9.1 Thesis 1

> **Challenge 1** *Configurable abstraction refinement-based model checking.* Most tools focus on a specific algorithm and formalism to solve a particular verification task. Is it possible to provide a generic, modular and configurable model checking framework that supports the development, evaluation and application of abstraction refinement-based algorithms for the reachability analysis of models in different formalisms?

In Chapter 3, we introduced Theta, a generic, modular and configurable model checking framework for abstraction refinement-based reachability analysis for different formalisms. We described the architecture that helps to implement, evaluate and combine various algorithms in a modular way for different formalisms. We also demonstrated the applicability of the framework by use cases for the verification of hardware, PLC, software and timed automata models. Results of the evaluation with configuring and combining different analysis modules support the need for a generic framework, such as Theta.

For the specific case of timed automata, in Chapter 4, we presented *an algorithmic framework* for the lazy abstraction based location reachability checking. We formalized the combination of abstractions and proved its properties. This framework allowed the straightforward implementation of efficient model checkers using configurable combined strategies.

We summarize Thesis 1 as follows.

> **Thesis 1** *A framework for abstraction refinement-based reachability checking.* I proposed solutions for making abstraction refinement based model checking configurable in terms of modeling formalism, abstract domain, and refinement strategy.
>
> 1.1 *Architecture of a configurable model checking framework.* I designed the architecture, interfaces and generic algorithmic components of Theta, a generic, modular, and configurable model checking framework that enables the combination of various abstract domains, interpreters, and strategies for abstraction and refinement, applied to models of various formalisms.

1.2 *A uniform formalization of abstraction refinement strategies for timed automata.* I proposed and proved correct a formal algorithmic framework that enables the uniform formalization and combined use of various abstract domains and abstraction refinement strategies for the location reachability checking of timed automata.

The results of Thesis 1 enabled the definition, implementation and empirical evaluation of novel algorithms and algorithm combinations. Related publications are the following: [*j*2; *c*6; *c*10] .

9.2 Thesis 2

Challenge 2 *Abstraction refinement for timed automata.* Abstraction refinement has been successfully used in model checking, and in particular for model checking software. Is it possible to provide abstraction refinement algorithms that are efficient in the domain of real-time systems?

In Chapter 5, we proposed a lazy reachability checking algorithm for timed automata based on interpolation for zones. Moreover, we proposed two refinement strategies, both a combination of forward search, backward search and interpolation. We demonstrated with experiments that - even without the use of extrapolation - the method is competitive with sophisticated non-convex abstractions in both execution time and memory consumption.

We summarize Thesis 2 as follows.

Thesis 2 *Lazy reachability checking for timed automata using interpolants.* I proposed a solution for the location reachability problem of timed automata based on the following steps.

- I defined interpolation for zones, and gave an algorithm for computing a zone interpolant from two inconsistent zones, represented as canonical difference bound matrices.
- Based on pre- and post-image computation for timed automata in the zone abstract domain, I generalized the notion of zone interpolation to sequences of interpolants, this way enabling its use for abstraction refinement-based location reachability checking of timed automata.
- I proposed forward and backward zone interpolation as approaches to lazy abstraction refinement.
- I experimentally evaluated the performance of the proposed abstraction refinement strategies, and showed that these compare favorably to known methods based on efficient lazy non-convex abstractions.

The proposed method is applicable to more expressive variants of timed automata, e.g. to automata with diagonal constraints in guards [BLR05], or to updatable timed automata [Bou04]. Related publications are the following: [*j*2; *c*8; *c*9] .

9.3 Thesis 3

Challenge 3 *Model checking timed automata with discrete variables.* For practical real-time systems, design models typically contain discrete data variables with nontrivial data flow besides real-valued clock variables. Is it possible to provide methods for alleviating state space explosion in such models?

In Chapter 6, we proposed a lazy algorithm for the location reachability problem of timed automata with discrete variables. The method is based on controlling the visibility of discrete variables by using interpolation for valuations of variables. We demonstrated with experiments that our abstraction and refinement strategy, combined with lazy methods for the abstraction of continuous clock variables, can achieve significant reduction in the size of the generated state space during search, typically with low or no overhead in execution time, and in cases even with an additional speedup.

We summarize Thesis 3 as follows.

> **Thesis 3** *Lazy reachability checking for timed automata with discrete variables.* I proposed a solution for the location reachability problem of timed automata with discrete variables based on the following steps.
>
> - I defined interpolation between a valuation and a formula, and gave an algorithm for computing valuation interpolants.
> - Based on weakest precondition computation for transitions of timed automata, I generalized the notion of valuation interpolation to sequences of interpolants, this way enabling its use for abstraction refinement-based location reachability checking.
> - I proposed forward and backward valuation interpolation as approaches to lazy abstraction refinement.
> - I experimentally evaluated the performance of the proposed abstraction refinement strategies, and showed that these are suitable to significantly reduce the number of states generated during state space exploration of timed automata models with many discrete variables.

The proposed method does not rely on SMT solving, and is thus applicable to models with arbitrary expressions and statements over discrete variables, e.g. division, multiplication between variables, etc. Related publications are the following: [*j*1; *j*2; *c*4; *c*7; *c*11; *c*12; *e*13] .

9.4 Thesis 4

> **Challenge 4** *Liveness checking for industrial real-time systems.* Requirements for industrial real-time systems are often formalized in terms of liveness properties. Is it possible to provide methods for liveness checking of such systems, while still supporting the various semantic features that are present in such models?

In Chapter 7, we proposed (1) the extension of calendar automata to provide the calendar system formalism that allows convenient modeling of the core protocols of communicating real-time systems, (2) the extension of k-induction based techniques to support the verification of both safety and liveness properties of calendar systems, and (3) the tool support to perform static analysis, derivation of invariants and artifacts required for k-induction based automated verification. The framework proved to be useful to find problems in industrial protocols.

In Chapter 8, we devised an approach which combines the decomposition of the temporal specification with abstraction. Fault abstraction is used to construct a single formal model that covers the effects of various transient faults that may disturb the operation of the protocol. This abstract model includes all behaviors of the system where a finite number of transient faults is allowed to occur. We proved the soundness of the approach. We introduced two decomposition rules for persistence properties in linear temporal logic which are tailored to the problem domain. When applying these rules, we exploited the composite structure of the system functionalities(behavior) to obtain simpler subtasks where the system could be simplified significantly by cone of influence reduction. By using the introduced approach, the verification of the protocol was successfully elaborated.

We summarize Thesis 4 as follows.

> **Thesis 4** *Improved methods for liveness checking of industrial real-time protocols.* During my research, I proposed improved methods for liveness verification of industrial real-time protocols.
>
> 4.1 *K-induction based liveness checking of real-time systems.* I proposed the calendar system formalism that allows convenient modeling of the core protocols of communicating real-time systems. By a series of transformation steps, I extended k-induction based model checking to support the verification of both safety and liveness properties of calendar systems. Moreover, I provided a tool-supported solution for the derivation of lemmas required for successful k-induction based automated verification.
>
> 4.2 *A decomposition method for liveness checking of hierarchical real-time protocols.* I proposed a generic decomposition scheme for the verification of real-time systems with a hierarchical structure in functionality. The method is applicable when a combination of safety and liveness properties shall be verified.

We successfully applied the method during the verification of a distributed safety critical protocol, whose main functionality is to guarantee reliable communication between components in a distributed SCADA (Supervisory Control and Data Acquisition) system. Related publications are the following: [*c*3; *c*5] .

Appendix **A**

Appendix

In this appendix, we include details that are relevant for the evaluation of technical soundness of the dissertation.

A.1 Lemmas and Proofs

Lemma 10. $A \preceq A{\restriction_X}$

Lemma 11. $A \preceq B \Rightarrow (\!|A|\!) \subseteq (\!|B|\!)$

Lemma 12. $A \preceq B \Rightarrow post_t(A) \preceq post_t(B)$

Lemma 13. $post_t(\!|A|\!) \subseteq (\!|post_t(A)|\!)$

Proof of Proposition 6. Assume $(s_1, s_2) \sqsubseteq (s_1', s_2')$. By Definition 4.9, we have $s_1 \sqsubseteq s_1'$ and $s_2 \sqsubseteq s_2'$. By soundness of $\mathbb{D}_1$ and $\mathbb{D}_2$, it follows that $[\![s_1]\!] \subseteq [\![s_1']\!]$ and $[\![s_2]\!] \sqsubseteq [\![s_2']\!]$. Thus $[\![s_1]\!] \cap [\![s_2]\!] \subseteq [\![s_1']\!] \cap [\![s_2']\!]$. By Definition 4.9, it follows that $[\![(s_1, s_2)]\!] \subseteq [\![(s_1', s_2')]\!]$.

By soundness of $\mathbb{D}_1$ and $\mathbb{D}_2$, we have $\Sigma_0 \subseteq [\![\mathsf{init}_1]\!]$ and $\Sigma_0 \subseteq [\![\mathsf{init}_2]\!]$. Thus $\Sigma_0 \subseteq [\![\mathsf{init}_1]\!] \cap [\![\mathsf{init}_2]\!]$. By Definition 4.9, we obtain $\Sigma_0 \subseteq [\![\mathsf{init}]\!]$.

By soundness of $\mathbb{D}_1$ and $\mathbb{D}_2$, we have $post_t[\![s_1]\!] \subseteq [\![\mathsf{post}_t(s_1)]\!]$ and $post_t[\![s_2]\!] \subseteq [\![\mathsf{post}_t(s_2)]\!]$. Thus $post_t[\![s_1]\!] \cap post_t[\![s_2]\!] \subseteq [\![\mathsf{post}_t(s_1)]\!] \cap [\![\mathsf{post}_t(s_2)]\!]$. Moreover, as, $post_t$ is an image, we obtain $post_t([\![s_1]\!] \cap [\![s_2]\!]) \subseteq post_t[\![s_1]\!] \cap post_t[\![s_2]\!]$. Altogether, we have $post_t([\![s_1]\!] \cap [\![s_2]\!]) \subseteq [\![\mathsf{post}_t(s_1)]\!] \cap [\![\mathsf{post}_t(s_2)]\!]$, from which $post_t[\![(s_1, s_2)]\!] \subseteq [\![\mathsf{post}_t(s_1, s_2)]\!]$ follows by Definition 4.9. □

Proof of Proposition 9. Assume $Z_1 \subseteq Z_2$. Then $(\!|Z_1|\!) \subseteq (\!|Z_2|\!)$ by the monotonicity of images in $\subseteq$.

By Lemma 10, we have $\Sigma_0 \preceq \Sigma_0{\restriction_C}$. Then by Lemma 11 it follows that $(\!|\Sigma_0|\!) \subseteq (\!|\Sigma_0{\restriction_C}|\!)$. As $\Sigma_0 = (\!|\Sigma_0|\!)$, we obtain $\Sigma_0 \subseteq (\!|\Sigma_0{\restriction_C}|\!)$.

By Lemma 10, we have $post_t(Z) \preceq post_t^C(Z)$. Thus by Lemma 11, we have $(\!|post_t(Z)|\!) \subseteq (\!|post_t^C(Z)|\!)$. By Lemma 13 it follows that $post_t(\!|Z|\!) \subseteq (\!|post_t^C(Z)|\!)$. □

Proof of Proposition 11. We have $W \sqsubseteq W' \Rightarrow [\![W]\!] \subseteq [\![W']\!]$ by Proposition 9. Moreover, $\Sigma_0 \subseteq [\![\top]\!]$ and $post_t[\![W]\!] \subseteq [\![\top]\!]$ trivially hold. □

Proof of Lemma 5. Assume $post_t^C(Z) \subseteq Z'$. Then $(\!|post_t^C(Z)|\!) \subseteq (\!|Z'|\!)$ by the monotonicity of images in $\subseteq$. Moreover, from Lemma 10, we obtain $post_t(Z) \preceq post_t^C(Z)$, from which $(\!|post_t(Z)|\!) \subseteq (\!|post_t^C(Z)|\!)$ follows by Lemma 11. Also, $post_t(\!|Z|\!) \subseteq (\!|post_t(Z)|\!)$ by Lemma 13. Thus $post_t(\!|Z|\!) \subseteq (\!|Z'|\!)$. □

Proof of Lemma 4.

$$\begin{aligned} & Z \cap pre_t^C(Z') \subseteq \bot \\ \Leftrightarrow\; & Z \cap (post_t^C)^{-1}(Z') \subseteq \bot && \text{(by definition)} \\ \Leftrightarrow\; & Z \subseteq ((post_t^C)^{-1}(Z'))^c \\ \Leftrightarrow\; & Z \subseteq (post_t^C)^{-1}((Z')^c) && \text{(property of images)} \\ \Leftrightarrow\; & post_t^C(Z) \subseteq (Z')^c && \text{(property of images)} \\ \Leftrightarrow\; & post_t^C(Z) \cap Z' \subseteq \bot && \square \end{aligned}$$

Proof of Proposition 16. $\nu = \nu' \Rightarrow (\!|\nu|\!) \subseteq (\!|\nu'|\!)$ trivially holds by congruence. The rest follows by a reasoning analogous to the one applied in Proposition 9. □

Proof of Proposition 18. Assume $\nu \preceq \nu'\restriction_{Q'}$ and $Q' \subseteq Q'$. Thus we have $\nu\restriction_Q \preceq \nu'\restriction_{Q'}$, and by Lemma 11 we obtain $(\!|\nu\restriction_Q|\!) \subseteq (\!|\nu'\restriction_{Q'}|\!)$. Moreover, $\Sigma_0 \subseteq (\!|\nu_0\restriction_\emptyset|\!)$ and $post_t(\!|\nu\restriction_Q|\!) \subseteq (\!|post_t^D(\nu)\restriction_\emptyset|\!)$ trivially hold. □

Proof of Lemma 7. Assume $\alpha \preceq \beta$. Then $post_t(\alpha) \preceq post_t(\beta)$ by Lemma 12. Thus $post_t^D(\alpha) \preceq post_t^D(\beta)$ by Lemma 3. □

Proof of Lemma 8. Assume $post_t^D(\nu) \preceq \nu'$. Then $(\!|post_t^D(\nu)|\!) \subseteq (\!|\nu'|\!)$ by Lemma 11. Moreover, from Lemma 10, we obtain $post_t(\nu) \preceq post_t^D(\nu)$, from which $(\!|post_t(\nu)|\!) \subseteq (\!|post_t^D(\nu)|\!)$ follows by Lemma 11. Also, $post_t(\!|\nu|\!) \subseteq (\!|post_t(\nu)|\!)$ by Lemma 13. Thus $post_t(\!|\nu|\!) \subseteq (\!|\nu'|\!)$. □

A.2 Tables

Table A.1: Execution time for PAT and MCTA models (full)

model	BBB	BBF	BBN	BFB	BFF	BFN	BLB	BLF	BLN	DBB	DBF	DBN	DFB	DFF	DFN	DLB	DLF	DLN
critical 3	2.2	2.1	1.6	2.1	2.1	1.7	2.6	2.6	1.9	2.8	2.7	2.0	2.7	2.6	2.0	2.7	2.7	1.8
critical 4	45.2	45.4	37.0	42.1	41.9	34.4	55.4	54.9	41.4	56.4	55.4	46.3	50.6	49.6	41.4	48.8	48.2	34.9
csma 9	12.8	13.0	8.2	13.4	13.4	8.7	11.7	11.6	7.2	20.0	20.2	16.3	22.0	22.0	18.4	35.9	36.1	32.1
csma 10	31.7	32.0	19.2	33.0	33.0	20.6	28.7	28.6	17.1	61.2	61.9	51.6	69.3	70.1	60.0	155.2	155.7	150.3
csma 11	82.4	82.3	49.7	85.6	85.9	53.2	72.4	71.8	43.2	229.3	230.8	207.4	270.6	273.0	254.7	-	-	-
csma 12	241.0	242.2	141.4	254.7	254.8	154.8	208.7	209.3	125.8	-	-	-	-	-	-	-	-	-
fddi 50	-	-	-	-	-	-	9.6	9.7	9.1	3.3	3.3	3.0	3.3	3.4	3.0	2.3	2.3	2.1
fddi 70	-	-	-	-	-	-	22.9	22.9	22.3	5.5	5.6	5.1	5.8	5.9	5.3	4.1	4.2	3.7
fddi 90	-	-	-	-	-	-	50.3	50.6	49.5	9.7	9.8	9.5	10.2	10.6	9.7	7.5	7.4	7.1
fddi 110	-	-	-	-	-	-	90.0	89.4	86.8	15.3	15.1	14.9	15.9	15.8	15.4	11.9	11.9	11.4
fischer 7	4.1	4.1	3.1	4.1	4.3	3.3	3.2	3.1	2.3	4.1	4.0	3.0	4.3	4.2	3.3	3.2	3.1	2.3
fischer 8	9.7	9.8	7.8	10.3	10.2	8.4	7.2	7.1	5.4	9.8	9.8	8.1	10.1	10.2	8.5	7.1	7.2	5.2
fischer 9	30.5	30.4	24.8	34.1	33.6	28.3	19.6	19.0	14.1	31.7	31.4	26.5	34.2	34.4	28.9	18.9	18.7	14.1
fischer 10	117.8	117.1	99.2	135.3	134.4	116.1	65.4	64.3	48.9	123.1	121.6	105.7	139.1	137.0	120.1	65.7	64.3	49.9
lynch 7	6.3	6.4	4.4	6.4	6.5	4.5	4.9	4.9	3.1	5.5	5.7	4.0	5.8	6.0	4.3	4.4	4.6	2.9
lynch 8	16.2	16.3	11.1	17.4	17.7	11.9	11.5	11.7	7.1	15.1	15.7	11.3	16.3	16.6	12.2	10.1	10.8	6.7
lynch 9	56.8	56.4	38.8	62.0	62.1	44.4	36.2	35.7	21.2	52.1	52.5	39.3	56.9	56.9	44.1	31.2	31.6	20.2
bocdp	13.1	19.4	9.5	13.2	19.3	10.2	11.5	17.9	6.1	10.8	15.3	9.0	10.2	14.4	8.5	10.1	14.3	6.0
bocdpf	17.1	21.7	19.9	15.9	22.5	20.9	14.5	20.9	12.1	10.1	13.9	15.8	9.3	12.5	16.2	9.3	13.6	10.3
brp	20.2	26.6	23.1	13.4	20.3	12.9	9.5	13.1	7.1	32.8	20.7	28.4	17.8	12.6	20.2	18.7	19.7	8.7
c1	4.9	5.1	2.6	4.4	4.4	2.3	5.4	5.7	3.0	3.4	3.8	2.1	3.1	3.4	1.7	3.6	3.8	2.0
c2	10.6	10.7	7.3	8.7	9.6	5.5	11.8	12.4	6.5	6.8	6.9	4.7	6.2	6.4	4.0	7.0	7.2	4.3
c3	11.7	12.3	8.0	9.8	10.8	6.4	13.6	14.6	8.1	7.7	8.4	5.3	7.1	7.6	4.7	8.2	8.8	4.8
c4	86.6	84.4	66.2	70.7	71.0	46.6	117.8	119.6	82.7	46.0	50.6	36.6	41.7	45.7	29.3	50.6	53.6	33.6
e1	6.0	6.2	4.8	5.5	5.8	3.9	6.5	7.2	4.4	4.7	4.8	2.9	4.1	4.5	2.5	4.6	5.0	2.6
m1	2.9	2.8	2.3	2.7	2.8	2.2	5.2	5.2	3.4	1.4	1.4	1.3	1.2	1.2	1.0	1.9	1.9	1.8
m2	8.1	8.0	6.1	7.1	7.2	5.2	14.7	15.3	9.4	2.5	2.6	3.1	2.4	2.4	2.6	4.8	4.8	4.4
m3	8.1	7.7	6.2	8.1	8.4	6.0	17.2	16.5	9.8	3.8	3.6	2.9	3.0	3.1	2.6	5.9	5.4	4.7
m4	32.4	33.5	21.2	28.9	28.1	17.9	84.8	87.6	43.9	6.5	7.0	7.4	6.3	6.8	6.1	16.3	17.7	10.8
n1	3.4	3.2	2.9	2.9	2.7	2.6	5.5	5.2	3.8	1.3	1.4	1.5	1.3	1.3	1.3	1.9	1.8	1.9
n2	8.8	8.7	7.9	7.4	7.4	7.0	17.7	16.9	11.9	2.8	3.0	3.4	2.8	2.9	3.1	5.4	5.2	4.3
n3	9.0	8.8	8.0	8.4	8.3	6.8	17.7	16.1	12.2	3.4	3.3	4.0	3.0	3.2	3.5	5.5	5.6	5.5
n4	35.4	36.2	31.0	30.9	30.7	28.9	87.7	87.3	57.5	7.1	7.8	9.3	6.6	6.6	8.7	22.3	20.6	21.3

Table A.2: Number of nodes for PAT and MCTA models (full)

model	BBB	BBF	BBN	BFB	BFF	BFN	BLB	BLF	BLN	DBB	DBF	DBN	DFB	DFF	DFN	DLB	DLF	DLN
critical 3	13641	13641	13641	12981	12981	12981	21699	21699	21699	19036	19036	19036	18310	18310	18310	25697	25503	25697
critical 4	434393	434393	433787	395188	395188	394525	772221	772221	777784	635308	635308	635308	564014	564014	564014	1043487	1045220	1043487
csma 9	78552	78552	78552	78552	78552	78552	78552	78552	78552	98989	98989	98989	98989	98989	98989	217656	217656	217656
csma 10	200649	200649	200649	200649	200649	200649	200649	200649	200649	274759	274759	274759	274759	274759	274759	745149	745149	745149
csma 11	501432	501432	501432	501432	501432	501432	501432	501432	501432	787898	787898	787898	787898	787898	787898	-	-	-
csma 12	1230757	1230757	1230757	1230757	1230757	1230757	1230757	1230757	1230757	-	-	-	-	-	-	-	-	-
fddi 50	-	-	-	-	-	-	2098	2098	2098	503	503	503	503	503	503	503	503	503
fddi 70	-	-	-	-	-	-	2961	2961	2961	703	703	703	703	703	703	703	703	703
fddi 90	-	-	-	-	-	-	3881	3881	3881	903	903	903	903	903	903	903	903	903
fddi 110	-	-	-	-	-	-	4678	4678	4678	1103	1103	1103	1103	1103	1103	1103	1103	1103
fischer 7	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405	26405
fischer 8	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353	95353
fischer 9	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211	339211
fischer 10	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211	1191211
lynch 7	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915	46915
lynch 8	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801	162801
lynch 9	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491	563491
bocdp	33591	33694	98314	32639	32627	94801	33030	33149	96460	32537	32252	97125	29846	29565	84643	33341	33052	97462
bocdpf	41707	36661	218745	38492	36443	212225	40083	36808	209430	29557	26946	196782	26544	23734	183402	30230	27612	197234
brp	52410	73202	110600	36761	55312	72117	58825	84355	115675	95439	63298	150970	56786	38752	111705	119826	128906	169672
c1	19041	19642	22157	17156	17230	20967	27058	27608	32963	15174	15174	18802	14973	14973	18614	18292	18292	22968
c2	51588	50192	73326	44906	45223	67433	71657	72459	103476	40179	40179	57896	39644	39644	57170	48069	48069	69760
c3	57676	57653	94286	50713	51927	86285	81524	82427	136015	47911	47911	77698	46593	46593	76335	56833	55936	95548
c4	378267	363199	968171	339560	332348	876266	502423	492180	1365289	327474	314683	758739	318480	304934	737964	389018	359139	932334
e1	26461	25866	35989	24677	23353	31247	37105	38938	47199	20520	20533	23729	20299	20300	23657	23931	23927	27513
m1	4907	4935	8998	4394	4511	8541	13171	13929	27216	2279	2279	4753	1901	1901	3625	4970	4727	15233
m2	18182	18398	40413	16246	16558	31932	44095	44812	112634	5723	5723	18737	5673	5673	15471	16603	15547	60995
m3	18447	18037	40054	18369	19188	38128	49032	46948	118485	9181	8592	17797	7181	7160	16189	20291	18202	68091
m4	69661	71845	172868	66255	63475	145378	157864	162564	464477	20787	20687	72302	20335	20335	61915	61606	60085	215984
n1	5163	5130	9030	4222	4095	7645	13731	13263	26467	2000	2000	4466	1921	1921	3898	4579	4363	13869
n2	18628	18441	40640	15648	15849	33054	49197	46568	122680	6070	6070	16477	5933	5933	15514	18315	17348	53212
n3	18779	18604	40983	17177	17295	32493	48007	44607	122178	7083	7083	20484	6536	6536	16677	18031	18596	74393
n4	71159	71250	178362	63674	63491	150864	160825	160154	493530	21150	21374	72527	18798	18277	69308	74430	65098	326938

Table A.3: Execution time for the diagonal version of Fischer's protocol (full)

model	BBB	BBF	BBN	BFB	BFF	BFN	BLB	BLF	BLN	DBB	DBF	DBN	DFB	DFF	DFN	DLB	DLF	DLN
diag 3	0.3	0.3	0.2	0.3	0.2	0.2	-	-	-	0.3	0.3	0.2	0.3	0.3	0.2	-	-	-
diag 4	0.7	0.7	0.6	0.7	0.7	0.6	-	-	-	1.0	1.0	0.9	0.8	0.8	0.7	-	-	-
diag 5	1.8	1.8	1.5	1.7	1.7	1.5	-	-	-	4.7	4.6	4.0	2.0	2.0	1.8	-	-	-
diag 6	5.8	5.8	4.9	5.7	5.7	4.9	-	-	-	62.2	61.0	56.1	6.9	6.8	6.0	-	-	-
diag 7	21.3	21.4	19.3	21.4	21.3	19.9	-	-	-	-	-	-	27.7	27.6	25.7	-	-	-
diag 8	108.3	106.7	99.2	111.8	112.2	104.1	-	-	-	-	-	-	153.6	152.7	144.2	-	-	-
split 3	0.6	0.5	0.8	0.3	0.3	0.7	0.4	0.4	0.6	0.7	0.7	1.1	0.5	0.5	0.8	0.4	0.4	0.6
split 4	4.2	4.3	19.7	1.0	1.0	7.1	1.9	1.9	5.5	9.0	8.9	30.0	1.9	1.9	5.4	2.5	2.4	5.3
split 5	74.6	74.7	-	3.1	3.1	-	19.9	20.4	259.4	-	-	-	11.8	11.5	-	45.4	45.8	-
split 6	-	-	-	11.6	11.7	-	-	-	-	-	-	-	-	-	-	-	-	-
split 7	-	-	-	58.5	58.7	-	-	-	-	-	-	-	-	-	-	-	-	-
split 8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
opt 3	0.4	0.4	0.3	0.3	0.3	0.3	0.3	0.3	0.2	0.4	0.4	0.3	0.4	0.4	0.4	0.3	0.3	0.2
opt 4	1.6	1.6	1.5	0.9	0.9	1.6	0.9	0.9	0.9	2.7	2.6	2.1	1.2	1.2	1.8	1.0	1.0	0.8
opt 5	9.9	10.0	11.8	2.8	2.8	12.7	4.3	4.4	4.8	79.1	80.7	35.4	7.8	8.1	15.8	4.5	4.6	4.1
opt 6	161.5	162.7	221.3	10.0	10.2	244.4	36.4	36.5	49.9	-	-	-	-	-	-	43.9	44.7	39.3
opt 7	-	-	-	47.1	47.4	-	-	-	-	-	-	-	-	-	-	-	-	-
opt 8	-	-	-	293.5	296.0	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.4: Number of nodes for the diagonal version of Fischer's protocol (full)

model	BBB	BBF	BBN	BFB	BFF	BFN	BLB	BLF	BLN	DBB	DBF	DBN	DFB	DFF	DFN	DLB	DLF	DLN
diag 3	199	199	199	193	193	193	-	-	-	246	246	246	220	220	220	-	-	-
diag 4	1045	1045	1045	933	933	933	-	-	-	1800	1800	1800	1262	1262	1262	-	-	-
diag 5	4926	4926	4926	4181	4181	4181	-	-	-	17929	17929	17929	5515	5515	5515	-	-	-
diag 6	21685	21685	21685	17815	17815	17815	-	-	-	264445	264445	264445	24772	24772	24772	-	-	-
diag 7	90252	90252	90252	73137	73137	73137	-	-	-	-	-	-	100147	100147	100147	-	-	-
diag 8	360233	360233	360233	291593	291593	291593	-	-	-	-	-	-	406392	406392	406392	-	-	-
split 3	585	585	2448	333	333	1929	664	664	3137	946	946	3277	492	492	2096	811	811	3322
split 4	8163	8163	79998	1833	1833	34579	7144	7144	68999	23459	23459	132835	3847	3847	31827	12527	12527	82939
split 5	121370	121370	-	9388	9388	-	90877	90877	1572515	-	-	-	27135	27135	-	207627	207627	-
split 6	-	-	-	45566	45566	-	-	-	-	-	-	-	-	-	-	-	-	-
split 7	-	-	-	211828	211828	-	-	-	-	-	-	-	-	-	-	-	-	-
split 8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
opt 3	341	341	621	252	252	619	350	350	621	401	401	652	372	372	639	399	399	655
opt 4	2726	2726	5534	1330	1330	5591	2591	2591	5666	5674	5674	8234	2305	2305	6092	3268	3268	5837
opt 5	24455	24455	53714	6550	6550	51465	20987	20891	51431	180464	180464	155731	23529	23529	63504	29124	29124	54586
opt 6	230929	232241	525802	30634	30634	494997	178954	178043	474498	-	-	-	-	-	-	272734	272802	541533
opt 7	-	-	-	137788	137788	-	-	-	-	-	-	-	-	-	-	-	-	-
opt 8	-	-	-	601970	601970	-	-	-	-	-	-	-	-	-	-	-	-	-

Publications

Publication List

Number of publications:	16
Number of peer-reviewed journal papers (written in English):	2
Number of articles in journals indexed by WoS or Scopus:	2
Number of publications (in English) with at least 50% contribution of the author:	9
Number of peer-reviewed publications:	16
Number of independent citations:	9

Publications Linked to the Theses

	Journal papers	International conference and workshop papers	Local events	Technical reports
Thesis 1	[j2]*	[c6]; [c10]	–	–
Thesis 2	[j2]*	[c8]†; [c9]	–	–
Thesis 3	[j1]; [j2]*	[c4]; [c7]†; [c11]; [c12]	[e13]	–
Thesis 4	–	[c3]; [c5]	–	–

* These publications are attached to multiple theses.
† In the years 2016 and 2017, the PhD Minisymposium, organized by the BUTE Department of Measurement and Information Systems, had international participation.
This classification follows the faculty's Ph.D. publication score system.

Journal Papers

[j1] Tamás Tóth and István Majzik. Formal verification of real-time systems with data processing. *Periodica Polytechnica Electrical Engineering and Computer Science* 61(2), 2017, pp. 166–174. DOI: 10.3311/PPee.9766.

[j2] Tamás Tóth and István Majzik. Configurable verification of timed automata with discrete variables. *Acta Informatica* (online first), 2020. DOI: 10.1007/s00236-020-00393-4.

International Conference and Workshop Papers

[c3] Tamás Tóth, András Vörös, and István Majzik. K-induction based verification of real-time safety critical systems. In: *Proceedings of the 8th International Conference on Dependability and Complex Systems, DepCoS-RELCOMEX 2013*, AISC, vol. 224, pp. 469–478. Springer, 2013. DOI: 10.1007/978-3-319-00945-2_43.

▷ *Own contributions (1) the calendar system formalism (2) the model checking approach (3) the implementation of the tool support (4) the modeling and verification of the case study.*

[c4] Tamás Tóth, András Vörös, and István Majzik. Verification of a real-time safety-critical protocol using a modelling language with formal data and behaviour semantics. In: *Computer Safety, Reliability, and Security. SAFECOMP 2014 Workshops*, LNCS, vol. 8696, pp. 207–218. Springer, 2014. DOI: 10.1007/978-3-319-10557-4_24.

▷ *Own contributions (1) the modeling formalism (2) the model checking approach (3) the implementation of the tool support (4) the modeling and verification of the case study.*

[c5] Tamás Tóth, András Vörös, and István Majzik. A decomposition method for the verification of a real-time safety-critical protocol. In: *Software Engineering for Resilient Systems. 7th International Workshop, SERENE 2015*, LNCS, vol. 9274, pp. 31–45. Springer, 2015. DOI: 10.1007/978-3-319-23129-7_3.

▷ *Own contributions (1) the model checking approach (2) the modeling and verification of the case study.*

[c6] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In: *Formal Techniques for Distributed Objects, Components, and Systems. 36th IFIP WG 6.1 International Conference, FORTE 2016*, LNCS, vol. 9688, pp. 158–174. Springer, 2016. DOI: 10.1007/978-3-319-39570-8_11.

▷ *Own contributions (1) some insight on the model checking approach (2) partial implementation of the tool support.*

[c7] Tamás Tóth and István Majzik. Formal modeling of real-time systems with data processing. In: *Proceedings of the 23rd PhD Mini-Symposium*, pp. 46–49. BME Department of Measurement and Information Systems. Accommodated by IEEE Hungary, 2016.

[c8] Tamás Tóth and István Majzik. Timed automata verification using interpolants. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 82–85. BME Department of Measurement and Information Systems, 2017. DOI: 10.5281/zenodo.291907.

[c9] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata using interpolants. In: *Formal Modeling and Analysis of Timed Systems. 15th International Conference, FORMATS 2017*, LNCS, vol. 10419, pp. 264–280. Springer, 2017. DOI: 10.1007/978-3-319-65765-3_15.

[c10] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In: *Proceedings of the 17th Conference on Formal Methods in Computer Aided Design, FMCAD 2017*, pp. 176–179. FMCAD Inc., 2017. DOI: 10.23919/FMCAD.2017.8102257.

▷ *Own contributions (1) the design of the architecture, interfaces, and generic algorithmic components of the framework (2) partial implementation of the tool support.*

[c11] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata with discrete variables. In: *Model Checking Software. 25th International Symposium, SPIN 2018*, LNCS, vol. 10869, pp. 235–254. Springer, 2018. DOI: 10.1007/978-3-319-94111-0_14.

[c12] Rebeka Farkas, Tamás Tóth, Ákos Hajdu, and András Vörös. Backward reachability analysis for timed automata with data variables. In: *Automated Verification of Critical Systems*, Electronic Communications of the EASST, vol. 76, pp. 1–20. 2018. DOI: 10.14279/tuj.eceasst.76.1076.
▷ *Own contributions (1) some insight on the model checking approach (2) partial implementation of the tool support.*

Local Conference and Workshop Papers

[e13] Tamás Tóth and István Majzik. A framework for formal verification of real-time systems. In: *Proceedings of the 22nd PhD Mini-Symposium*, pp. 12–13. BME Department of Measurement and Information Systems. Accommodated by IEEE Hungary, 2015.

Supplementary Material

[s14] Tamás Tóth and István Majzik. Supplementary Material for the Paper "Configurable Verification of Timed Automata with Discrete Variables". Zenodo. 2020. DOI: 10.5281/zenodo.3965792.

Additional Publications (Not Linked to Theses)

International Conference and Workshop Papers

[c15] Gyula Sallai and Tamás Tóth. Boosting software verification with compiler optimizations. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 66–69. BME Department of Measurement and Information Systems, 2017. DOI: 10.5281/zenodo.291903.

[c16] Bence Czipó, Ákos Hajdu, Tamás Tóth, and István Majzik. Exploiting hierarchy in the abstraction-based verification of statecharts using SMT solvers. In: *International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2017*, EPTCS, vol. 245, pp. 31–45. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.245.3.

[c17] Gyula Sallai, Ákos Hajdu, Tamás Tóth, and Zoltán Micskei. Towards evaluating size reduction techniques for software model checking. In: *Fifth International Workshop on Verification and Program Transformation, VPT 2017*, EPTCS, vol. 253, pp. 75–91. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.253.7.

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science* 126(2), 1994, pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In: *Static Analysis*, LNCS, vol. 7460, pp. 300–316. Springer, 2012. DOI: 10.1007/978-3-642-33125-1_21.

[Alb+12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: a framework for abstraction- and interpolation-based software verification. In: *Computer Aided Verification*, LNCS, vol. 7358, pp. 672–678. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_48.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters* 21(4), 1985, pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0.

[ÁSH21] Zsófia Ádám, Gyula Sallai, and Ákos Hajdu. GAZER-THETA: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 12652, pp. 433–437. Springer, 2021. DOI: 10.1007/978-3-030-72013-1_27.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science* 66(2), 2002, pp. 160–177. DOI: 10.1016/S1571-0661(04)80410-9.

[BC00] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In: *Formal Methods in Computer-Aided Design*, LNCS, vol. 1954, pp. 409–426. Springer, 2000. DOI: 10.1007/3-540-40922-X_23.

[BC05] Patricia Bouyer and Fabrice Chevalier. On conciseness of extensions of timed automata. *Journal of Automata, Languages and Combinatorics* 10(4), 2005, pp. 393–405. DOI: 10.25596/jalc-2005-393.

[Beh+03] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 2619, pp. 254–270. Springer, 2003. DOI: 10.1007/3-540-36577-X_18.

[Beh+04] Gerd Behrmann, Patricia Bouyer, Kim G. 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*, LNCS, vol. 2988, pp. 312–326. Springer, 2004. doi: 10.1007/978-3-540-24730-2_25.

[Beh+06] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In: *Quantitative Evaluation of Systems*, pp. 125–126. IEEE, 2006. doi: 10.1109/QEST.2006.59.

[Ber+10] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over EMF models. In: *Model Driven Engineering Languages and Systems*, LNCS, vol. 6394, pp. 76–90. Springer, 2010. doi: 10.1007/978-3-642-16145-2_6.

[Bér+98] Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae* 36(2,3), 1998, pp. 145–182. doi: 10.3233/FI-1998-36233.

[Bey+07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *Software Tools for Technology Transfer* 9(5), 2007, pp. 505–525. doi: 10.1007/s10009-007-0044-z.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 9636, pp. 887–904. Springer, 2016. doi: 10.1007/978-3-662-49674-9_55.

[Bey21] Dirk Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 12652, pp. 401–422. Springer, 2021. doi: 10.1007/978-3-030-72013-1_24.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 1579, pp. 193–207. Springer, 1999. doi: 10.1007/3-540-49059-0_14.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: a tool for configurable software verification. In: *Computer Aided Verification*, LNCS, vol. 6806, pp. 184–190. Springer, 2011. doi: 10.1007/978-3-642-22110-1_16.

[BL13] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In: *Fundamental Approaches to Software Engineering*, LNCS, vol. 7793, pp. 146–162. Springer, 2013. doi: 10.1007/978-3-642-37057-1_11.

[BLR05] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: forward analysis of timed systems. In: *Formal Modelling and Analysis of Timed Systems*, LNCS, vol. 3829, pp. 112–126. Springer, 2005. doi: 10.1007/11603009_10.

[Bou03] Patricia Bouyer. Untameable timed automata! In: *Theoretical Aspects of Computer Science*, LNCS, vol. 2607, pp. 620–631. Springer, 2003. doi: 10.1007/3-540-36494-3_54.

[Bou04] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design* 24(3), 2004, pp. 281–320. doi: 10.1023/B:FORM.0000026093.21513.31.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 2031, pp. 268–283. Springer, 2001. DOI: 10.1007/3-540-45319-9_19.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In: *Computer Aided Verification*, LNCS, vol. 2102, pp. 260–264. Springer, 2001. DOI: 10.1007/3-540-44585-4_25.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In: *Verification, Model Checking, and Abstract Interpretation*, LNCS, vol. 6538, pp. 70–87. Springer, 2011. DOI: 10.1007/978-3-642-18275-4_7.

[Büc62] Julius R. Büchi. On a decision method in restricted second order arithmetic. In: *Logic, Methodology, and Philosophy of Science*, pp. 1–11. Stanford University Press, 1962.

[Bur+92] Jerry R. Burch, Edmund L. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 1992, pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: semantics, algorithms and tools. In: *Advances in Petri Nets*, LNCS, vol. 3098, pp. 87–124. Springer, 2004. DOI: 10.1007/978-3-540-27755-2_3.

[Cab+16] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, Keijo Heljanko, and Jason Baumgartner. Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation* 9, 2016, pp. 135–172.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Principles of Programming Languages*, pp. 238–252. ACM, 1977. DOI: 10.1145/512950.512973.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In: *Principles of Programming Languages*, pp. 269–282. ACM, 1979. DOI: 10.1145/567752.567778.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems* 16(5), 1994, pp. 1512–1542. DOI: 10.1145/186025.186051.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CGR10] Alessandro Carioni, Silvio Ghilardi, and Silvio Ranise. MCMT in the land of parametrized timed automata. In: *International Verification Workshop (VERIFY-2010)*, pp. 47–64. 2010.

[CGS04] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23(7), 2004, pp. 1113–1123. DOI: 10.1109/TCAD.2004.829807.

[CGS08] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 4963, pp. 397–412. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_30.

[Cha+02] Pankaj Chauhan, Edmund M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: *Formal Methods in Computer-Aided Design*, LNCS, vol. 2517, pp. 33–51. Springer, 2002. doi: 10.1007/3-540-36126-X_3.

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In: *Computer Aided Verification*, LNCS, vol. 1855, pp. 154–169. Springer, 2000. doi: 10.1007/10722167_15.

[Cla+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5), 2003, pp. 752–794. doi: 10.1145/876638.876643.

[Cla+05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SatAbs: SAT-based predicate abstraction for ANSI-C. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 3440, pp. 570–574. Springer, 2005. doi: 10.1007/978-3-540-31980-1_40.

[Coh91] Joëlle Cohen-Chesnot. On the expressive power of temporal logic for infinite words. *Theoretical Computer Science* 83(2), 1991, pp. 301–312. doi: 10.1016/0304-3975(91)90281-6.

[Cra57] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic* 22(3), 1957, pp. 269–285. doi: 10.2307/2963594.

[CS12] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In: *Formal Methods in Computer-Aided Design*, pp. 52–59. IEEE, 2012.

[CU98] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In: *Computer Aided Verification*, LNCS, vol. 1427, pp. 293–304. Springer, 1998. doi: 10.1007/BFb0028753.

[Die+17] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. Newton in software model checking. In: *Foundations of Software Engineering*, pp. 487–497. ACM, 2017. doi: 10.1145/3106237.3106307.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In: *Computer Aided Verification*, LNCS, vol. 407, pp. 197–212. Springer, 1990. doi: 10.1007/3-540-52148-8_17.

[DKL07] Henning Dierks, Sebastian Kupferschmid, and Kim G. Larsen. Automatic abstraction refinement for timed automata. In: *Formal Modelling and Analysis of Timed Systems*, LNCS, vol. 4763, pp. 114–129. Springer, 2007. doi: 10.1007/978-3-540-75454-1_10.

[DS04] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, LNCS, vol. 3253, pp. 199–214. Springer, 2004. doi: 10.1007/978-3-540-30206-3_15.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 1384, pp. 313–329. Springer, 1998. doi: 10.1007/BFb0054180.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2(3), 1982, pp. 241–266. doi: 10.1016/0167-6423(83)90017-5.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science* 89(4), 2003, pp. 543–560. DOI: 10.1016/S1571-0661(05)82542-3.

[Fer+15] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor M. González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics* 11(6), 2015, pp. 1400–1410. DOI: 10.1109/TII.2015.2489184.

[GMS18] Paul Gastin, Sayan Mukherjee, and Balaguru Srivathsan. Reachability in timed automata with diagonal constraints. In: *International Conference on Concurrency Theory*, LIPIcs, vol. 118, 28:1–28:17. Dagstuhl, 2018. DOI: 10.4230/LIPIcs.CONCUR.2018.28.

[GMS19] Paul Gastin, Sayan Mukherjee, and Balguru Srivathsan. Fast algorithms for handling diagonal constraints in timed automata. In: *Computer Aided Verification*, LNCS, vol. 11561, pp. 41–59. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_3.

[GMS20] Paul Gastin, Sayan Mukherjee, and Balguru Srivathsan. Reachability for updatable timed automata made faster and more effective. In: *Foundations of Software Technology and Theoretical Computer Science*, LIPIcs, vol. 182, 47:1–47:17. Dagstuhl, 2020. DOI: 10.4230/LIPIcs.FSTTCS.2020.47.

[Gru06] Orna Grumberg. Abstraction and refinement in model checking. In: *Formal Methods for Components and Objects*, LNCS, vol. 4111, pp. 219–242. Springer, 2006. DOI: 10.1007/11804192_11.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In: *Computer Aided Verification*, LNCS, vol. 1254, pp. 72–83. Springer, 1997. DOI: 10.1007/3-540-63166-6_10.

[Gul+08] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 4963, pp. 443–458. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_33.

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In: *Principles of Programming Languages*, pp. 58–70. ACM, 2002. DOI: 10.1145/503272.503279.

[Hen+04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In: *Principles of Programming Languages*, pp. 232–244. ACM, 2004. DOI: 10.1145/964001.964021.

[Her+11] Frédéric Herbreteau, Dileep Kini, Balaguru Srivathsan, and Igor Walukiewicz. Using non-convex approximations for efficient analysis of timed automata. In: *Foundations of Software Technology and Theoretical Computer Science*, LIPIcs, vol. 13, pp. 78–89. Dagstuhl, 2011. DOI: 10.4230/LIPIcs.FSTTCS.2011.78.

[HM17] Ákos Hajdu and Zoltán Micskei. Exploratory analysis of the performance of a configurable CEGAR framework. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 34–37. BME Department of Measurement and Information Systems, 2017. DOI: 10.5281/zenodo.291895.

[Hoj+14] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In: *Horn Clauses for Verification and Synthesis*, EPTCS, vol. 169, pp. 39–52. Open Publishing Association, 2014. doi: 10.4204/EPTCS.169.6.

[HSW12] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Better abstractions for timed automata. In: *Logic in Computer Science*, pp. 375–384. IEEE, 2012. doi: 10.1109/LICS.2012.48.

[HSW13] Frédéric Herbreteau, Balaguru Srivathsan, and Igor Walukiewicz. Lazy abstractions for timed automata. In: *Computer Aided Verification*, LNCS, vol. 8044, pp. 990–1005. Springer, 2013. doi: 10.1007/978-3-642-39799-8_71.

[IW14] Tobias Isenberg and Heike Wehrheim. Timed automata verification via ic3 with zones. In: *Formal Methods and Software Engineering*, LNCS, vol. 8829, pp. 203–218. Springer, 2014. doi: 10.1007/978-3-319-11737-9_14.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: high-performance language-independent model checking. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 9035, pp. 692–707. Springer, 2015. doi: 10.1007/978-3-662-46681-0_61.

[KJN12a] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. Beyond lassos: complete SMT-based bounded model checking for timed automata. In: *Formal Techniques for Distributed Systems*, LNCS, vol. 7273, pp. 84–100. Springer, 2012. doi: 10.1007/978-3-642-30793-5_6.

[KJN12b] Roland Kindermann, Tommi Junttila, and Ilkka Niemelä. SMT-based induction methods for timed systems. In: *Formal Modelling and Analysis of Timed Systems*, LNCS, vol. 7595, pp. 171–187. Springer, 2012. doi: 10.1007/978-3-642-33365-1_13.

[Kla02] Felix Klaedtke. Complementation of Büchi automata using alternation. In: Erich Grädel, Wolfgang Thomas, and Thomas Wilke (eds.), *Automata Logics, and Infinite Games*, LNCS, vol. 2500, pp. 61–77. Springer, 2002. doi: 10.1007/3-540-36387-4_4.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994. doi: 10.1515/9781400864041.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with Wolverine. In: *Computer Aided Verification*, LNCS, vol. 6806, pp. 573–578. Springer, 2011. doi: 10.1007/978-3-642-22110-1_45.

[LNZ04] Denis Lugiez, Peter Niebert, and Sarah Zennou. A partial order semantics approach to the clock explosion problem of timed automata. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 2988, pp. 296–311. Springer, 2004. doi: 10.1007/978-3-540-24730-2_24.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 4963, pp. 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In: *Computer Aided Verification*, LNCS, vol. 2725, pp. 1–13. Springer, 2003. doi: 10.1007/978-3-540-45069-6_1.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In: *Computer Aided Verification*, LNCS, vol. 4144, pp. 123–136. Springer, 2006. doi: 10.1007/11817963_14.

[McM10] Kenneth L. McMillan. Lazy annotation for program testing and verification. In: *Computer Aided Verification*, LNCS, vol. 6174, pp. 104–118. Springer, 2010. DOI: 10.1007/978-3-642-14295-6_10.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control* 9(5), 1966, pp. 521–530. DOI: 10.1016/S0019-9958(66)80013-X.

[Mol+18] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The GAMMA statechart composition framework: design, verification and code generation for component-based reactive systems. In: *International Conference on Software Engineering*, pp. 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

[MOS03] Leonardo de Moura, Sam Owre, and Natarajan Shankar. *The SAL Language Manual.* Tech. rep. SRI-CSL-01-02 (Rev. 2). SRI International, 2003.

[MPS11] Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully symbolic model checking for timed automata. In: *Computer Aided Verification*, LNCS, vol. 6806, pp. 616–632. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_50.

[MRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: from refutation to verification. In: *Computer Aided Verification*, LNCS, vol. 2725, pp. 14–26. Springer, 2003. DOI: 10.1007/978-3-540-45069-6_2.

[Mur89] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE* 77(4), 1989, pp. 541–580. DOI: 10.1109/5.24143.

[Pik05] Lee Pike. *Real-Time System Verification by k-induction.* Tech. rep. NASA/TM-2005-213751. National Aeronautics and Space Administration, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. In: *Foundations of Computer Science*, pp. 46–57. IEEE, 1977. DOI: 10.1109/SFCS.1977.32.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In: *International Symposium on Programming*, LNCS, vol. 137, pp. 337–351. Springer, 1982. DOI: 10.1007/3-540-11494-7_22.

[Rey07] Pierre-Alain Reynier. *Diagonal Constraints Handled Efficiently in UPPAAL.* Tech. rep. LSV-07-02. Laboratoire Spécification et Vérification, ENS Cachan, France, 2007.

[RSM19] Victor Roussanaly, Ocan Sankur, and Nicolas Markey. Abstraction refinement algorithms for timed automata. In: *Computer Aided Verification*, LNCS, vol. 11561, pp. 22–40. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_2.

[Saf88] Shmuel Safra. On the complexity of ω-automata. In: *Foundations of Computer Science*, pp. 319–327. IEEE, 1988. DOI: 10.1109/SFCS.1988.21948.

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science* 149(1), 2006, pp. 79–96. DOI: 10.1016/j.entcs.2005.11.018.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In: *Formal Methods in Computer-Aided Design*, LNCS, vol. 1954, pp. 127–144. Springer, 2000. DOI: 10.1007/3-540-40922-X_8.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In: *Formal Methods and Testing*, LNCS, vol. 4949, pp. 1–38. Springer, 2008. DOI: 10.1007/978-3-540-78917-8_1.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In: *Logic in Computer Science*, pp. 322–331. IEEE, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation* 115(1), 1994, pp. 1–37. doi: 10.1006/inco.1994.1092.

[WJ15] Weifeng Wang and Li Jiao. Difference bound constraint abstraction for timed automata reachability checking. In: *Formal Techniques for Distributed Systems*, LNCS, vol. 9039, pp. 146–160. Springer, 2015. doi: 10.1007/978-3-319-19195-9_10.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In: *Foundations of Computer Science*, pp. 185–194. IEEE, 1983. doi: 10.1109/SFCS.1983.51.