VERIFICATION OF REAL-TIME SYSTEMS: IMPROVING TOOL SUPPORT

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of doctor of philosophy

By

Rodolfo Sabás Gómez

October 2006

Abstract

We address a number of limitations of Timed Automata and real-time model-checkers, which undermine the reliability of formal verification. In particular, we focus on the model-checker Uppaal as a representative of this technology.

Timelocks and Zeno runs represent anomalous behaviours in a timed automaton, and may invalidate the verification of safety and liveness properties. Currently, model-checkers do not offer adequate support to prevent or detect such behaviours. In response, we develop new methods to guarantee timelock-freedom and absence of Zeno runs, which improve and complement the existent support. We implement these methods in a tool to check Uppaal specifications.

The requirements language of model-checkers is not well suited to express sequence and iteration of events, or past computations. As a result, validation problems may arise during verification (i.e., the property that we verify may not accurately reflect the intended requirement). We study the logic PITL, a rich propositional subset of Interval Temporal Logic, where these requirements can be more intuitively expressed than in model-checkers. However, PITL has a decision procedure with a worst-case non-elementary complexity, which has hampered the development of efficient tool support. To address this problem, we propose (and implement) a translation from PITL to the second-order logic WS1S, for which an efficient decision procedure is provided by the tool MONA. Thanks to the many optimisations included in MONA, we obtain an efficient decision procedure for PITL, despite its non-elementary complexity.

Data variables in model-checkers are restricted to bounded domains, in order to obtain fully automatic verification. However, this may be too restrictive for certain kinds of specifications (e.g., when we need to reason about unbounded buffers). In response, we develop the theory of Discrete Timed Automata as an alternative formalism for real-time systems. In Discrete Timed Automata, WS1S is used as the assertion language, which enables MONA to assist invariance proofs. Furthermore, the semantics of urgency and synchronisation adopted in Discrete Timed Automata guarantee, by construction, that specifications are free from a large class of timelocks. Thus, we argue that well-timed specifications are easier to obtain in Discrete Timed Automata than in Timed Automata and most other notations for real-time systems.

Acknowledgements

I am indebted to my family for the love and support I always find in them.

Papá, Mamá, Ger y Luchito: Gracias por tanto cariño y comprensión!

I also wish to extend my gratitude to my relatives in Córdoba, who always have a place for me in their hearts and thoughts (*Tíos y Primos: Gracias!*). My gratitude and love also go to Ismini, who believes in me and makes me feel whole ($\Sigma'\alpha\gamma\alpha\pi\acute{\alpha}\omega\ \pi o\lambda\acute{\upsilon},\ M\eta\nu\eta\ \mu o\upsilon$).

This thesis would have not been possible without the support, guidance and encouragement of my supervisor, Dr. Howard Bowman. He is a continuous source of inspiration, and his clear and brilliant thinking, and optimism and perseverance, have helped me to overcome many difficult situations. I am indebted to Howard for the opportunity he gave me to come to England as a PhD student, and for believing in me (*Thanks Howard!*). I am also indebted to Dr. Juan Carlos Augusto, who introduced me to research and, in particular, to real-time systems and temporal logics. Juan's enthusiasm for research, and his tenacity to achieve the goals, are just but a few of many qualities that I admire in him (*Gracias Juan!*). Special thanks go to my friends in Argentina: Diego and his family, José and Analía, Bernardo, Fernando and family, Esteban and Adrián, Esteban, Gabriel, Marcos, Yanina, Roberto, Matías, Laura, Marcelo, Mercedes, Karina and so many others (*Gracias chicos!*). I owe my happiest moments in Canterbury to my friends Gift, Axel, Clara and Lars-Åke, Julia, Nadia, Su-Wei and Pei Chuen, Kathrin, Adolfo and Nicia, Leo, Alex, Marcel and Lauana, Renato and Viviana, Xinbei, Lester and Sara, Ed, Brad, Matt and Carrie, Su Li, Vorapol, Damian, Christian, Keith, Miguel, Peter and Sharon and John, my friends in the "Saturday Legion", "The Oldies", and so many others (*Thanks guys!*).

I also thank Andy King, Peter and Janet Linington, Simon Thompson, Stefan Kahrs, Eerke Boiten, John Derrick, Janet Carter, Florence Benoy and other people at the Computing Laboratory, and Kim Larsen, Frits Vaandrager, Ben Moszkowski and Dimitar Guelev, for fruitful discussions, support and encouragement. I am grateful for the financial support provided by the Computing Laboratory and the ORS Award Scheme; my studies would have not been possible without it.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

1.1 Context: Formal Methods for Real-time Systems

Most systems are concurrent by nature; systems are made up of a number of processes (components) evolving in parallel, which interact to realise the overall system behaviour. In computer science, the need for concurrent theories is evident, and has been so for a long time [115, 86]. Our task is to map systems to representations that are amenable to computation; clearly, more accurate mappings produce better models, which in turn enable a better analysis of the original system, which in turn allows us to construct better (and more reliable) software and hardware devices.

The sequential, monolithic abstractions which characterised our first attempts to model systems, gave way to concurrent ones, and so the power of reasoning about a system in terms of its components was gained, together with the inevitable complexity that arises from considering the component interactions. Then, concurrency theory evolved to accommodate the analysis of *real-time systems*, whose correct behaviour depends upon time constraints [1, 110, 129, 145, 168]. For example, the correct behaviour of communication protocols over unreliable media typically depends on determining the minimum length of timeout intervals; if transmission delays are not properly taken into account, data may be lost if timeouts occur too soon.

Real-time systems constitute the core of many safety-critical applications, where much depends on guaranteeing that systems behave as expected. Safety-critical applications range from simple gas burners, to gate controllers in train crossings, self-destruction mechanisms in rockets, and air-traffic control systems. It is not difficult to imagine, for instance, how air-traffic control systems that do not meet safety criteria put lives at risk.

Real-time systems are amongst the most challenging systems to analyse; verifying correctness, and having confidence in the verification method itself, are both crucial. This motivates

the use of formal methods for the analysis of real-time systems. For lack of a better definition, a formal method usually consists of a computational model to describe the system behaviour; a specification language in which the system can be modelled; a requirements language in which the correctness criteria can be expressed; and verification techniques to prove that requirements are met.

For example, the behaviour of real-time systems can be captured by timed transition systems [129, 168, 31, 144]. Real-time systems can be specified in Timed Automata [7], and properties can be expressed in requirements languages such as Computation Tree Logic (CTL) [58], Timed Computation Tree Logic (TCTL) [6], or Timed Büchi Automata [5]. Model-checkers such as HyTech [82], Uppaal [22], Kronos [169] and Red [165], can be used to explore the state space of the timed automaton (a suitable finite-state abstraction of the timed transition system [102]), looking for models (or counterexamples) of the stated property.

Formal methods are usually classified as deductive or algorithmic. Deductive methods were the first formalisms to be developed; they are characterised by very expressive specification and requirements languages, and verification techniques are based on proof systems (axioms and inference rules). Correctness is then asserted by means of a mathematical proof and requires considerable user interaction; verification is typically assisted by theorem provers. Timed I/O Automata [92], the Temporal Logic of Actions [1] and Clock Transition Systems [110], are probably the most mature deductive frameworks, supported by theorem provers such as PVS [107], Isabelle [90], HOL [163] and Step [25].

Algorithmic methods, on the other hand, compromise expressiveness (both in the system specification language and in the requirements language) to allow for automatic verification. Timed Automata and real-time model-checkers are representative of algorithmic methods, with Uppaal being one of the most developed model-checkers currently available.

Deductive and algorithmic approaches have sustained continuous development over the years; as a result, practitioners now have powerful notations and tools to analyse a broad range of non-trivial problems. Nonetheless, much work is still needed to improve notations and the related tool support. In particular, this thesis is concerned with the *reliability* of formal methods for real-time systems; i.e., to what extent practitioners can trust their specifications, requirements and the results of verification. The next section explores this issue in more detail.

1.2 Motivation: Reliability Issues in Model-checking

The previous section introduced our concern with reliability in formal methods. We want to use formal methods to guarantee the correctness of our real-time systems: Formal methods provide the means to deal with the complexity of real-time systems, and give us confidence in the results

of our analysis (formal methods are grounded in sound mathematical theories). However, we must be aware of the extent to which a formal method is able to guarantee correctness.

We know that systems can be proven correct only with respect to requirements, i.e., correctness is a relative concept. Similarly, our confidence in formal verification depends on how faithfully our specification (e.g., a network of timed automata) represents the real system, and to what extent our properties (e.g., temporal logic formulae) represent the system requirements.

In what follows, we comment on a number of limitations of specification and requirements languages which undermine confidence in verification.

We will see that timed automata[1] may represent abnormal behaviours that are not possible in real systems, and which compromise verification. This is an example of *unwanted* expressive power in the formal notation. However, in addition, problems arise due to *limited* expressiveness. For example, model-checkers must deal with finite-state systems, variables are restricted to bounded domains, and requirements that refer to sequences and iteration of events, or past computations, cannot be intuitively expressed.

1.2.1 Timelocks in Timed Automata

In specifications of real-time systems, we often need to represent *urgent* actions, i.e., those whose execution cannot be delayed beyond a given time bound. For instance, in certain communication protocols, the sender will timeout and retransmit the last message, if an acknowledgment has not been received by a certain time.

In all formal notations for real-time systems, the semantics of urgent actions is achieved by preventing time from passing beyond a given bound. When such a bound is reached, the system may evolve only through the immediate execution of an action, which (in this sense) becomes urgent at that point.

In timed automata, the specifier may express constraints to prevent time from passing beyond a certain bound; hence, urgent actions can be represented indirectly. However, it is the specifier's responsibility to ensure that, whenever a state is reached where time cannot pass any further, a sequence of actions can be performed which lead to a state where time may pass again.

A timelock is a state where time cannot diverge (i.e., time cannot pass beyond a certain bound).[2] Timelocks are counterintuitive situations, and are caused by specification errors. For instance, a timelock is reached when a constraint in the timed automaton prevents time from passing any further, but no action is enabled at that point (or the enabled actions do not lead to a state where time may diverge).

[1] Throughout this thesis, we will use "timed automata" (in lower case) to denote particular specifications, and "Timed Automata" (in upper case) to denote the formal notation.

[2] In this thesis, divergence refers to time-divergence.

Specifications cannot be reliably verified in specifications where timelocks occur. For instance, a property stating that a certain unwanted state is never reached (namely, a safety property) may hold in a timelocked specification just because a timelock prevents time from passing up to a point where such a state becomes reachable. Thus, because timelocks cannot be implemented, this unwanted state may still occur in some execution of the implemented system.

A related problem concerns the detection of Zeno runs, which denote arbitrarily fast executions. Zeno runs, like timelocks, are counterintuitive: a process cannot be infinitely fast. However, unlike timelocks, Zeno runs may not be caused by specification errors, but simply because the specification is realised at a high-level of abstraction (e.g., when lower time bounds for the execution of actions are irrelevant for the intended analysis). Zeno runs are not necessarily undesirable, and do not imply the occurrence of timelocks.

For example, if the property to verify depends only on the ordering of events and not on their relative timing, an "untimed" abstraction (of a more detailed specification) may suffice to perform verification. Moreover, for complex specifications, such abstractions may be the only way to cope with state explosion. Note that, loops in this untimed abstraction will naturally induce Zeno runs, because lower time bounds were removed from the more detailed specification.

Nonetheless, in general, verification in real-time systems is well-defined only over divergent executions. To some extent, the problem is similar to that of verification of liveness properties without fairness assumptions: we do not want to consider runs where the system "chooses" to perform infinitely fast. Thus, we need methods to guarantee absence of Zeno-runs, but we also need methods to guarantee timelock-freedom that are insensitive to the occurrence of Zeno runs.

In the context of timed automata, timelocks have been investigated by a number of researchers, including Henzinger et al. [85], Bornot et al. [33], Tripakis [156, 157] and Bowman [38, 39]. Yet, model-checkers do not offer satisfactory support for timelock detection.

Model-checkers require the specification to be timelock-free (and in some cases, in addition, free from Zeno runs). However, few model-checkers support detection of timelocks and Zeno runs, and the methods implemented suffer from a number of shortcomings.

In addition, although timed automata models have been proposed where (a certain class of) timelocks can be prevented by construction [33, 39], most model-checkers cannot support such notations, which are fundamentally different in the semantics of urgency and synchronisation. Therefore, we aim to improve timelock-detection in model-checkers, giving practitioners better tools to guarantee the sanity of specifications.

1.2.2 Requirements in Branching-time Logics

It is generally accepted that branching-time logics, such as CTL and TCTL, have expressiveness limitations that hamper the natural representation of requirements [161]. Among other limitations, formulae in branching-time logics do not characterise requirements in an intuitive manner (practical experience with these logics supports this claim; see e.g., [14]).

This introduces a validation problem: a specification may satisfy a certain formula, but such a formula may not properly represent the intended requirement, thus affecting our confidence in the results of verification.

Real-time model-checkers use CTL- and TCTL-based requirements languages, and therefore inherit such expressiveness limitations of branching-time logics. For example, requirements that refer to sequencing and iteration of events, or past computations are difficult (or even impossible) to express in model-checkers.

Linear-time logics, such as Linear Temporal Logic (LTL) [111], allow for more intuitive formulae, but sequences and iterations cannot be modelled naturally (although, we can reason about past computations if we consider LTL with past operators).[3] Among other researchers, Wolper [166], Pnueli [136, 140] and Vardi [161] have recognised that linear-time requirements languages are preferred over branching-time ones, but also that both, the power of regular expressions and the flexibility offered by past operators, are necessary to facilitate the formulation of requirements in a natural way.

Interval Temporal Logic (ITL) [121] is a linear-time formalism that is expressive enough to overcome the limitations of point-based logics such as CTL and LTL. ITL models are finite sequences of states, called intervals (ITL has also been extended to handle infinite intervals, see e.g., [64, 128]). ITL has the power of regular expressions, and includes operators akin to concatenation and Kleene-star (hence, sequence and iteration can be intuitively expressed) and past operators. Moreover, quantitative time constraints can also be expressed in ITL, assuming a discrete-time model (this, for instance, has been exploited to specify multimedia documents and their related requirements [41]).

Although there has been a fair amount of research on proof systems for ITL (e.g., [125, 128, 49]), tool support has evolved at a much slower pace [122, 75, 98, 64], hampered by the non-elementary complexity of the decision procedures. In particular, there is still a lack of efficient implementations of decision procedures for propositional subsets of ITL.

This motivates our interest in the Weak Monadic Second-order Theory of 1 Successor (WS1S) and the related tool, MONA [96]. WS1S [54, 66] has the expressive power of regular expressions on finite words, and MONA implements a decision procedure for WS1S, based on a translation

[3] To our knowledge, RT-Spin [158], DT-Spin [35] and Profounder [159] are the only real-time model-checkers that use linear-time requirements languages.

from WS1S formulae to finite automata. MONA uses Binary Decision Diagrams (BDDs) [53, 52] to encode the automata, and implements many optimisations to the translation algorithm. This allows MONA to solve many non-trivial problems [96], despite the non-elementary complexity of WS1S. The expressiveness of WS1S suggests that propositional ITL could be encoded, so MONA could be used (indirectly) as an efficient decision procedure for propositional ITL.

We cannot yet expect real-time model-checkers based on ITL. Nonetheless, exploring to what extent practical tool support can be obtained for ITL, and informing what the strengths and limitations of ITL are as a requirements language for real-time systems, are necessary steps in this direction.

1.2.3 Data in Timed Automata

Most model-checkers for timed automata do not support variables other than in simple data domains, e.g., integers in a known range. Uppaal does better, and offers structured types such as arrays of integers and clocks. However, model-checkers limit variables to bounded domains to preserve a finite-state space. Despite the advantages of fully automatic verification, we may find bounded domains too restrictive for our purposes.

For instance, when modelling communication protocols, we may need to represent unbounded buffers. Given that this would not be possible in the specification language of the model-checker, we would have to settle for some arbitrary buffer size (which is usually small, in order to avoid state-explosion). However, our analysis would sacrifice generality, thus affecting our confidence in the system at hand.

Deductive methods do better in this respect. For instance, Timed I/O Automata use first order logic as the underlying assertion language, where variables in richer, unbounded data domains can be represented [92]. The main drawback of deductive methods is that fully automatic verification is sacrificed to obtain greater expressiveness. Tool support for deductive methods is usually given by theorem provers [107, 90, 163]; however with theorem provers much interaction and expertise are usually required from the user in order to complete the proofs. Another disadvantage of deductive methods is that proving timelock-freedom is particularly difficult [1, 91, 25, 100].

This trade-off between expressive power and automatic verification encourages research into the integration of algorithmic and deductive methods. This motivates our interest in the application of WS1S and MONA (which we have introduced in the previous section) in formal verification of real-time systems.

Among the many applications of MONA, the work of Smith and Klarlund [148] is particularly relevant here. Smith and Klarlund used WS1S, MONA and (untimed) I/O Automata [109] to formally specify and verify a sliding window protocol. This suggests a way to pair

richer specification and requirements languages (e.g., WS1S is expressive enough to represent unbounded sets, and also permits quantification over sets) with automatic verification (MONA is an automatic decision procedure for WS1S).

Following from [148], we aim to obtain a timed notation where specifications are based on WS1S, and verification is assisted by MONA. In addition, we want to apply the semantics of urgency and synchronisation of [33, 39] to yield a safer notation, i.e., one where timelocks can be prevented by construction.

1.3 Contributions

1.3.1 Overall Contribution

This thesis improves the available tool support for formal verification of real-time systems, with the general aim of obtaining more reliable methods. We address specific issues concerning timelock detection and expressiveness of requirements and specification languages. In this respect, we highlight three main contributions.

1. We develop new methods to detect timelocks in timed automata, and implement these methods in a tool to analyse Uppaal specifications. Our methods improve the support provided by model-checkers in general, and that of Uppaal in particular. In this way, we offer better tools to guarantee the sanity of timed automata, increasing the reliability of model-checkers (and that of Uppaal, in particular).

2. We use MONA to implement an efficient decision procedure for PITL, a rich propositional subset of ITL. Thanks to the many optimisations included in MONA, we offer an efficient implementation despite the complexity of PITL. On the one hand, this work improves the tool support that is currently available for propositional subsets of ITL. On the other hand, we close the gap between expressive requirements languages and practical implementations. This is a step towards model-checking requirements languages that are more natural than branching-time logics, which will allow users to better map requirements to formulae (and therefore increase the confidence in verification).

3. We present Discrete Timed Automata, a deductive framework for real-time systems based on WS1S and MONA. WS1S is powerful enough to overcome some of the data limitations in timed automata (e.g., WS1S can represent unbounded structures), while MONA facilitates invariance proofs. In addition, specifications in Discrete Timed Automata are safer, because an important class of timelocks are prevented by construction. Again, this positively affects reliability: specifications are more flexible to better represent the system

under analysis, well-timed specifications are encouraged by the notation, and much of the tedious (and error-prone) tasks in a proof are automatically handled by MONA.

Another contribution of this thesis is a comparison between Timed Automata, PITL and Discrete Timed Automata, which reveals strengths and weaknesses of these formalisms and their related tools (Uppaal and MONA).

This comparison is valuable and far from obvious. For example, unlike Timed Automata and Uppaal, WS1S and MONA were not intended to be used in formal verification of real-time systems. Nonetheless, we show that there are positive aspects in using WS1S and MONA to define a deductive real-time framework. Similarly, decision procedures for propositional ITL are traditionally tableau-based; however, we show the advantages of using WS1S and MONA to achieve efficient implementations based on automata.

This comparison is stated in more detail in Chapters 7 and 8, with Chapter 9 presenting a summary of our main findings. For the time being, let us just show you how it follows from the analysis performed in the thesis.

1. We start by analysing the problems of timelocks in timed automata, and show to what extent timelocks can be prevented and detected by model-checkers. However, other issues also affect timed automata and model-checkers, such as the limited expressiveness of branching-time logics and the restrictive support for data in timed automata models. This motivates our work on PITL, WS1S and MONA.

2. We show that PITL can naturally express requirements that refer to sequences, iterations and past computations, but data is limited to propositional variables, and the interleaving of actions is not easy to model (which limits PITL's suitability as a specification language).

3. On the other hand, Discrete Timed Automata is a natural specification language, and thanks to the expressive power of WS1S, unbounded data domains can be represented. Another advantage of Discrete Timed Automata, with respect to Timed Automata, is that an important class of timelocks is prevented by construction. However, addition and multiplication are restricted in WS1S; thus, certain time constraints are difficult (or even impossible) to express in Discrete Timed Automata.

4. Another limitation of PITL and Discrete Timed Automata, compared with Timed Automata, is that they are restricted to discrete-time models (the advantages of dense-time over discrete-time models are well understood [5]). While discrete-time is inherent to the semantics of PITL, in Discrete Timed Automata it stems from the semantics of WS1S, which is interpreted in the natural numbers.

1.3.2 Improving Timelock Detection in Timed Automata

We distinguish between two classes of timelocks: time-actionlocks and Zeno-timelocks. Time-actionlocks are states where neither the passage of time nor the execution of actions are possible. Zeno-timelocks are states where time cannot pass beyond a certain bound, but actions continue to be performed. Such a distinction is important because different approaches are needed to deal with each class of timelocks. In this thesis, we focus on methods to detect Zeno runs and Zeno-timelocks.

We present a classification of timelocks in Chapter 3, and explain the different consequences that timelocks and Zeno runs have on verification. In addition, we compare timelock detection in Uppaal and Kronos (which are based on model-checking liveness properties) with respect to the strong non-Zenoness property [156], a static check that guarantees absence of Zeno runs.

We are interested in strong non-Zenoness for many reasons. This property can be statically checked, is compositional, holds in most specifications, and implies absence of Zeno-timelocks. Hence, in specifications where strong non-Zenoness holds timelock-freedom corresponds precisely to deadlock-freedom. Therefore, and because deadlock-freedom is usually asserted before any other verification takes place, in most cases, strong non-Zenoness suffices to guarantee timelock-freedom (avoiding the demanding checks of liveness formulae in Uppaal or Kronos).

Nonetheless, strong non-Zenoness can be improved. Compared with the original definition [156], we show that the application of strong non-Zenoness can be weakened to yield a more efficient and comprehensive check, which permits the identification of a bigger class of safe specifications.

On the other hand, we may have to deal with specifications where Zeno runs occur and do not affect verification. However, we still need to guarantee timelock-freedom. In such cases, methods that guarantee absence of Zeno runs, such as strong non-Zenoness, are unsatisfactory: they will rule out specifications as unsafe whether or not they are timelock-free. In response to this problem, we develop static checks that guarantee absence of Zeno-timelocks, but which are insensitive to the occurrence of Zeno runs. We present these checks (including our refined application of strong non-Zenoness) in Chapter 4.

The static checks we present in Chapter 4 are sufficient-only, and so nothing can be concluded from the specification when the checks fail.[4] In safety-critical systems, however, we need to obtain greater confidence in the specification: inconclusive answers are not good enough. In response to this limitation of static checks, we develop sufficient-and-necessary conditions for absence of Zeno-timelocks and Zeno runs, based on reachability analysis. These conditions are

[4]Let p be an assertion such as absence of timelocks or Zeno runs. A check *ck* is considered *sufficient-only* for p if a positive outcome of *ck* implies p, but a negative outcome does not necessarily imply $\neg p$. Correspondingly, a check *ck* is considered *sufficient-and-necessary* for p if the validity of p can always be decided by the outcome of *ck*. We will use this interpretation throughout the thesis.

characterised by reachability formulae obtained from the syntactic structure of loops in the timed automaton.

These reachability-based checks are interesting for a number of reasons. Firstly, they show that timelock-freedom and absence of Zeno runs can be reduced to a combination of static and simple reachability analysis (so far, time-divergence has been characterised only by liveness properties). Secondly, these checks offer sufficient-and-necessary conditions and are available to all model-checkers. In addition, depending on the specification at hand, these checks may be more efficient than similar checks in Uppaal or Kronos, which require more demanding model-checking algorithms. We present our semantic checks in Chapter 5, together with a tool that we have implemented to perform both the static and semantic checks over Uppaal specifications.

1.3.3 An Efficient Decision Procedure for Propositional ITL

Chapter 7 introduces PITL, a propositional subset of ITL with finite quantification, future operators such as *len*, *chop*, *chop-star* and *projection*, and past operators such as *previous* and *since*. With these, PITL can naturally model requirements that refer to quantitative time constraints, sequences and iteration of events, and past computations.

We explain how the semantics of PITL formulae can be expressed in WS1S, which allows us to derive a translation from PITL to WS1S formulae. Briefly, for any PITL formula, a WS1S formula can be derived where propositions and intervals are represented by finite sets in $\mathbb{N}$, and PITL operators are encoded as WS1S operations over sets. MONA constructs an automaton for the resulting WS1S formula and looks for paths to accepting and rejecting states. These paths represent the models and counterexamples of the PITL formula.

We implement a tool, PITL2MONA, which converts PITL formulae to MONA programs and uses MONA (indirectly) as an efficient, automata-based decision procedure for PITL. To our knowledge, this is the first implementation of an automata-based decision procedure for PITL. We show that our implementation is efficient despite the non-elementary complexity of PITL, and that it outperforms the tool LITE [99], a tableau-based implementation of another propositional subset of ITL.

1.3.4 Discrete Timed Automata

Chapter 8 introduces Discrete Timed Automata, a deductive framework for real-time systems, where WS1S is the underlying assertion language, and MONA assists invariance proofs [112].

Discrete Timed Automata is an assertional notation. Specifications consist of a set of WS1S variables, which define the state space, and a set of actions defined through preconditions, deadlines and effects, which are WS1S formulae. We use a distinguished variable $T \in \mathbb{N}$ to denote (discrete) time, which actions can read but not modify: T is assumed to increment

implicitly. Urgency is defined by deadlines as in [33, 39]: actions must be performed no later than the deadline allows.

Deductive verification of invariance properties (a useful class of safety properties) is realised by application of the well-known invariance rule [112]. This is possible by translating discrete timed automata to (untimed) basic transition systems [111]. MONA assists proofs by automatically validating the proof obligations that result from the invariance rule.

Discrete Timed Automata presents a number of convenient features. Thanks to WS1S, one can represent unbounded data structures. Thanks to MONA, proofs that refer to such structures are also facilitated. Thanks to the inclusion of deadlines, not only are we able to model urgency, but we can do so more safely than in other frameworks (both deductive and algorithmic): Deadlines, and parallel composition in Discrete Timed Automata, are defined in such a way that time-actionlocks are prevented by construction. The limitations of Discrete Timed Automata are also discussed in Chapter 8.

1.4 Organization

Throughout the thesis, we offer small examples to illustrate the concepts, and more substantial examples as case studies to evaluate our solutions. Alongside our contributions, we discuss related work, and suggest further research.

We have organized this thesis in four parts. In the first part we present an introduction to the thesis (this chapter); the last part draws conclusions (Chapter 9).

In the second part of the thesis, we study the problem of timelocks in timed automata specifications, and propose techniques to improve the support for timelock-detection that is currently available in real-time model-checkers.

We present the theory of Timed Automata and comment on Uppaal (as a representative model-checker) in Chapter 2. In Appendix A, we comment on reachability analysis and some optimisations implemented in Uppaal. In Chapter 3, we present a classification of timelocks, discuss the different effects of timelocks and Zeno runs on verification, and review existing methods to guarantee timelock-freedom. In Chapter 4 we introduce new static checks to guarantee absence of Zeno-timelocks and Zeno runs. In Chapter 5 we develop sufficient-and-necessary checks for Zeno-timelocks and Zeno runs, which are based on reachability analysis. We also comment on a tool that we have implemented, which performs our static and semantic checks over Uppaal specifications. In Appendix B, we offer some simplifications for the semantic checks.

The third part of the thesis studies the applicability of WS1S and MONA to support expressive requirements languages and deductive frameworks. As we explained before, this analysis is motivated by shortcomings in timed automata and model-checkers. We present WS1S and

MONA in Chapter 6. In Chapter 7, we use MONA as an efficient decision procedure for PITL. Theorems related to the translation from PITL to WS1S are proven in Appendix C. In Chapter 8, we present Discrete Timed Automata, which is based on WS1S and allows MONA to assist invariance proofs.

1.5 Publications

This thesis presents the latest update on a number of the author's publications. The classification of timelocks (Chapter 3), the static checks for non-Zenoness (Chapter 4), and the tool for non-Zenoness detection over Uppaal specifications (Chapter 5), evolved from our initial presentation in [47]. The semantic checks for non-Zenoness (Chapter 5) appear in [46] (which has been recently accepted for publication). Both static and semantic checks refine our presentation in [45]. Chapter 7 extends our results in [72] (e.g., we extended PITL with propositional quantification). The theory of Discrete Timed Automata (Chapter 8) has evolved from our presentations in [71] and [45].

Part II

Timed Automata

Chapter 2

Timed Automata and Uppaal

Abstract. Timed Automata, and the model-checker Uppaal as tool support, are widely used to represent and verify real-time systems. Timed Automata are a simple, yet quite expressive graphical notation, and Uppaal is one of the most developed tools to verify timed automata specifications. In this chapter we present the main elements of the theory of Timed Automata, and we give an overview of Uppaal and real-time model-checking.

This chapter provides necessary background for the remainder of the thesis. Chapters 3, 4 and 5 explain the problem of timelocks in timed automata and model-checkers (and Uppaal in particular), and propose solutions. Then, in Part III, we will study the application of WS1S and MONA (which we will introduce in Chapter 6) to deal with other limitations of timed automata and Uppaal. These limitations concern the expressiveness of the requirements language (Chapter 7), and the support for data variables and prevention of timelocks (Chapter 8).

Organization. We introduce Timed Automata in Section 2.1, and Uppaal in Section 2.2. We comment on related work in Section 2.3. Section 2.4 concludes this chapter with a discussion about limitations of timed automata and Uppaal (and model-checkers, in general). Appendix A offers a more detailed explanation of reachability analysis and particular features of Uppaal.

2.1 Timed Automata

Timed Automata [7] is a formal notation to represent real-time systems, such as embedded controllers and communication protocols (see, e.g., [22] and [70]). Timed Automata are a simple, yet quite expressive graphical notation, which allows fully automatic verification via real-time model-checkers (Uppaal [22], Kronos [169], Red [165], etc).

The literature on timed automata is very rich, and many variations of the original model [7] have been proposed (see, e.g., [85, 156, 8]) and adopted by model-checkers. Here we present a

basic timed automata model that suffices to illustrate the main elements of the theory. This model is similar to Timed Safety Automata of Henzinger et al. [85], and also corresponds closely to the specification language of Uppaal (Section 2.2).

A timed automaton is a finite automaton[1] extended with clocks, clock constraints and synchronisation primitives. A timed automaton represents the possible executions of a real-time system. Locations represent the different states a system goes through; when execution reaches a location in the automaton, the system stays there for a period of time, without performing any visible activity. Transitions represent the possible actions the system may perform; in addition, actions are assumed to occur instantaneously (in the semantics of a timed automaton, time passes only in locations).

A *clock* is a variable in $\mathbb{R}^+$ whose value increases autonomously, representing the passage of (dense) time. The value of all clocks in the automaton increase synchronously, and they do so at the same rate (a global clock is assumed implicit in the model). Performing transitions can also cause some clocks to be reset (in our model, clocks can be reset only to zero). This is specified by annotating transitions with *reset sets*.

Expressions on clocks represent time constraints on the system's behaviour. Clock constraints are attached to transitions in the form of *guards*, which determine when transitions may be performed, and to locations in the form of *invariants*, which determine for how long executions may stay at a particular location. A transition is *enabled* at a given location when both (a) the current valuation[2] satisfies the guard, and (b) the invariant in the target location holds after the transition has been performed (taking into account the current valuation and the reset set).

Enabled transitions are not guaranteed to be performed; instead, the execution may simply remain in the current location, just passing time. Invariants can be used to force the executions of actions at a given time. Executions can stay at a location only as long as the resulting valuations satisfy the invariant; when this period expires (i.e. when time cannot progress any further without invalidating the invariant), the execution must leave the location immediately. When this happens, enabled transitions are regarded as *urgent* and (at least one of them) must be performed for the execution to proceed.

In well-behaved specifications, whenever the execution must leave the current location, another location can be reached that allows time to progress further; any possible execution of the specification allows time to diverge (i.e. time increases ad infinitum). Unfortunately, specifications may contain *timelocks*: states from which no further execution allows time to diverge (we will discuss timelocks in detail in Chapter 3).

[1] In the usual sense of the word, a finite automaton is composed of a finite set of states and labelled transitions. Here, states in the finite automaton will be referred to as *locations*, to avoid confusion with semantic states corresponding to the automaton's executions.

[2] The current clock valuation denotes the value of all clocks at any given state in the execution of a timed automaton.

Complex systems can be represented by a network of automata executing in parallel. Concurrency is modelled by interleaving, and communication is achieved through binary synchronisation (similar to synchronisation in CCS [116]). Transitions in the automata are labelled with *completed actions*, which are performed autonomously; or with *half actions*, which are performed in pairs (i.e., half actions are the synchronisation primitives of timed automata).[3] Half actions can be output actions (e.g., $a!$, where a denotes some action label), or input actions (e.g., $a?$). A pair of output and input actions (in different components) is said to *match* if they refer to the same action label (e.g., $a!$ and $a?$).

Synchronisation may happen only between matching half actions that are enabled. When synchronisation occurs, both parties perform the half actions simultaneously. However, synchronisation does not necessarily occur as soon as both parties are enabled. Executions may choose to "miss" possible synchronisations. Again, this behaviour may be constrained with invariants, but timelocks may occur due to mismatched synchronisations (e.g., when an output action is urgent but the input action is not enabled).

The behaviour of a network corresponds to the interleaving of action and time transitions. An action transition results from some component performing a completed action, or from a pair of components synchronising on half actions. A time transition simply represents a possible delay in the current locations (one current location in every component). The behaviour of a network can be represented, as well, by means of a product automaton: this is a timed automaton that can be constructed syntactically from the components, and can be regarded as the result of parallel composition. Before we formally present syntax and semantics of timed automata, we offer the following example to illustrate the main elements of the model.

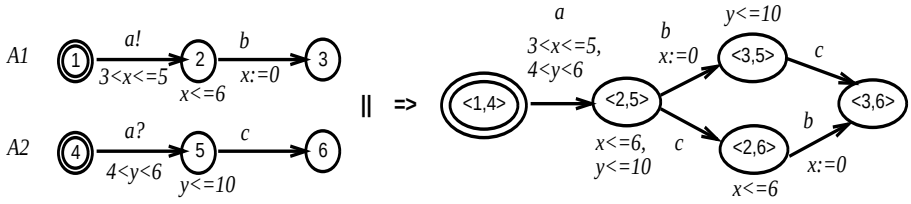


Figure 2.1: A Network of Timed Automata (left) and Its Product Automaton (right)

Figure 2.1 shows a network of automata (*A1* and *A2*) and the corresponding product automaton (right). In the figure, initial locations are distinguished with a double circle, and *true*-invariants are omitted (these invariants do not constrain the passage of time). Clocks are initially set to zero. Reset sets are shown simply as clock assignments.

[3] When confusion cannot arise, we will use the terms "transition" and "action" interchangeably. Similarly, we will refer to transitions labelled with completed actions or half actions simply as "completed actions" and "half actions", respectively.

The guard attached to transition $a!$ $(3 < x \leq 5)$, and the invariant in location 2 $(x \leq 6)$, allow this transition to be performed at any time during the interval $(3, 5]$. However, because the transition is labelled with a half action, $a!$, it can only be performed if the second automaton is willing to synchronise (i.e., perform transition $a?$) during $(3, 5]$. Therefore, the guards attached to $a!$ and $a?$ imply that synchronisation can only occur in the interval $(4, 5]$.

Synchronisation between $a!$ and $a?$ results in transition a (a completed action) in the product automaton, with guards conjoined. If synchronisation occurs (instead, components may remain indefinitely in their locations), then $A1$ may perform transition b at any time, provided $v(x) \leq 6$;[4] similarly, $A2$ may perform transition c as long as $v(y) \leq 10$. Note that, while the network is in locations 2 and 5, the valuation is such that $v(x) = v(y)$.

Consider, now, an execution where $A1$ is still in location 2 when $x = 6$. Time cannot pass, because the invariant in location 2 would be invalidated, and so b must be performed immediately (i.e., b becomes urgent when $v(x) = 6$). However, even at this point, interleaving with other actions is possible: c can be performed before b, although time would not be able to pass until b is performed. This can be seen in the product automaton, if the path $\langle 2, 5\rangle$ c $\langle 2, 6\rangle$ b $\langle 3, 6\rangle$ is executed when $v(x) = 6$ in $\langle 2, 5\rangle$ (at any location vector, the invariant results from conjoining the invariants of the component locations).

2.1.1 Syntax

Basic Notation. $CAct$ is a set of completed actions. $HAct = \{\, a?, a! \mid a \in CAct \,\}$ is a set of half actions. Two half actions, $a?$ and $a!$, can synchronise and generate a completed action a. $Act = HAct \cup CAct$ is the set of all actions. $\mathbb{C}$ is the set of clocks, all of which take values in the positive reals ($\mathbb{R}^{+0}$). CC is a set of clock constraints, described by the following BNF.

$$\phi ::= x \sim c \mid x - y \sim c \mid \phi_1 \wedge \phi_2$$

where $c \in \mathbb{N}$, $x, y \in \mathbb{C}$, $\phi, \phi_1, \phi_2 \in CC$ and $\sim \in \{<, >, =, \leq, \geq\}$ (we will use *true* to denote the constraint $x \geq 0$). $Clocks(\phi)$ is the set of clocks occurring in $\phi \in CC$; $C \subseteq \mathbb{C}$ is the set of clocks of a particular automaton; and CC_C is the set of constraints restricted to C. $\mathbb{V} : \mathbb{C} \to \mathbb{R}^{+0}$ is the space of clock valuations, and $\mathbb{V}_C : C \to \mathbb{R}^{+0}$ is the space of valuations restricted to C. For any $\phi \in CC$ and $v \in \mathbb{V}$, $v \models \phi$ denotes that v satisfies ϕ (equivalently, v is in the solution set of ϕ). For any $\delta \in \mathbb{R}^{+0}$, $v + \delta$ is the valuation s.t. $(v + \delta)(x) = v(x) + \delta$, for all $x \in \mathbb{C}$. For any reset set $r \in \mathcal{P}(\mathbb{C})$, $r(v)$ is the valuation s.t. $r(v)(x) = 0$ for all $x \in r$ and $r(v)(y) = v(y)$ for all $y \notin r$.

[4] For any clock x, $v(x)$ denotes the value of x at any given time.

Timed Automaton. A timed automaton is a tuple $A = (L, l_0, TL, C, T, I)$, where the elements are defined as follows.

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- $TL \subseteq Act$ is a finite set of transition labels.
- $C \subseteq \mathbb{C}$ is a finite set of clocks.
- $T \subseteq L \times TL \times CC_C \times \mathcal{P}(C) \times L$ is a transition relation. Transitions $(l, a, g, r, l') \in T$ are usually denoted,

  $$l \xrightarrow{a,g,r} l'$$

  where $a \in TL$ is the *action*, $g \in CC_C$ is the *guard* and $r \in \mathcal{P}(C)$ is the *reset set*.
- $I : L \to CC_C$ is a mapping that associates invariants with locations.

A *network of timed automata* is denoted $|A = |\langle A_1, \ldots, A_n \rangle$, where A_i is a timed automaton. Usually, we would expect components to specify only possible synchronisations: if a component includes a half action $a!$, then another component should include the complementary action, $a?$.

Product Automaton. Consider a network $|A = |\langle A_1, \ldots, A_n \rangle$, where $A_i = (L_i, l_{i,0}, TL_i, C_i, T_i, I_i)$. Let u and u' denote location vectors in $L_1 \times \cdots \times L_n$ (e.g., $u = \langle u_1, \ldots, u_n \rangle$). We use the substitutions

$$\langle u_1, \ldots, u_j, \ldots, u_n \rangle [l \to j] \quad \text{for} \quad \langle u_1, \ldots, u_{j-1}, l, u_{j+1}, \ldots, u_n \rangle; \text{ and}$$

$$u[l_1 \to i_1, \ldots, l_m \to i_m] \quad \text{for} \quad u[l_1 \to i_1] \ldots [l_m \to i_m]$$

The product automaton for $|A$ is defined as the timed automaton Π,

$$\Pi = (L, l_0, TL, C, T, I)$$

where

- L is the smallest set of location vectors which includes l_0 and is closed under the transition relation T,

  $$L = \{ l_0 \} \cup \{ u' \mid \exists\, u \in L,\ a \in TL,\ g \in CC_C,\ r \in \mathcal{P}(C) \,.\, u \xrightarrow{a,g,r} u' \in T \}$$

- l_0 is the initial location vector, $l_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$;
- TL is the set of actions labelling transitions in T,

$$TL = \{\, a \mid \exists\, u, u' \in L,\ g \in CC_C,\ r \in \mathcal{P}(C)\,.\, u \xrightarrow{a,g,r} u' \in T \,\}$$

- C is the set of clocks of the product automaton, $C = \bigcup_{i=1}^{n} C_i$;
- T is the transition relation defined by the following rules ($1 \leq i \neq j \leq n$),

$$(P1)\ \frac{u_i \xrightarrow{a?,g_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,r_j}_j l'}{u \xrightarrow{a,g_i \wedge g_j,r_i \cup r_j} u[l \to i, l' \to j]} \qquad (P2)\ \frac{u_i \xrightarrow{a,g,r}_i l \quad a \in CAct}{u \xrightarrow{a,g,r} u[l \to i]}$$

- I is the function which associates invariants to location vectors,

$$I(\langle u_1, ..., u_n \rangle) = \bigwedge_{i=1}^{n} I_i(u_i)$$

Rule (P1) adds a completed action in the product for every possible synchronisation between components. The guard and reset set in this action correspond to the conjunction of guards and the union of the reset sets in the synchronising transitions, respectively. This rule asserts that synchronisation is only possible if both parties are enabled. Rule (P2) denotes the interleaving of completed actions.

The following concepts will become useful when we analyse techniques for detection of time-locks and Zeno runs in Chapters 3, 4, and 5. We present these here to add to the general terminology related to timed automata, as used in this thesis.

Loops. Let A be a timed automaton. A *simple loop* is an elementary cycle in A; i.e., a sequence of transitions,

$$l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} l_2 \ \cdots\ l_{n-1} \xrightarrow{a_n,g_n,r_n} l_n$$

where $l_0 = l_n$ and $l_i \neq l_j$ for all $0 \leq i \neq j < n$. A *non-simple loop* is a strongly connected subgraph[5] of A, which is not itself a simple loop. By definition, a non-simple loop contains at least (all the transitions of) two simple loops.

We will denote loops as a sequence of actions that starts and ends in the same location. In the case of non-simple loops, this sequence will contain repeating actions. This sequence of actions, both for simple and non-simple loops, is used just for notational purposes, and is

[5] A directed graph is strongly connected if there exists a path between any two nodes.

not intended to reflect a traversal of the loop during the execution of the timed automaton. In addition, unless we explicitly restrict their scope, our definitions and results apply to both simple and non-simple loops.[6]

By way of example, Figure 2.2(i) shows two simple loops, $\langle a, b\rangle$ and $\langle c, d\rangle$; and one non-simple loop, $\langle a, c, d, b\rangle$. Similarly, Figure 2.2(ii) depicts two simple loops, $\langle e, f\rangle$ and $\langle g, h, f\rangle$, and one non-simple loop, $\langle e, f, g, h, f\rangle$.

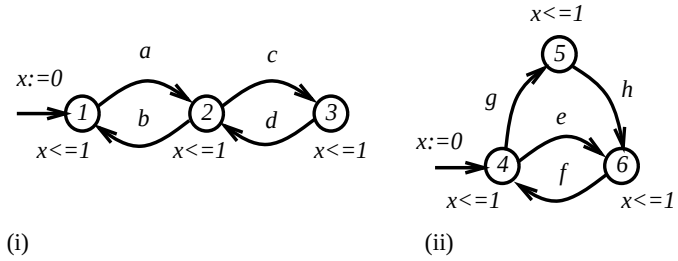


Figure 2.2: Simple and Non-simple Loops

Let A be a timed automaton, and lp a loop in A. We define the following sets. $\mathit{Loops}(A)$ is the set of all loops in A. $\mathit{SimpleLoops}(A) \subseteq \mathit{Loops}(A)$ is the set of all simple loops in A. $\mathit{Loc}(lp)$ is the set of all locations of lp; $\mathit{Clocks}(lp)$ is the set of all clocks occurring in any invariant of lp; $\mathit{Trans}(lp)$, $\mathit{Guards}(lp)$ and $\mathit{Resets}(lp)$ are, respectively, the sets of all transitions of lp, all guards of lp, and all clocks that are reset in lp; and $\mathit{Act}(lp)$ is the set of all actions labelling transitions in lp. In the following definitions, we use $\sqsubseteq$ ($\sqsupseteq$) to denote any element in $\{<, =, \leq\}$ ($\{>, =, \geq\}$); and we use $t \in \phi$ to denote that t is a conjunct of formula ϕ.

Half Loops, Completed Loops and Matching Loops. lp is a *half loop* if it contains at least one transition labelled with a half action (formally, $\mathit{HAct} \cap \mathit{Act}(lp) \neq \emptyset$). lp is a *completed loop* if all its transitions are labelled with completed actions (formally, $\mathit{Act}(lp) \subseteq \mathit{CAct}$). Two half loops lp_1 and lp_2 (in different network components) are referred to as *matching loops* if they contain at least a pair of matching half actions (formally, $\exists\, a?, a! \in \mathit{HAct}.\ a? \in \mathit{Act}(lp_1)\ \wedge\ a! \in \mathit{Act}(lp_2)$).

Bounded from Below (clock). Given a clock constraint ϕ, a clock x is *bounded from below* in ϕ, if $x \sqsupseteq c \in \phi$ or $x - y \sqsupseteq c \in \phi$, where y is a clock and $c > 0$. By extension, a clock is bounded from below in a given location or transition if it is bounded from below in the location's invariant, or in the transition's guard, respectively.

[6] Our reasoning about simple and non-simple loops (as found in Chapters 3, 4 and 5) will be concerned with the structure of the loops (the subgraph formed by locations and transitions), and not with the possible ways in which the loops can be traversed by executions of the timed automaton.

Bounded from Above (clock). Given a clock constraint ϕ, a clock x is *bounded from above* in ϕ if $x \sqsubseteq c \in \phi$; or $x - y \sqsubseteq c \in \phi$; or $y - x \sqsupseteq c \in \phi$ (where the clock y is bounded from above in ϕ and $c > 0$). By extension, a clock is bounded from above in a given location or transition if it is bounded from above in the location's invariant, or in the transition's guard, respectively.

Smallest Upper Bound (clock). Let lp be a loop and x a clock in lp, where at least one invariant in the loop contains a conjunct $x \sqsubseteq c$ ($c > 0$). We define $c_{\min}(x, lp) \in \mathbb{N}$ to be the smallest upper bound for x occurring in any invariant in lp, i.e., $c_{\min}(x, lp) \leq c'$, for any conjunct $x \sqsubseteq c'$ occurring in any invariant of the loop ($c' > 0$).

2.1.2 Semantics

We formalise, here, the behaviour of a timed automaton in terms of a timed transition system (TTS) [31, 45]. We assume that the automaton contains only completed actions, thus, its behaviour can be completely determined from its own structure. With this approach, the behaviour of a network corresponds to the TTS of the product automaton.[7]

Let $A = (L, l_0, TL, C, T, I)$ be a timed automaton where all actions are completed actions ($TL \subseteq CAct$). The behaviour of A is represented by the timed transition system $TS_A = (S, s_0, Lab, T_S)$, which is defined as follows.

- $S \subseteq L \times \mathbb{V}_C$ is the set of reachable states in the executions of A; i.e., the smallest set of states which includes s_0 and is closed under the transition relation T_S (a state is of the form $s = [l, v]$, where l is a location in A and v is a clock valuation),

$$S = \{\, s_0 \,\} \;\cup\; \{\, s' \mid \exists\, s \in S, \gamma \in Lab \,.\, s \stackrel{\gamma}{\Longrightarrow} s' \in T_S \,\}$$

- $s_0 = [l_0, v_0]$ is the initial state, where l_0 is the initial location in A, and v_0 is the initial valuation, which sets all clocks to 0;

- $Lab = TL \cup \mathbb{R}^+$ is the labels for transitions in T_S;

- $T_S \subseteq S \times Lab \times S$ is the transition relation. Transitions can be of one of two types: action transitions (*actions*), e.g. (s, a, s'), where $a \in TL$, or time transitions (*delays*), e.g. (s, δ, s'), where $\delta \in \mathbb{R}^+$ and the passage of δ time units is denoted. Transitions are denoted,

$$s \stackrel{\gamma}{\Longrightarrow} s'$$

[7] Bengtsson and Yi [21] present a TTS-based semantics for networks of timed automata directly in terms of the component automata. This is equivalent to ours.

where $\gamma \in Lab$. The transition relation is defined by the following inference rules.

$$\text{(R1)}\ \frac{l \xrightarrow{a,g,r} l' \quad v \models g \quad r(v) \models I(l')}{[l,v] \xRightarrow{a} [l', r(v)]} \qquad \text{(R2)}\ \frac{\forall\, \delta' \leq \delta\,,\ (v+\delta') \models I(l)}{[l,v] \xRightarrow{\delta} [l, v+\delta]}$$

Note that rules (R1) and (R2) represent the interleaving of actions with the passage of time. A transition $l \xrightarrow{a,g,r} l'$ is said to be *enabled* in the state $[l,v]$, if the current valuation satisfies the guard ($v \models g$), and performing the transition does not invalidate the invariant of the target location ($r(v) \models I(l')$).

Runs. A *run* is a path ρ in the automaton's TTS,

$$\rho \triangleq s_1 \xRightarrow{\gamma_1} s_2 \xRightarrow{\gamma_2} s_3 \xRightarrow{\gamma_3} \ldots$$

where $s_i \in S$ and $\gamma_i \in Act \cup \mathbb{R}^+$ ($i \in \mathbb{N},\ i \geq 1$), such that ρ ends in some state $s_n \in S$ (if ρ is finite). We use $\rho \subseteq \rho'$ to denote that the sequence ρ is a prefix of ρ'. $Runs(s)$ and $FiniteRuns(s) \subseteq Runs(s)$ denote the set of all runs starting from s, and the set of all finite runs starting from s, respectively. We use $s \xRightarrow{\gamma}_{\rho} s'$ to denote that $s \xRightarrow{\gamma} s'$ is performed at some point in ρ. Similarly, $s \in \rho$ denotes that s is reachable in ρ; $s \xRightarrow{*}_{\rho} s'$ denotes that s' is reachable from s in ρ; and $s \xRightarrow{*} s'$ denotes that s' is reachable from s (equivalently, $\exists\, \rho \in Runs(s).\, s' \in \rho$).

$Trans(\rho)$ and $Trans^{\infty}(\rho) \subseteq Trans(\rho)$ denote the set of all automata transitions visited by ρ, and the set of all transitions that are visited infinitely often by ρ, respectively. Formally,

$$\begin{aligned} Trans(\rho) &= \{\, l \xrightarrow{a,g,r} l' \mid \exists\, v.\, v \models g \wedge [l,v] \xRightarrow{a}_{\rho} [l', r(v)] \,\} \\ Trans^{\infty}(\rho) &= \{\, l \xrightarrow{a,g,r} l' \mid \forall s \in \rho.\, \exists\, v.\, s \xRightarrow{*}_{\rho} [l,v] \wedge v \models g \wedge [l,v] \xRightarrow{a}_{\rho} [l', r(v)] \,\} \end{aligned}$$

Regarding loops, we say that a run ρ *visits* a loop lp if all transitions of lp occur in ρ (not necessarily consecutively); i.e., if $Trans(lp) \subseteq Trans(\rho)$. More interestingly, we will be concerned with those runs that visit a certain loop *infinitely often*. Thus, we introduce next the concept of *covering runs*, and offer the related Lemma 2.1.

Covering Runs. Let lp be a loop, and ρ an infinite run. We say that ρ *covers* lp if it visits lp infinitely often. Formally, ρ covers lp if $Trans(lp) \subseteq Trans^{\infty}(\rho)$. We use $CoveringRuns(s, lp)$ to denote the set of all runs starting from s that cover lp.

LEMMA **2.1.** *If actions occur infinitely often in a run, then that run covers some simple loop.*

Proof. Let ρ be a run where actions occur infinitely often. Automata transitions are finite, so ρ must visit a (simple or non-simple) loop infinitely often. Every non-simple loop contains a simple loop. Hence, even if ρ visits a non-simple loop infinitely often, it must necessarily visit a simple loop infinitely often. □

Consider again, for instance, the non-simple loop $\langle e, f, g, h, f\rangle$ in Figure 2.2(ii). Any run where the sequences of actions (1) g, h, f and (2) e, f occur infinitely often is considered to cover the non-simple loop $\langle e, f, g, h, f\rangle$ (and, necessarily, to cover both simple loops $\langle g, h, f\rangle$ and $\langle e, f\rangle$). On the other hand, a run that only visits e and f infinitely often, will be considered to cover the simple loop $\langle e, f\rangle$, but not to cover the non-simple loop $\langle e, f, g, h, f\rangle$ (even if the run visits g and h a finite number of times). In addition, note that the definition of covering run is not concerned with the order in which transitions are visited (e.g., if there are many "entry points" to the loop, different traversals may be possible).

Regarding the time spent by executions, we define $delay(\rho)$ to be the limit of the sum of all delays occurring in ρ (if the limit exists), or ∞ otherwise. A run ρ is *divergent* if $delay(\rho) = \infty$ (otherwise, the run is *convergent*). A timed automaton may exhibit runs that cannot be extended to divergent runs, and runs where actions occur infinitely often in a finite period of time (which we call *Zeno runs*); none of these runs correspond to natural executions. We use $ZRuns(s) \subseteq Runs(s)$ to denote the set of Zeno runs starting from s.[8]

Chapter 3 will discuss Zeno runs and timelocks, their consequences to verification and ways to detect them (or avoid them). For the time being, let us illustrate the semantics of a timed automaton with the following example.

Example. Figure 2.3 illustrates the behaviour of a timed automaton. Executions may remain in location 1 for a maximum of 5 time units. Transition a can be performed at any time during $[2, 5]$, and is urgent when $v(x) = 5$. The automaton may perform b at any time during $[0, 3]$; thereafter (location 2), it can perform c at any time. The figure (right) shows part of the TTS that represents the behavior of the automaton (vertical arrows denote time transitions).

[8] Non-Zeno runs are also known as *finitely variable*; e.g. [85, 144].

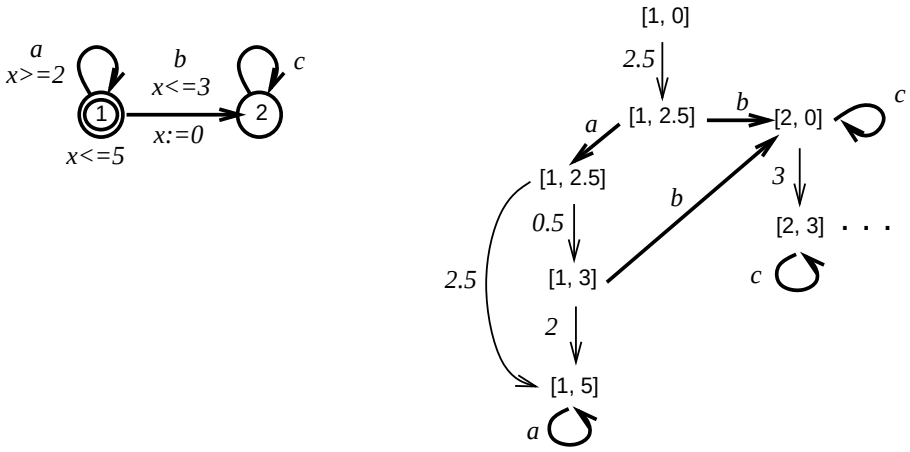


Figure 2.3: A Timed Automaton and (a fragment of) its Timed Transition System

The automaton contains several Zeno runs, generated by transitions a and c. For example,

$$\rho_1 = [1,0] \xrightarrow{2.5} [1,2.5] \xrightarrow{a} [1,2.5] \xrightarrow{0.5} [1,3] \xrightarrow{b} [2,0] \cdots$$

$$\rho_2 = [1,0] \xrightarrow{2.5} [1,2.5] \xrightarrow{b} [2,0] \xrightarrow{c} [2,0] \xrightarrow{3} [2,3] \xrightarrow{c} [2,3] \cdots$$

$$\rho_3 = [1,0] \xrightarrow{2.5} [1,2.5] \xrightarrow{a} [1,2.5] \xrightarrow{2.5} [1,5] \xrightarrow{a} [1,5] \xrightarrow{a} [1,5] \cdots$$

where, for every state $s = [l, v]$, l denotes a location, and v denotes the value of x. The run ρ_2 may represent one of them: once in location 2, the automaton may continuously perform c without engaging in time transitions. The run ρ_3 is another Zeno run, because a is performed infinitely often when $v(x) = 5$. However, ρ_2 and ρ_3 lead to different scenarios. Once the automaton is in location 2 it may perform c infinitely often, but time is allowed to diverge: From any state reached in location 2, a time transition is always enabled. On the other hand, if the automaton is in location 1 when $v(x) = 5$, the invariant prevents time from passing any further. The only enabled transition, a, is then performed infinitely often in a finite period of time. Effectively, the state $s = [1, v]$ $(v(x) = 5)$ is a timelock.

2.1.3 Example: A Multimedia Stream

Figure 2.4 depicts a simple abstraction of the flows of data in a multimedia system (see, e.g., [28]). A *Source* process generates a continuous sequence of packets (representing the transmission of an audio or video stream), which are relayed by the *Channel* to a *Sink* process, which in turn displays the packets. Actions *sourceOut*, *sinkIn* and *play*, transfer packets from the *Source* to the *Channel*, from the *Channel* to the *Sink*, and display them at the *Sink*, respectively.

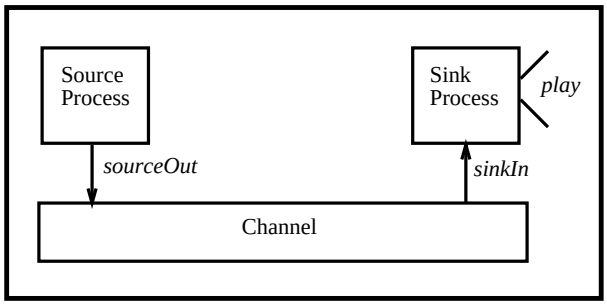


Figure 2.4: A Multimedia Stream (from [44])

As is usually the case for multimedia streams, the flows of data must observe certain timing constraints.[9] For the stream in Figure 2.4, we will assume the following settings. The *Channel* is reliable and does not lose packets. The *Source* transmits a packet every 50 ms. Packets arrive at the *Sink* between 80 ms and 90 ms after being sent; this specifies the latency of the *Channel*. Whenever the *Sink* receives a packet, it needs 5 ms to process it (*play*) before it can accept the next packet. In addition, we assume that the actual contents of the packets are irrelevant to the protocol.

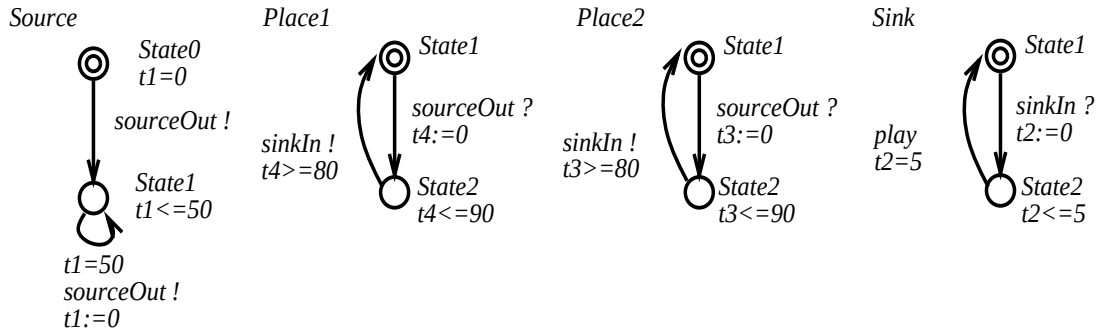


Figure 2.5: Timed Automata Specification of the Multimedia Stream

Figure 2.5 shows a possible timed automata specification for the multimedia stream (this specification was first presented in [44]). The specification consists of four components: *Source*, *Place1*, *Place2* and *Sink*. The *Source* synchronises with either *Place1* or *Place2*, representing that a packet is delivered to the *Channel*. The *Channel* process itself is viewed as a buffer that stores the packets for a period of time (the latency of the *Channel*) before delivering them to the *Sink*. The automata *Place1* and *Place2* represent the *Channel* as a 2-place buffer: giving the rate of *Source* and the latency of the *Channel*, there is always at most 2 packets travelling through the *Channel* at any given time.

[9] For example, when a movie is played, subtitles should be displayed synchronised with the image stream.

The initial location in the *Source*, *State0*, is annotated with the invariant $t1 = 0$ to ensure that the first packet is sent immediately (via a *sourceOut!* action). In *State1*, *sourceOut!* is enabled (and urgent) once every 50 ms. When synchronisation between *sourceOut!* (*Source*) and *sourceOut?* (*Place1*) happens, *t4* is reset and *Place1* moves to *State2*. Simultaneously, the *Source* moves to *State1* if it was in *State0*, or resets *t1* and remains in *State1*. Alternatively, the *Source* automaton may synchronise with *Place2*.

After *Place1* has synchronised with the *Source* (i.e., received a packet), it cannot do so with the *Sink* (i.e., deliver the packet) until at least 80 ms have passed (the same occurs to *Place2*). At the latest, this synchronisation would occur after 90 ms have passed. These delays effectively model the latency of the *Channel*. Note, as well, that a new synchronisation between *Place1* and *Source* cannot occur before *Place1* and *Sink* have synchronised. In other words, *Place1* is not always ready to accept the next packet from the *Source*: the *Source* is faster than *Place1* (the same reasoning applies to *Place2*).

However, verification would confirm that some *sourceOut?* action, either in *Place1* or in *Place2*, is always enabled when *sourceOut!* becomes urgent in the *Source* (otherwise, and because of the urgency of *sourceOut!*, the specification would contain a timelock). Finally, note that *Sink* synchronises with *Place1* or *Place2* by offering a *sinkIn?* action. The *Sink* needs exactly 5 ms to play a packet, before it is ready to accept the next one.

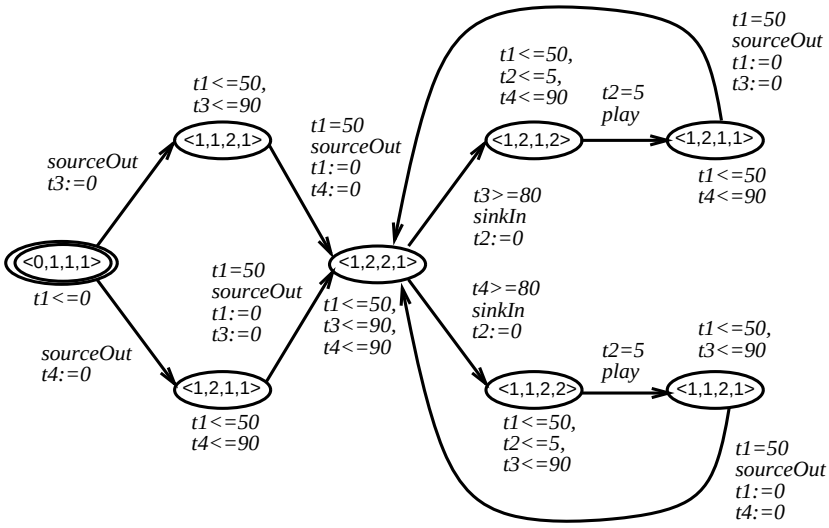


Figure 2.6: Product Automaton For The Multimedia Stream

Figure 2.6 shows part of the product automaton for this network. By way of example: location $\langle 0, 1, 1, 1\rangle$ corresponds to *Source.State0*, *Place1.State1*, *Place2.State1* and *Sink.State1*; transition *sourceOut*, from $\langle 0, 1, 1, 1\rangle$ to $\langle 1, 2, 1, 1\rangle$, results from synchronisation between *sourceOut!* in *Source.State0* and *sourceOut?* in *Place1.State1* (a similar transition exists from $\langle 0, 1, 1, 1\rangle$ to $\langle 1, 1, 2, 1\rangle$, which corresponds to initial synchronisation between *Source* and *Place2*); and the

invariant in location $\langle 1, 2, 1, 1 \rangle$, $t1 \leq 50 \wedge t4 \leq 90$, corresponds to the conjunction of invariants in *Source.State1*, *Place1.State2*, *Place2.State1* and *Sink.State1*.

2.2 Uppaal: A Model-checker for Timed Automata

Uppaal [22] is a tool for reachability analysis of networks of timed automata. Uppaal has been successfully applied in the verification of communication protocols and embedded systems, and is under continuous development (see [22] for a description of case studies and other tools based on Uppaal). Among real-time model-checkers, Uppaal represents the state-of-the-art thanks to its efficient reachability engine, a rich modelling language, and a well-developed graphical interface (this features a built-in simulator).

2.2.1 The Specification Language

Uppaal's modelling language is an extension of the basic timed automata model presented in this chapter. Here we discuss a few relevant features of the language. The full syntax can be obtained from the online tutorial included in the tool.[10] In what follows, bear in mind that *assignments* and *channels* in Uppaal play the role of reset sets and half actions, respectively.

Variables. Data variables (in bounded domains) can be declared, including integers and arrays of integers and clocks. Variables can be declared local to components, or shared by all components in the network. Variables are initialised when declared, and can be assigned new values when transitions are performed. Guards, invariants and assignments may contain expressions on variables (with certain restrictions), written in C-like syntax (e.g., the auto-increment expression `i++` can be used in assignments).

Guards, Invariants and Clock Resets. Both guards and invariants are conjunction of atomic constraints, which refer to single clocks and data variables. Disjunction and negation can be used in subexpressions, but guards and invariants must be conjunctions at the top level. Expressions on clocks denote lower and upper bounds on clocks and clock differences, where the bound is given by an integer expression. However, clock expressions in invariants are restricted to upper bounds (on single clocks, or difference between clocks). Clocks can be reset to any integer expression (including data variables), but they cannot be assigned to other clocks.

[10] `http://www.uppaal.com`

Urgent and Committed Locations. Locations in Uppaal's timed automata can also be declared as *urgent* or *committed*, in addition to the default type [20]. Both, urgent and committed locations, block the passage of time as soon as they are entered (thus making outgoing actions, urgent). Committed locations, however, further restrict the interleaving of actions. When one or more components are in urgent locations, time cannot pass but any component is allowed to perform (enabled) actions, even those components that are not in urgent locations. On the other hand, if one or more components are in committed locations, then only these components are allowed to progress. When interleaving is not necessary, committed locations reduce the state-space generated during verification [19].

Synchronisation. Apart from binary synchronisation (as in the basic timed automata model of this chapter), Uppaal supports *urgent* and *broadcast* synchronisation. Synchronisation on urgent channels is binary, and occurs as soon as both parties are enabled. In transitions with urgent channels, guards cannot refer to clocks. This restriction prevents the occurrence of disjunctions of constraints during reachability analysis. The problem with disjunctions is that they cannot be represented efficiently, and "splitting" them into zones (conjunctions) is a time-consuming operation. A broadcast channel allows a component (output action) to synchronise with more than one component (input action) at the same time. Broadcast synchronisation is nonblocking: A given output action, say $a!$, where a is declared as a broadcast channel, can be performed even when input actions (`a?`) are not enabled. Multiway synchronisation (e.g., as in [169]), and other synchronisation modes, can be implemented with committed locations [22].

2.2.2 The Requirements Language and Verification

Uppaal is a model-checker biased towards reachability analysis, where correctness properties are expressed as formulae in a subset of CTL [58]. Uppaal constructs a *reachability graph*, which is a finite-state abstraction describing the semantics of the network.

A node in the reachability graph represents a pair (l, Z), where l is a location vector and Z, called a *zone*, is a conjunction of clock constraints. Informally, the pair (l, Z) is a symbolic state in the execution of the network, denoting a (typically infinite) number of reachable states $[l, v],\ v \models Z$. For every node (l, Z), and every transition a in the network from l to l' that is enabled at some $v \models Z$, there is a successor node (l', Z') denoting all states that are reachable in l' after a is performed (and letting some time pass in l').

Uppaal does not construct the product automaton a priori; instead, it generates the symbolic states on demand until the fragment of the graph (generated so far) suffices to determine the satisfiability of the property. We explain the fundamentals of reachability analysis in Appendix A, where we also give more details of Uppaal's optimisations.

Uppaal's requirements language is limited to a subset of CTL that can be verified very efficiently using reachability analysis, and which is expressive enough to characterise many useful properties. Safety properties, for instance, can be decided by checking that every reachable state is "correct" with respect to some state formula (or alternatively, that a "bad" state is unreachable). Bounded-response properties are usually of the form "whenever a ϕ-state is reached, a ψ-state is reached in every possible execution thereafter, within n time units". In Uppaal, bounded-response can be verified via a *leads-to* formula, which involves nested reachability checks, or alternatively as a safety property [22], or by *test automata* [2].

The following BNF describes the formulae that can be expressed in Uppaal's requirements language.

$$\begin{aligned} \varphi &::= \texttt{E<>}\,\phi \mid \texttt{A[]}\,\phi \mid \texttt{E[]}\,\phi \mid \texttt{A<>}\,\phi \mid \phi_1 \texttt{ --> } \phi_2 \\ \phi &::= \mathit{loc} \mid \mathit{cc} \mid \texttt{deadlock} \mid \texttt{not } \phi \mid \phi_1 \texttt{ and } \phi_2 \mid \phi_1 \texttt{ or } \phi_2 \mid \phi_1 \texttt{ => } \phi_2 \end{aligned}$$

where φ is a top level Uppaal formula; ϕ, ϕ_1, ϕ_2 are state formulae; *loc* refers to a location in the network and *cc* is an atomic constraint on clocks, clock differences or variables.

The semantics of the requirements language can be given in terms of the reachability graph. State formulae characterise the symbolic states of interest. For example, the state formula

```
A.loc1 and (x<=1 or y>x)
```

is satisfiable by any symbolic state (l, Z) such that $l =$ `A.loc1` and Z satisfies either $v(\texttt{x}) \leq 1$ or $v(\texttt{y}) > v(\texttt{x})$. The formula `deadlock` is satisfiable in (l, Z) if there exists $s = [l, v] \in (l, Z)$ where no action is enabled. Top-level formulae, with `E<>`, `A[]`, `E[]`, `A<>` and `-->`, represent paths in the reachability graph (i.e., executions of interest). Operators `E` and `A` refer to some path, or to all paths in the graph, respectively. In turn, `<>` and `[]` refer to some state, or to all states in a path (or all paths, depending on the preceding `E`/`A` operator).

- `E<>` ϕ holds if there exists a reachable state satisfying ϕ.
- `A[]` ϕ holds if every reachable state satisfies ϕ.
- `E[]` ϕ holds if there exists a path in the graph where every state satisfies ϕ.
- `A<>` ϕ holds if there exists a state, along every path in the graph, which satisfies ϕ.
- The *leads-to* formula, ϕ_1`-->`ϕ_2, is equivalent to `A[]`$(\phi_1$ `=>` `A<>`$\phi_2)$. The formula ϕ_1`-->` ϕ_2 holds if, whenever a ϕ-state is reached, a ψ-state is reachable in every possible execution thereafter (leads-to formulae are the only form of nested modalities expressible in Uppaal).

Note, that the semantics of `E<>` ϕ, `A[]` ϕ, `E[]` ϕ, `A<>` ϕ and ϕ_1 `-->` ϕ_2 correspond, respectively, to that of CTL formulae `EF` ϕ, `AG` ϕ, `EG` ϕ, `AF` ϕ and `AG`$(\phi_1 \Rightarrow$ `AF` $\phi_2)$ [58]. Figure 2.7 illustrates this interpretation (where black nodes represent states in the reachability graph that satisfy ϕ).

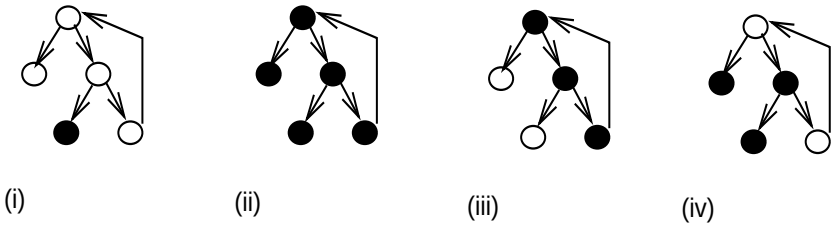


Figure 2.7: Semantics of `E<>` ϕ (i), `A[]` ϕ (ii), `E[]` ϕ (iii), and `A<>` ϕ (iv)

Example: A Safety Property for the Multimedia Stream. Section 2.1.3 (Figure 2.5) described a timed automata specification for a multimedia stream, where the *Channel* was represented by two automata *Place1* and *Place2*. Then, if two automata suffice to correctly implement the *Channel*, synchronisation between *Source* and either *Place1* or *Place2* should always be possible. In other words, it should not be possible to reach state where *Source* must send a packet but neither *Place1* nor *Place2* can accept it; this is expressed by the following safety property.

```
A[] not ((t1==0 or t1==50) and Place1.State2 and Place2.State2)
```

2.2.3 Timed Automata with Urgent and Committed Locations

Section 2.2.1 explained the semantics of urgent and committed locations in Uppaal specifications. Here, we show a simple extension of our basic timed automata model that considers urgent and committed locations. We also show that, from a network of extended automata, we can obtain an equivalent single automaton in the basic model (i.e., without urgent or committed locations). The construction is syntactic and resembles that of the product automaton for the basic model (see Section 2.1.1), but preserves the semantics of urgent and committed locations.

Extended Timed Automata. An extended timed automaton is a tuple $A = (L, l_0, TL, C, T, I)$, where

- L is partitioned in three sets, $L = NonUrg \cup Urg \cup Comm$, with $Urg \subseteq L$ and $Comm \subseteq L$ being the (possibly empty) sets of urgent and committed locations;
- $I(l) \triangleq true$, for all $l \in Urg \cup Comm$; and
- the rest of the elements of A are defined as in our basic timed automata model (Section 2.1.1).

Interpreting Extended Automata in the Basic Model. We define below a process of *augmentation*, which resets a new clock z every time an urgent or committed location is to be entered. Let $A = (NonUrg \cup Urg \cup Comm, l_0, TL, C, T, I)$ be an extended timed automaton and $t \triangleq l \xrightarrow{a,g,r} l'$ a transition in T. The transformation $augment(t, z)$, for transition t and clock z, is defined as follows.

$$augment(t, z) \triangleq \begin{cases} l \xrightarrow{a,g,r \cup \{z\}} l' & \text{if } l' \in Urg \cup Comm \\ t & \text{otherwise} \end{cases}$$

The extended automaton A can then be transformed into a basic automaton $A' = (NonUrg \cup Urg \cup Comm, l_0, TL, C \cup \{z\}, T', I')$ where

- $z \notin C$;
- $I'(l) \triangleq z \leq 0$, for all $l \in Urg \cup Comm$; and
- $T' = \{\ augment(t, z) \mid t \in T\ \}$.

Consider a network of extended timed automata $|A = |\langle A_1, \ldots, A_n\rangle$, where $A_i = (NonUrg_i \cup Urg_i \cup Comm_i, l_{i,0}, TL_i, C_i, T_i, I_i)$. Let $|A' = |\langle A'_1, \ldots, A'_n\rangle$ be the network of basic automata obtained from $|A$ by the process of augmentation. We have, then, $A'_i = (NonUrg_i \cup Urg_i \cup Comm_i, l_{i,0}, TL_i, C_i \cup \{z_i\}, T'_i, I'_i)$. Let u and u' denote location vectors in $L_1 \times \cdots \times L_n$ (e.g., $u = \langle u_1, \ldots, u_n\rangle$). We use the substitutions $u[l \rightarrow j]$ and $u[l \rightarrow i, l' \rightarrow j]$ as explained in Section 2.1.1.

From $|A'$, we can build a single basic automaton, Π, which is semantically equivalent to $|A$ (w.r.t. Uppaal's semantics for urgent and committed locations). We define Π as follows.

$$\Pi = (L, l_0, TL, C, T, I)$$

where

- L is the smallest set of location vectors which includes l_0 and is closed under the transition relation T,

$$L = \{\, l_0 \,\} \cup \{\, u' \mid \exists\, u \in L,\ a \in TL,\ g \in CC_C,\ r \in \mathcal{P}(C)\,.\, u \xrightarrow{a,g,r} u' \in T \,\}$$

- l_0 is the initial location vector, $l_0 = \langle l_{1,0}, \ldots, l_{n,0}\rangle$;
- TL is the set of actions labelling transitions in T,

$$TL = \{\, a \mid \exists\, u, u' \in L,\ g \in CC_C,\ r \in \mathcal{P}(C)\,.\, u \xrightarrow{a,g,r} u' \in T \,\}$$

- C is the set of clocks, $C = \bigcup_{i=1}^{n} (C_i \cup \{z_i\})$;
- T is the transition relation defined by the following rules ($1 \leq i \neq j \leq n$, $1 \leq k \leq n$),

$$(P^C1)\ \frac{u_i \xrightarrow{a?,g_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,r_j}_j l' \quad (u_i \in Comm_i \ \vee \ u_j \in Comm_j)}{u \xrightarrow{a,g_i \wedge g_j,r_i \cup r_j} u[l \rightarrow i, l' \rightarrow j]}$$

$$(P^C2)\ \frac{u_i \xrightarrow{a,g,r}_i l \quad a \in CAct \quad u_i \in Comm_i}{u \xrightarrow{a,g,r} u[l \rightarrow i]}$$

$$(P^C3)\ \frac{u_i \xrightarrow{a?,g_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,r_j}_j l' \quad (\nexists k.\, u_k \in Comm_k)}{u \xrightarrow{a,g_i \wedge g_j,r_i \cup r_j} u[l \rightarrow i, l' \rightarrow j]}$$

$$(P^C4)\ \frac{u_i \xrightarrow{a,g,r}_i l \quad a \in CAct \quad (\nexists k.\, u_k \in Comm_k)}{u \xrightarrow{a,g,r} u[l \rightarrow i]}$$

- I is the function which associates invariants to location vectors,

$$I(\langle u_1, ..., u_n \rangle) = \bigwedge_{i=1}^{n} I'_i(u_i)$$

Rule (P^C1) adds a completed action in Π for every possible synchronisation between components, whenever at least one of the parties is in a committed location. Rule (P^C2) denotes the interleaving of completed actions from components in committed locations. Rule (P^C3) adds a completed action in Π for every possible synchronisation between components that are not in committed locations, but only if there are no components in committed locations. Similarly, rule (P^C4) denotes the interleaving of completed actions from components that are not in committed locations, but only if there are no components in committed locations. The semantics of urgent locations is obtained in the augmentation step; urgent locations are irrelevant in the construction of Π because they do not affect interleaving.

By way of example, Figure 2.8 illustrates the syntactic transformation of specifications with urgent and committed locations, into our basic timed automata model ($||^*$ denotes the product automaton construction that preserves the semantics of urgent and committed locations). Figure 2.8(i) shows two extended timed automata; the uppermost contains an urgent location. In the same figure, you can see how this automaton is replaced by its semantically equivalent counterpart, where the urgent location has been replaced by one with the invariant $y \leq 0$ (y is a new clock) and y has been added to the reset set of its ingoing transition ($a!$). In the resulting basic automaton, transition b will be performed without delay since $v(y) = 0$ when location 2 is entered (and correspondingly, when locations $\langle 2, 5 \rangle$ and $\langle 2, 6 \rangle$ are entered in the automaton),

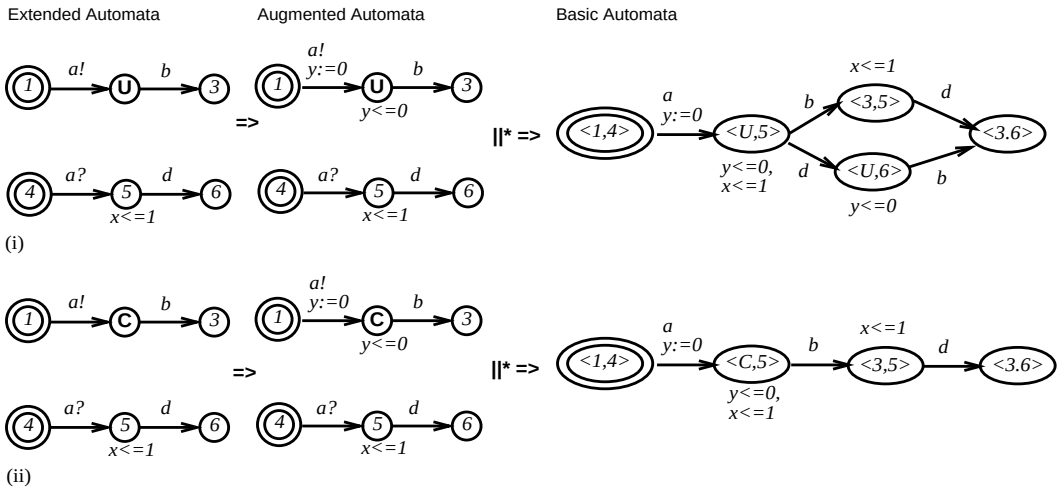


Figure 2.8: The Semantics of Urgent and Committed Locations in Uppaal

but interleaving with transition d is still possible. Indeed, this interleaving causes the branching with sequences b,d and d,b. Compare this with the network of Figure 2.8(ii), where the use of a committed location (instead of an urgent location) restricts the interleaving of b and d: transition b *must* be performed before d.

2.3 Related Work

Alur and Madhusudan [8] give a survey of known results (and open problems) for decidability questions in timed automata theory.

Aceto et al. [2] present a temporal logic, $L_{\forall S}$, which characterises the set of properties decidable by reachability analysis. For any property ϕ in $L_{\forall S}$, a *test automaton* T_ϕ can be constructed with a distinguished rejecting location, l. Given a network $|A$, ϕ is satisfiable in $|A$ if and only if l is not reachable in $|A \,||\, T_\phi$. Importantly, test automata can be used to verify certain properties that are not directly expressible in Uppaal's requirements language (such as, bounded response).

Tripakis [156] presents algorithms to verify timed automata with respect to TCTL and Timed Büchi Automata (TBA), which are implemented in Kronos. As a result, Kronos is able to verify properties that are not expressible in Uppaal. However, full TCTL formulae are verified with a fixpoint algorithm [85], which is computationally demanding. For example, this algorithm requires the product automaton to be constructed before-hand, it cannot provide an answer until all the states satisfying the formula have been computed, and it can only provide limited diagnostics (a symbolic representation of the states satisfying the formula). For reachability and TBA properties, Kronos does better in terms of efficiency, because the reachability graph is computed on-the-fly (although, the algorithms and representation of symbolic states used in

Kronos lack the optimisations included in Uppaal). Another disadvantage of Kronos is that it provides no graphical interface or simulation facilities.

Bouajjani et al. [36] present an on-the-fly model-checking algorithm for the logic TECTL$^*_\exists$, which is shown to be more expressive than TCTL.[11] The paper shows a number of experiments where this algorithm outperforms the fixpoint-based algorithm implemented in Kronos.

Recently, BDD-like data structures have been used in the verification of timed automata based on fixpoint calculations (in the spirit of [85]). These BDD structures are used to represent the transition relation in the reachability graph, and allow fixpoint calculations to be efficiently implemented. Representatives of this technology are the model-checkers Red [164], Rabbit [23] and NuSMVDP [154]. Experiments described in [164] and [23] show that Red and Rabbit outperformed Uppaal in some specifications, such as Fisher's mutual exclusion protocol and the FDDI Token Ring protocol.[12]

Beyer and Noack [24] compare DBMs (as used by Uppaal and Kronos), CDDs (including Clock Restriction Diagrams, a variation of CDDs used in Red) and BDDs (as used in Rabbit). Experiments show that both DBMs and CRDs are, in contrast to BDDs, almost insensitive to the values of constants in the specification, and that efficient use of BDDs and CRDs depends upon finding good variable orderings (in this case, heuristics used in Rabbit make this tool more scalable than Red).

Wang [164] points out that CRDs outperform DBMs when the number of clocks in the specification increases. On the other hand, fixpoint calculations on CRDs consume much time and space in handling intermediate data structures, which need intensive use of garbage-collection.

Thomas, Pandya and Chakraborty [154] show that transition relations can be partitioned into classes, where transitions correspond to the passage of time, synchronisations, or interleaving of actions. Distributing the fixpoint operations over these classes is more efficient than applying them directly over the transition relation (as a whole). Partition heuristics (including strategies to distribute the fixpoint operations) are the main feature of NuSMVDP, and are not included in Rabbit or Red.

Unlike Uppaal and Red, where networks represent only one level of components, Rabbit deals with modular specifications, representing hierarchies of components (this facilitates the specification of complex systems). Unlike Uppaal and Rabbit, Red is able to verify full TCTL formulae with fairness assumptions [165]; in turn, the requirements language of both Rabbit and NuSMVDP is restricted to reachability formulae; thus, it cannot express Uppaal's leads-to formulae. The support for data variables and clock constraints is more limited in Red, Rabbit

[11] A summary of these results are presented in Tripakis PhD thesis [156]; Tripakis refers to this logic as ETCTL$^*_\exists$.

[12] However, these experiments were performed using an old version of Uppaal (v 3.2.4), which lacked many of the optimisations that are currently implemented in the tool.

and NuSMVDP, than it is in Uppaal (e.g., neither Red nor Rabbit support arrays, or clock difference constraints in guards).

2.4 Limitations of Timed Automata and Model-checkers

Timed automata may express behaviours that do not correspond to real executions; in this sense, the notation can be considered "too expressive". In addition, for certain systems, the data objects may be difficult to represent in the specification language of model-checkers, which restrict data support to bounded domains. In this sense, model-checkers may not be expressive enough to reason about the original system. Similarly, certain requirements may not be easily formalised in the model-checker's logic.

In what follows, we explain how these issues affect the reliability of timed automata and model-checkers (see also Chapter 1). This motivates our study of timelock detection methods in the remainder of Part II, and the study of WS1S and MONA in real-time settings, in Part III.

2.4.1 Timelocks in Timed Automata

Timelocks are states where time cannot diverge, and Zeno runs denote arbitrarily fast executions. Both, timelocks and Zeno runs, are abnormal behaviours in the semantics of a timed automaton: they cannot occur in real systems. Moreover, the verification of safety and liveness properties may be meaningless if such abnormal behaviours occur in the model. States of interest may become unreachable when time stops, and most model-checkers (including Uppaal) cannot distinguish between divergent and non-divergent executions.

In what remains of Part II (Chapters 3, 4 and 5), we study timelocks and Zeno runs, and address methods for timelock detection. In Part III (Chapter 8), we use WS1S and MONA to develop a deductive framework (Discrete Timed Automata) where an important class of timelocks can be prevented by construction.

2.4.2 Requirements in Branching-time Logics

Requirements languages for real-time model-checkers are based on branching-time logics, e.g., CTL (Uppaal) and TCTL (Kronos). Many properties can be expressed with branching-time logics, but this is not always easy (see, e.g. [161, 14]).

For instance, requirements that refer to sequences of events (e.g., *if* e_1 *is followed by* e_2 *in less than* n *time units, then* e_3 *should occur no later than* m *time units*), iteration (e.g., e_1 *never occurs more than* n *times in less than* m *time units*) or past computations (e.g., *if* e_1 *occurs, then* e_2 *must have occurred no earlier than* n *time units before*) do not have an intuitive interpretation in Uppaal's requirements language.

Therefore, certain requirements will be difficult (or even impossible) to express in the model-checker's logic, and this undermines our confidence in verification: The model-checker may be able to confirm that a formula holds, but such a formula may not represent the intended requirement. In Chapter 7, we investigate PITL, a linear-time logic where sequences and iteration of events, and past computations, can be naturally expressed. PITL is a powerful and intuitive notation; however, it has a non-elementary worst-case complexity. This has hampered the development of efficient tool support.

Thanks to MONA, we show that the situation looks more promising now. In Chapter 7, we develop a translation from PITL to WS1S, which allows us to use MONA as an efficient decision procedure for PITL. We believe this is an important step towards the development of PITL-based verification tools (e.g., PITL-based real-time model-checkers).

2.4.3 Data in Timed Automata

Among real-time model-checkers, Uppaal offers (to our knowledge) the most comprehensive support for data variables. For instance, Uppaal handles integers, and arrays of clocks and integers. However, data domains must be bounded to permit fully automatic verification.

For certain real-time systems, we may find bounded domains too restrictive. For instance, when modelling communication protocols, we often need to represent unbounded buffers to reason about messages in transit (or similarly, bounded buffers of unknown size). Therefore, in order to use a model-checker, we usually have to fix the size of the buffer to some known value (and this value must be small to avoid state explosion). As a result, the analysis sacrifices generality in favour of automatic verification, which affects our confidence in the system at hand (e.g., would the requirements hold for a different buffer size?).

In Chapter 8, we investigate Discrete Timed Automata, an infinite-state deductive framework, where variables are not restricted to bounded domains. We use WS1S as the assertion language, and MONA provides tool support to facilitate invariance proofs.

Chapter 3

Timelocks and Zeno Runs

Abstract. Timelocks and Zeno runs are abnormal behaviours in the semantics of a timed automaton, which compromise the verification of safety and liveness properties. This chapter, together with Chapters 4 and 5, deal with the issue of timelocks in timed automata and model-checkers. Here, we explain how timelocks and Zeno runs may arise in timed automata, and how they challenge formal verification in different ways. We comment on known methods to guarantee timelock-freedom and absence of Zeno runs, and analyse the support for timelock detection that is provided by model-checkers (in particular, Uppaal and Kronos).

We reveal limitations in model-checkers which motivate our development of new methods in Chapters 4 and 5. In addition, we explain that timelocks and Zeno runs cannot be prevented by construction in Timed Automata, but show that this is possible in Timed Automata with Deadlines [33, 39]. This notation offers a different interpretation of urgency and synchronisation, which we later exploit in Chapter 8 to define Discrete Timed Automata, a deductive framework that also prevents (a certain class of) timelocks by construction.

Organization. Section 3.1 introduces the problem of timelocks and Zeno runs in Timed Automata. We classify timelocks in Section 3.2. In Section 3.3 we discuss how timelocks and Zeno runs affect verification. In Section 3.4 we discuss Timed Automata with Deadlines. In Section 3.5 we review known methods to guarantee timelock-freedom and absence of Zeno runs. We discuss related work in Section 3.6, and Section 3.7 gives a summary of this chapter.

3.1 Introduction

A timelock is a state where time cannot diverge. Timelocks may occur due to errors in the specification of urgent behaviour. As we explained in Chapter 2, urgent actions are obtained by disallowing the progress of time in certain states. A timelock occurs when time stops but

no action is enabled, for example, when a component A may wish to synchronise urgently with another component B, but B is not yet ready to do so. This is what we call a *time-actionlock*. A timelock can also occur when a state is reached where there are enabled actions, which do not lead to divergent runs (e.g., runs may be trapped in a loop of urgent actions). This is what we call a *Zeno-timelock*. This classification of timelocks is explained in more detail in Section 3.2.

Specifications cannot be reliably verified when timelocks occur. For instance, a property stating that a certain unwanted state is never reached (namely, a safety property) may hold in a timelocked specification just because a timelock prevents time from passing up to a point where such a state becomes reachable. However, because timelocks cannot be implemented, such an unwanted state may still occur in some execution of the implemented system.

A related problem concerns the detection of Zeno runs, i.e., convergent runs where actions occur infinitely often (see Section 2.1.2). Zeno runs, like timelocks, are counterintuitive: a process cannot be infinitely fast. However, while timelocks always denote errors, Zeno runs may occur in abstract specifications, and are not necessarily undesirable. Furthermore, Zeno runs do not imply the occurrence of timelocks, and safety properties are not compromised in (timelock-free) specifications with Zeno runs. However, liveness properties may be affected by Zeno runs, and, in general, verification in real-time systems is well-defined only over divergent runs (see Section 3.3). Therefore, we need methods to guarantee timelock-freedom and absence of Zeno runs, and also methods that guarantee timelock-freedom with no concern for Zeno runs.

Timelocks and Zeno runs have been investigated by a number of researchers, including Henzinger et al. [85], Bornot et al. [33], Tripakis [156, 157] and Bowman [38, 39]. Timed Automata with Deadlines [33, 39] is a notation where time-actionlocks are ruled out by construction (Section 3.4.1). However, Zeno-timelocks and Zeno-runs can still occur, and most model-checkers do not support this notation (the interpretation of urgency and synchronisation is fundamentally different).[1] On the other hand, timelocks and Zeno runs cannot be prevented by construction in timed automata, thus we must focus on detection methods instead.

Model-checkers require the specification to be timelock-free (and in some cases, in addition, free from Zeno runs). However, few model-checkers support detection of timelocks and Zeno runs, and the methods implemented suffer from a number of shortcomings (and are confined to particular tools).

A liveness formula can be verified in Uppaal that denotes absence of timelocks and Zeno runs, but the algorithm involves a demanding nested reachability analysis, and rules out timelock-free specifications where Zeno runs occur. In the case of Kronos and Red, where the requirements language is more expressive than Uppaal's (TCTL), absence of timelocks can be characterised in a more precise way, but verification requires fixpoint algorithms which can be computationally

[1] However, the tool IF [50] handles a similar notation (Section 3.6).

expensive. Another limitation of Kronos is that absence of Zeno runs cannot be checked, and the product automaton (in order to verify networks of automata) must be constructed before timelock-freedom can be checked (more about timelock detection in Uppaal and Kronos can be found in Section 3.5.1). In addition, let us mention that both Red [165] and Profounder [159] are able to detect Zeno runs during verification, but these require demanding algorithms.

Tripakis introduced in [156] the *strong non-Zenoness* property, a static check based on the structure of loops in the automaton, which guarantees absence of Zeno runs. This property is compositional, holds in most specifications, and implies absence of Zeno-timelocks. Hence, in specifications where strong non-Zenoness hold, timelock-freedom corresponds precisely to deadlock-freedom. Therefore, and because deadlock-freedom is usually asserted before any other verification takes place, in most cases, strong non-Zenoness suffices to guarantee timelock-freedom (avoiding the demanding checks of liveness formulae in Uppaal or Kronos).

However, one problem with the definition of strong non-Zenoness is that non-simple loops must be detected, which is computationally expensive (Section 2.1.1). Another problem is that many useful specifications, which are free from timelocks and Zeno runs, cannot be recognised as safe by the compositional application of strong non-Zenoness. In addition, strong non-Zenoness is of no use if we want to guarantee timelock-freedom independently of absence of Zeno runs, and has not been implemented in any model-checker (we study strong non-Zenoness in Section 3.5.2).

3.2 A Classification of Timelocks

Generally speaking, progress in timed automata executions can be prevented by *deadlocks* or *timelocks*. A deadlock is a state where, for however long time is allowed to pass, no further actions can be performed. Formally, given a timed automaton $A = (L, TL, T, l_0, C, I)$ with timed transition system $TS_A = (S, Lab, T_S, s_0)$, a state $s \in S$ is a deadlock if

$$\forall\, d \in \mathbb{R}^{+0}.\, (s+d) \in S \;\Rightarrow\; \nexists\, a \in TL\,.\, (s+d) \stackrel{a}{\Longrightarrow}$$

where, if $s = [l, v]$, then $s + d = [l, v + d]$. On the other hand, a *timelock* is a state $s \in S$ where time is not able to pass beyond a certain bound.

$$\forall\, \rho \in \mathit{Runs}(s).\, \mathit{delay}(\rho) \neq \infty$$

A timed automaton is deadlock-free (timelock-free) if none of its reachable states is a deadlock (timelock). Deadlocks and timelocks can be further classified as *pure-actionlocks*, *time-actionlocks* or *Zeno-timelocks*.

Pure-actionlock. A pure-actionlock is a state where the system cannot perform any actions, but time is allowed to diverge. Formally, a state s is a pure-actionlock if

$$\forall\, d \in \mathbb{R}^{+0}\,.\,(s+d) \in S \;\wedge\; \nexists\, a \in TL\,.\,(s+d) \stackrel{a}{\Longrightarrow}$$

Time-actionlock. A time-actionlock is a state where neither actions can be performed nor time can pass. Formally (recall that $Lab = TL \cup \mathbb{R}^{+}$), $s \in S$ is a time-actionlock if

$$\nexists\, \gamma \in Lab\,.\,s \stackrel{\gamma}{\Longrightarrow}$$

Zeno-timelock. A Zeno-timelock is a state where the system can still perform actions, but time cannot diverge. This represents a situation where the system performs an infinite number of actions in a finite period of time. Formally, $s \in S$ is a Zeno-timelock if (a) there are no divergent runs starting from s, and (b) all finite runs starting from s can be extended to Zeno runs (i.e., convergent runs where actions occur infinitely often). $ZRuns(s)$ denotes the set of Zeno runs starting from s (see Section 2.1.2).

$$\forall\, \rho \in Runs(s).\, delay(\rho) \neq \infty \;\wedge\; \forall\, \rho' \in FiniteRuns(s).\, \exists\, \rho'' \in ZRuns(s).\, \rho' \subseteq \rho''$$

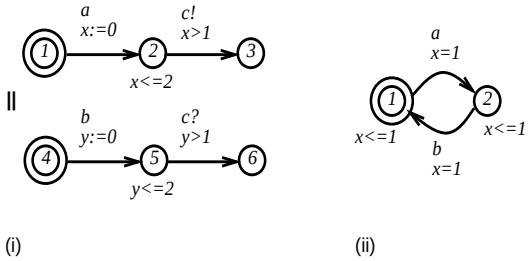


Figure 3.1: Timelocks. (i) Time-actionlock. (ii) Zeno-timelock

Figure 3.1 illustrates the occurrence of time-actionlocks and Zeno-timelocks. Figure 3.1(i) shows a network where a time-actionlock may occur. Suppose that b is initially performed, and a is performed 2 time units later. At this point $v(x) = 0$ and $v(y) = 2$, and so the automata must synchronise on c, but this is not possible because $c!$ is not enabled. Therefore, a time-actionlock arises (the invariant in location 5 prevents time from passing). The error was to force synchronisation at a time when components may not be ready to do so(perhaps, the specifier forgot to synchronise a and b first). Figure 3.1(ii) shows a timed automaton where a Zeno-timelock occurs. The invariants prevent time from passing any further when $v(x) = 1$, but all actions in the loop are enabled. Therefore, the only possible evolutions are characterised by Zeno runs in a finite period of 1 time unit (perhaps, the specifier forgot to reset x).

3.2.1 Example: A Multimedia Stream Model with Time-actionlocks

Consider the specification shown in Figure 3.2, where packets can be sent to the *Sink* by two different sources, *Source1* and *Source2*. The behaviour of *Source2* is similar to that of *Source1*, although *Source2* can send the first packet at any time, and it sends the following packets faster (1 packet every 25 ms). This configuration, however, causes time-actionlocks in the network.

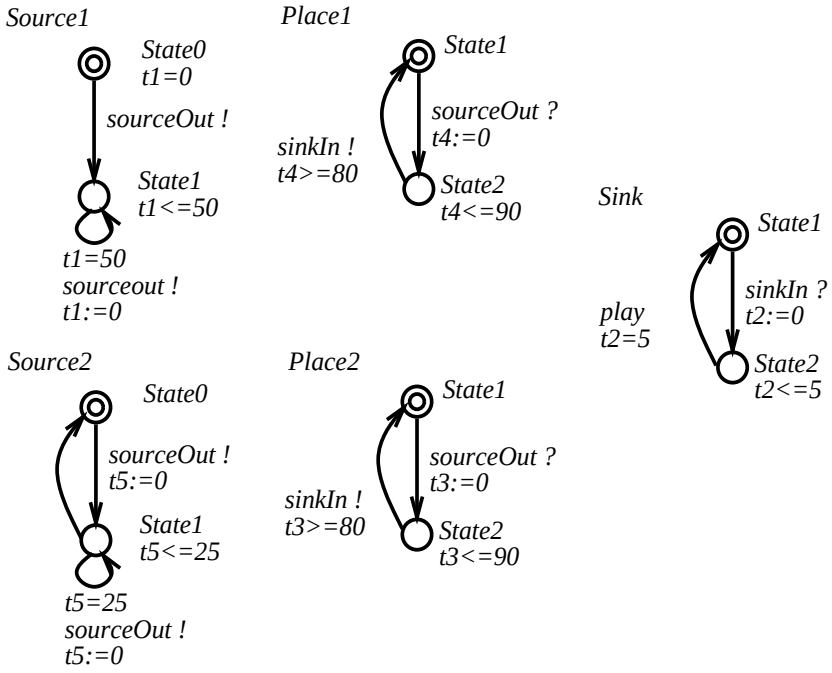


Figure 3.2: A Multimedia Stream with Two Sources and a Time-actionlock

For instance, the following scenario will conclude in a time-actionlock. *Source1* sends its first packet to *Place1*, at time $t = 0$. Later, say at $t = 10$, *Source2* sends its first packet to *Place2*. By the time that *Source2* attempts to send the second packet ($t = 35$), neither *Place1* nor *Place2* can offer a matching *sourceOut?*, because the previous packets have not yet arrived to the *Sink* (the transmission delay is at least 80 ms). At this point, a time-actionlock occurs: *sourceOut!* in *Source2* is urgent at $t = 35$, but no action is enabled. The time-actionlock occurs because *Source2* makes the *sourceOut!* action urgent even when it is not enabled, as synchronisation with *Place1* or *Place2* is not possible.

We argue that it should only be possible to make an action urgent if it is enabled, i.e., as Bowman pointed out in [39], *"must requires may or, in other terms, you can only force what is possible."*. Such an interpretation of urgency arises in timed automata with deadlines (which we will discuss in Section 3.4.1). Unlike timed automata, timed automata with deadlines are inherently free from time-actionlocks (by construction).

3.2.2 Example: A Multimedia Stream Model with Zeno-timelocks

Figure 3.3 shows a multimedia stream similar to the one described in Section 3.4 (Figure 3.2), but where a new component *Source3* has been added. This automaton models an unreliable source, which will attempt to send packets at a speed of 1 packet per 100 ms but, occasionally, a *failure* may occur that forces the source to enter an *Offline* state. *Source3* will remain in *Offline* for an unspecified period of time, and then it will restart the sequence again.

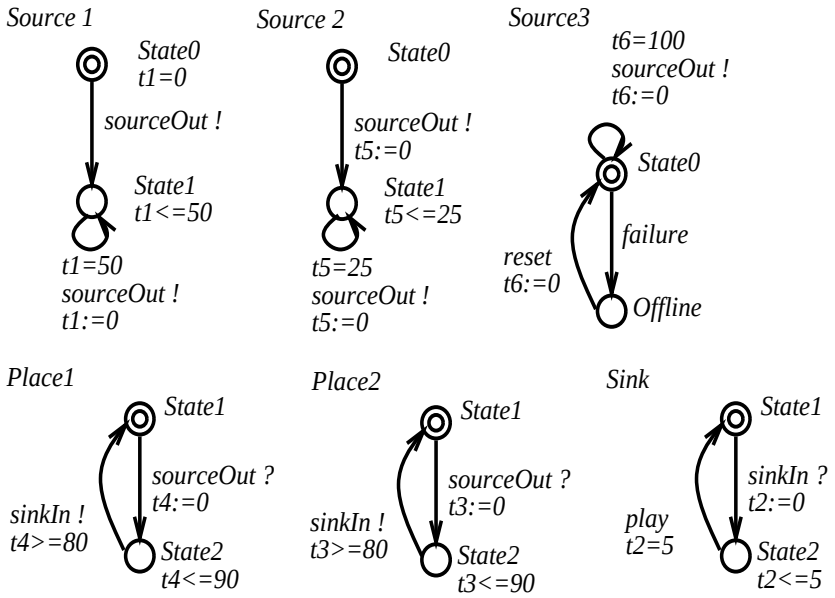


Figure 3.3: A Multimedia Stream with Three Sources and a Zeno-timelock

Consider, once more, the scenario that in Figure 3.2 caused a time-actionlock: *Source1* sends at $t = 0$ to *Place1*; *Source2* sends at $t = 10$ to *Place2* and attempts to send again at $t = 35$, but is blocked because synchronisation with *sourceOut?* in either *Place1* or *Place2* is not possible. At this point, $v(t5) = 25$, and so the invariant $t5 \leq 25$ in *Source2.State1* prevents time from passing any further. However, unlike in the specification of Figure 3.2, a time-actionlock does not occur because transition *failure* at *Source3.State0* is enabled. Moreover, all infinite runs starting from *Source3.State0* converge, because the loop $\langle failure, reset \rangle$ can be visited infinitely often while time is blocked by *Source2*. A Zeno-timelock occurs, then, at a state $s = [l, v]$ where l is a location vector denoting *Source1.State1*, *Source2.State1*, *Source3. State0*, *Place1.State2*, *Place2.State2* and *Sink.State1*, and v is s.t. $v(t) = 25$ for $t \in \{t3, t5\}$ and $v(t) = 35$ for $t \in \{t1, t2, t4, t6\}$.

3.3 The Nature of Timelocks and Zeno Runs

Timelocks are quite different from deadlocks. Firstly, unlike deadlocks, timelocks are counter-intuitive: systems cannot "stop time", but it is perfectly possible that a system reaches a state

where it cannot perform further actions. Secondly, a timelock always prevents the whole system from evolving any further (because automata synchronise on the passage of time), whereas a (local) deadlock does not necessarily block other components in the system.

Time-actionlocks are different from Zeno-timelocks. A time-actionlock is usually caused by mismatched synchronisation, while Zeno runs and Zeno-timelocks are typically caused by under-specification of time constraints (see Sections 3.2.1 and 3.2.2).

A change in the semantics of urgency and synchronisation in timed automata would allow us to prevent time-actionlocks by construction; in fact, Section 3.4.1 discusses Timed Automata with Deadlines, where this is indeed the case. However, it seems impossible to prevent Zeno runs and Zeno timelocks by construction, without disallowing abstract specifications.

Absence of Zeno runs implies absence of Zeno-timelocks (by definition of Zeno-timelock), but not necessarily absence of time-actionlocks. In fact, in specifications where Zeno runs do not occur, timelock-freedom is equivalent to deadlock-freedom. On the other hand, timelock-freedom does not guarantee the absence of Zeno runs.

Timelocks have undesirable consequences on verification. Unless a specification is guaranteed to be timelock-free, the results obtained during verification may be meaningless. Figure 3.4 shows a network where a timelock occurs in *A1* when $v(x) = 1$. Consider the verification of the safety property $\texttt{AG}(\neg\, A2.Error)$, which states that location *Error* in *A2* is never reachable. This location can only be reached after transition b is performed, but this will never happen: b is enabled only when $v(x) > 2$, but the timelock in *A1* does not allow time to progress beyond $v(x) = 1$. Reachability analysis would determine that the safety property is satisfiable; however, the error-state (which is characterised by *A2* reaching *Error*) may be reachable at some later implementation stage.

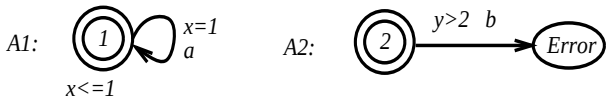


Figure 3.4: Timelocks and Safety Properties (Location *Error* is Unreachable)

The effect of timelocks and Zeno runs on verification, however, is different. In a timelock-free specification, there is always at least one divergent run, even from those states where Zeno runs are possible. Firstly, Zeno runs do not prevent states from being reached by real executions, because they do not stop time. Secondly, Zeno runs cannot "produce" states that are not already reachable by real executions. Nonetheless, in general, verification of real-time systems should only consider divergent runs. The following example shows that Zeno runs may invalidate the verification of liveness properties, even when the specification is timelock-free.

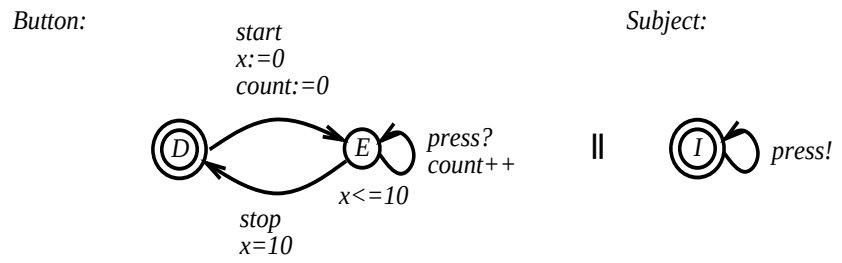


Figure 3.5: Zeno Runs and Response Properties (A Model for the Button-test)

Consider here an experiment to measure how fast a button can be pressed in a 10-second period (Figure 3.5). A subject is asked to press the button as many times as she can: a variable *count* is incremented every time the button is pressed. When the period expires, the button is disabled. Note that the specifier did not assume anything about the speed of the subject; therefore no time constraints were placed on the *press!* action. Obviously, the subject cannot press the button infinitely fast, but the specification does not prevent such interaction. This is reflected by Zeno runs where the *press* action (resulting from synchronising *press?* and *press!*) occurs arbitrarily fast. Despite the occurrence of Zeno runs, the network is timelock-free: time can always pass in location *D*, which in turn is always reachable from *E*.

Consider the requirement "from any state where the button is enabled, we can always eventually reach a state where the button is disabled". In Uppaal, we could verify this requirement using the leads-to formula,

```
Button.E --> Button.D
```

This formula is satisfiable if, from every state where the button is enabled, *every* run thereafter reaches a state where the button is disabled (Section 2.2). However, this formula does not hold in the specification of Figure 3.5, because Zeno runs occur when executions remain in *E*.

We conclude this section with the following observation. In the verification of real-time systems, it is often the case that we are not interested in general response properties, but rather in bounded response. Bounded-response properties are of the form "whenever P holds, Q must eventually hold before n time units". It is well-known that bounded-response can be expressed as safety (see, e.g., [85]); thus Zeno runs will not affect the verification of bounded-response properties if they are mapped to safety properties.[2]

[2]Behrmann et al. [22, p. 228]give a method to translate bounded-response to safety formulae in Uppaal.

3.4 Prevention of Time-actionlocks by Construction

Time-actionlocks occur in timed automata because urgency is given an excessively strong interpretation: actions may be urgent even though they are not enabled. We illustrated this issue in the multimedia stream example of Figure 3.2. This section presents Timed Automata with Deadlines [32, 33, 38, 39], where time-actionlocks are prevented by construction.

3.4.1 Timed Automata with Deadlines

Informally speaking, timed automata with deadlines (or TADs, for short) are very much like timed automata, but they do not include invariants as time-progress conditions. Instead, urgency is expressed by *deadlines*, which are clock constraints associated with transitions. Time can pass in any location as long as the current valuation does not satisfy any outgoing deadline (i.e., deadlines attached to outgoing transitions). In this sense, deadlines and invariants can be regarded as dual; in timed automata, time can pass as long as the invariant holds, whereas in TADs time can pass only until a deadline holds. The main differences between timed automata and TADs are that deadlines hold only if the corresponding transitions are enabled, and synchronisation is urgent only if both parties are enabled. This ensures that TADs are, by construction, free from time-actionlocks (this is also known as *time reactivity* [32, 33]: from every reachable state, either time may pass or some action may be performed).

Different variants of TADs have been proposed, which differ in the treatment of synchronisation (although all of them preserve time reactivity), e.g., standard TADs, sparse TADs and TADs with minimal priority escape transitions [39]. The presentation of TADs in this section follows the model of sparse TADs.

Example. Figure 3.6(i) shows a network of two TADs[3], and the corresponding product automaton to the right. Assuming all clocks are initially set to zero, transition *b!* is enabled in the time interval $[1, \infty)$, and is urgent in $[2, \infty)$: the deadline $(x \geq 2)$ expresses that, during $[2, \infty)$, synchronisation must happen *as soon as possible*. In the second automaton, *b?* is enabled during $[3, \infty)$, and is urgent in $[4, \infty)$. What happens when $v(x) = v(y) = 2$? At this point in execution, transition *b!* is enabled and enters its urgency interval, but *b?* is not yet enabled, and so synchronisation cannot occur.

Deadlines on half actions enforce urgency only if synchronisation can be achieved. Equivalently, synchronisation is urgent as soon as (a) one of the parties is urgent, and (b) both parties can synchronise (i.e., when both half actions are enabled). As a result, *b!* "waits" for *b?* to become enabled, and so synchronisation occurs when $v(x) = v(y) = 3$. Note that time passes until

[3] In our TAD figures, deadlines are shown in brackets; "," denotes conjunction; and "|" denotes disjunction.

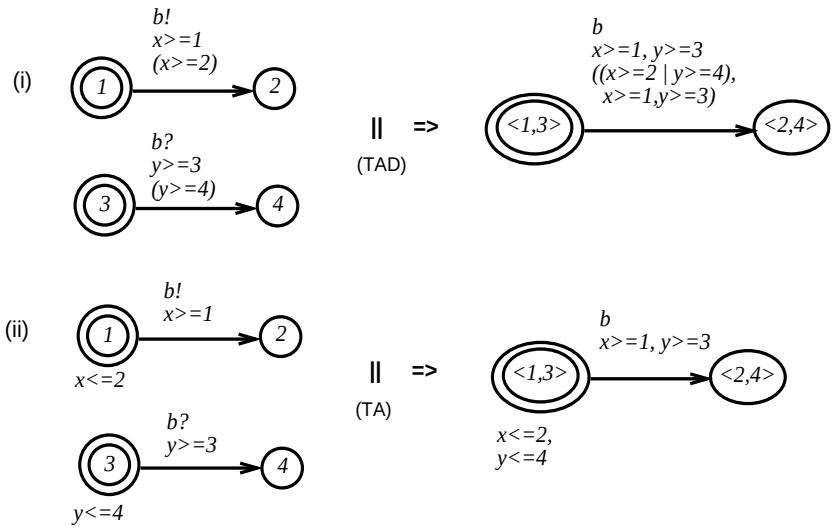


Figure 3.6: Parallel Composition in TADs and TAs

synchronisation is possible, hence synchronisation mismatches cannot cause time-actionlocks. This behaviour is evident from the structure of the product automaton; the guard of b results from conjoining the component guards, and the deadline assigned to b is defined as the disjunction of the component deadlines, conjoined with the component guards. In this way, the deadline of b implies its guard.

The network of timed automata in (ii), on the other hand, shows that invariants enforce a stronger form of urgency than deadlines. The timed automata in (ii) "intend" to mimic the TADs in (i); for example, the invariant $x \leq 2$ (in location 1), makes $b!$ urgent at $v(x) = v(y) = 2$. However, the timed automata contain a time-actionlock at $v(x) = v(y) = 2$; synchronisation cannot yet occur, but the invariant in location 1 ($x \leq 2$) prevents time from passing any further. Even though, in this example, half actions are not urgent until they are enabled (e.g., $b!$ becomes enabled at $v(x) = 1$, and is not urgent until $v(x) = 2$), invariants will enforce this "local" urgency regardless of whether synchronisation is possible. This can be seen in the product automaton; the invariant in $\langle 1, 3\rangle$ prevents time from passing beyond $v(x) = v(y) = 2$, even though b is not yet enabled.

Next, we comment on the basic elements of the theory, and refer the reader to [32, 33, 38, 39] for a more comprehensive presentation. Unless stated otherwise, the notation respects the sets and conventions defined for timed automata in Section 2.1.1.

Syntax. A timed automaton with deadlines is a tuple of the form $A = (L, l_0, TL, T, C)$, where L is a finite set of locations; $l_0 \in L$ is the initial location; $TL \subseteq Act$ is a set of labels; T is

a transition relation and C is a set of clocks. Transitions in T are denoted $l \xrightarrow{a,g,d,r} l'$, where $l, l' \in L$ are automata locations; $a \in TL$ is the action labelling the transition; $g \in CC_C$ is a guard; $d \in CC_C$ is a deadline; and $r \in \mathcal{P}(C)$ is a reset set. In addition, deadlines and guards satisfy the following properties.

1. Deadlines imply guards,

$$\text{(C1)} \quad l \xrightarrow{a,g,d,r} l' \implies (d \Rightarrow g)$$

2. If both a deadline and its corresponding guard denote the same solution set, then this set must denote a left-closed time interval,

$$\text{(C2)} \quad l \xrightarrow{a,g,d,r} l' \implies ((d = g) \Rightarrow \exists\, v \,.\, (v \models g) \wedge \forall\, v' \,.\, (v' \models g) \Rightarrow v' \geq v)$$

Let us illustrate the necessity for condition (C2) with the following example. Transition a in the TAD of Figure 3.7 has a guard $g = x > 1$ and a deadline $d = x > 1$. Notice that both g and d denote the same solution set, which corresponds to the left-open interval $v(x) \in (1, \infty)$. The deadline does not allow time to pass beyond $v(x) = 1$ (to see why, check the semantic rule S2 below); therefore time stops but the action is not yet enabled, and a time-actionlock occurs in location 1. This example shows that TADs that do not fulfill (C2) are not guaranteed to be time reactive, even if deadlines imply guards (C1).

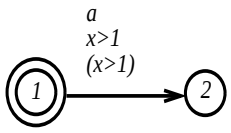


Figure 3.7: Left-open Intervals in TADs

Semantics. Let $A = (L, l_0, TL, T, C, I)$ be a TAD where all actions are completed (i.e., $TL \subseteq CAct$). The semantics of A are given by the TTS (S, s_0, Lab, T_S), where

- $S \subseteq L \times \mathbb{V}_C$ is the set of reachable states; i.e., the smallest set of states that includes s_0 and is closed under the transition relation T_S,

$$S = \{\, s_0 \,\} \;\cup\; \{\, s' \mid \exists\, s \in S, \gamma \in Lab \,.\, s \stackrel{\gamma}{\Longrightarrow} s' \in T_S \,\}$$

- $s_0 = [l_0, v_0]$ is the initial state;

- $Lab = TL \cup \mathbb{R}^+$ is the set of labels for transitions in T_S; and

- $T_S \subseteq S \times Lab \times S$ is the transition relation defined by the following rules,

$$\text{(S1)}\quad \frac{l \xrightarrow{a,g,d,r} l' \quad v \models g}{[l,v] \overset{a}{\Longrightarrow} [l', r(v)]}$$

$$\text{(S2)}\quad \frac{\forall\, l'\,,\, l \xrightarrow{a,g,d,r} l' \;\Rightarrow\; \forall\, \delta' < \delta \in \mathbb{R}^+\,,\, (v+\delta') \not\models d}{[l,v] \overset{\delta}{\Longrightarrow} [l, v+\delta]}$$

Parallel Composition Let $|A = \langle A_1, \ldots, A_n \rangle$ be a network of TADs, where $A_i = (L_i, l_{i,0}, TL_i, T_i, C_i)$. Let u, u', etc. denote location vectors. The product automaton is defined as

$$\Pi = (L, l_0, TL, T, C)$$

where

- L is the smallest set of location vectors which includes l_0 and is closed under the transition relation T,

$$L = \{\, l_0 \,\} \;\cup\; \{\, u' \mid \exists\, u \in L,\, a \in TL,\, g, d \in CC_C,\, r \in \mathcal{P}(C)\,.\, u \xrightarrow{a,g,d,r} u' \in T \,\}$$

- l_0 is the initial location vector, $l_0 = \langle l_{1,0}, \ldots, l_{n,0} \rangle$;

- TL is the set of actions labelling the transitions in T,

$$TL = \{\, a \mid \exists\, u, u' \in L,\, g, d \in CC_C,\, r \in \mathcal{P}(C)\,.\, u \xrightarrow{a,g,d,r} u' \in T \,\}$$

- T is the transition relation defined by the following rules ($1 \leq i \neq j \leq n$),

$$\text{(TAD1)}\quad \frac{u_i \xrightarrow{a?,g_i,d_i,r_i}_i l \quad u_j \xrightarrow{a!,g_j,d_j,r_j}_j l'}{u \xrightarrow{a,g',d',r_i \cup r_j} u[l \to i, l' \to j]}$$

$$\text{(TAD2)}\quad \frac{u_i \xrightarrow{a,g,d,r}_i l \quad a \in CAct}{u \xrightarrow{a,g,d,r} u[l \to i]}$$

  where $g' \triangleq g_i \wedge g_j$ and $d' \triangleq g_i \wedge g_j \wedge (d_i \vee d_j)$; and

- C is the set of clocks, $C = \bigcup_{i=1}^{n} C_i$

Rule (TAD1) defines synchronisation in TADs. As in timed automata, guards and reset sets of component transitions (matching half actions) are combined in the resulting transition in the product automaton (completed actions). The disjunction of component deadlines ensures that

synchronisation is made urgent if at least one of the half actions involved is urgent. Conjoining the component guards ensures that deadlines in the product automaton's transitions imply their guards; in this way time reactivity is preserved. In other words, synchronisation is urgent only if it can be performed. Finally, rule (TAD2) gives the standard interpretation for completed actions in component TADs.

3.4.2 Example: A TAD Model For The Multimedia Stream

Figure 3.8 shows a TAD specification for the multimedia stream, which corresponds to the network of timed automata shown in Figure 3.2. Transitions have been annotated with the necessary deadlines (shown in brackets). For example, *sourceOut?* in *Source1.State1* is made urgent as soon as it is enabled (with a deadline $t1 = 50$). Let us revisit the scenario which caused a time-actionlock in the timed automata specification of Figure 3.2. *Source1* sends at $t = 0$; *Source2* sends at $t = 10$ and attempts to send the second packet at $t = 35$. At this point, *Source2* blocks because synchronisation with *sourceOut?* in either *Place1* or *Place2* is not possible. However, unlike in the timed automata specification, time is not prevented from passing and all other components can evolve normally; deadlines attached to half actions are only enforced if synchronisation can be achieved (rule TAD1).

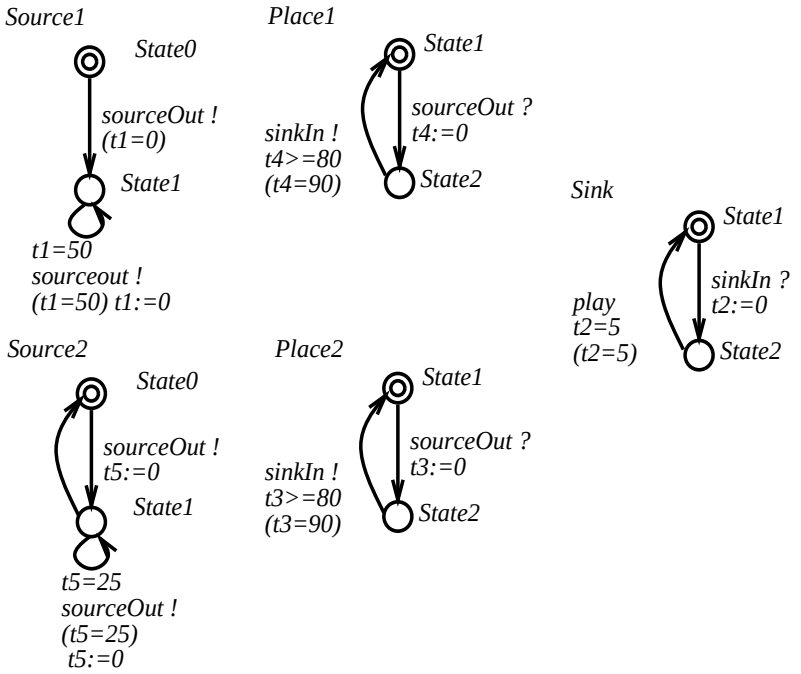


Figure 3.8: A TAD Specification for the Multimedia Stream (with Two Sources)

3.5 Detection of Timelocks and Zeno Runs

This section studies the support for timelock detection available in Uppaal and Kronos, which is characterised by the verification of liveness properties, and also analyses the strong non-Zenoness property [156], a static check that guarantees absence of Zeno runs.

3.5.1 Liveness Properties in Kronos and Uppaal

Detection of Timelocks in Kronos. In Kronos, timelock-freedom can be asserted by model-checking the formula,

$$\lambda_K \triangleq \texttt{init impl ab ed\{=1\} true}$$

which corresponds to the TCTL formula $\forall\Box\exists\Diamond_{=1} true$. This formula denotes that 1 time unit can pass from every reachable state; in other words, divergent runs are possible from every state.

The nature of λ_K. The formula λ_K is sufficient-and-necessary for timelock-freedom [85], thus it cannot be used to conclude absence (or occurrence) of Zeno runs. λ_K is not compositional; moreover, Kronos requires the product automaton to be constructed a priori.

Detection of Timelocks and Zeno Runs in Uppaal. Uppaal's requirements language is not expressive enough to characterise a formula equivalent to λ_K. However, Uppaal can verify a CTL formula that guarantees *both* timelock-freedom and absence of Zeno runs. This is done via a test automaton [2] and a leads-to formula [160]. Given a network of timed automata, a test automaton is added as a new component as shown in Figure 3.9 (the test automaton consists of a single location, T). The network is free from timelocks and Zeno runs if the leads-to formula λ_U holds, where

$$\lambda_U \triangleq \texttt{t==0 --> t==1}$$

λ_U if from every reachable state, every run allows a 1 time unit delay.

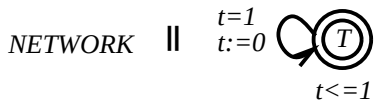


Figure 3.9: The Test Automaton Approach

The nature of λ_U. λ_U is a stronger property than λ_K, and is sufficient-only both for timelock-freedom and absence of Zeno runs. Therefore, in specifications where λ_U does not hold, we cannot determine whether timelocks or Zeno runs occur (or both). λ_U is not compositional; however, model-checking λ_U is done on-the-fly (Uppaal does not build the product automaton).

3.5.2 The Strong Non-Zenoness Property

The strong non-Zenoness property [156] is a condition on the guards and resets of a loop, which guarantees that in every iteration of the loop time passes at least by d time units ($d \in \mathbb{N},\ d \geq 1$). Therefore, any run that covers (i.e., visits infinitely often) a strongly non-Zeno loop is necessarily divergent. If every loop in the automaton is strongly non-Zeno, actions occur infinitely often only in divergent runs. By definition, then, Zeno runs (and therefore, Zeno-timelocks) cannot occur. We recall the definition of strong non-Zenoness below. The definition applies both to simple and non-simple loops (both kinds are loops are considered "structural loops" in [156]).

Strong Non-Zenoness (SNZ). Let A be a timed automaton, and lp a loop in A. The (simple or non-simple) loop lp is called *strongly non-Zeno* (or an SNZ-loop) if there exists a clock x and two transitions t_1 and t_2 in the loop (not necessarily different) such that x is reset in t_1 and bounded from below in t_2. If every loop in A is SNZ, then A is said to be SNZ. Figure 3.10 shows an SNZ-loop.

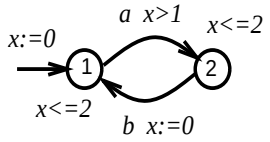


Figure 3.10: A Strongly Non-Zeno Loop

The Nature of Strong Non-Zenoness. Strong non-Zenoness is a sufficient-only condition to guarantee the absence of Zeno runs (Lemma 3.1 and Corollary 3.2). In addition, strong non-Zenoness is compositional: if every component in a network is strongly non-Zeno, then the network is free from Zeno runs (Lemma 3.3).

LEMMA **3.1.** *Any run that covers a SNZ-loop is necessarily divergent.*

Proof. Let lp be a SNZ-loop. Let $t_1, t_2 \in \mathit{Trans}(lp)$, s.t. $v(x) = 0$ immediately after t_1 is performed, and t_2 requires $v(x) \geq c,\ (c \in \mathbb{N}, c > 0)$ to be performed. Then, any run that covers lp accumulates a delay of at least c time units in every iteration; i.e., it is a divergent run. Hence, the lemma holds. □

COROLLARY **3.2.** *Zeno runs cannot cover SNZ-loops.*

LEMMA **3.3.** *Let A be a timed automaton. If A is SNZ, then A is free from Zeno runs (and thus, free from Zeno-timelocks). Moreover, if all components in a network $|A = |\langle A_1, \ldots, A_n\rangle$ are SNZ, then the network $|A$ is free from Zeno runs (and thus, free from Zeno-timelocks).*

Proof. See [156]. □

Zeno runs cannot occur in specifications that are strongly non-Zeno. However, false checks can be obtained for safe specifications. Figure 3.11(i) is an example of the sufficient-only nature of strong non-Zenoness. The loop $\langle a, b, c\rangle$ is not SNZ (x is not reset anywhere in the loop), but it does not contain Zeno runs or timelocks (after transition c is performed, transition b is disabled). On the other hand, the automaton in 3.11(ii) is not SNZ (x is not bounded from below) and contains Zeno runs, but is timelock-free.

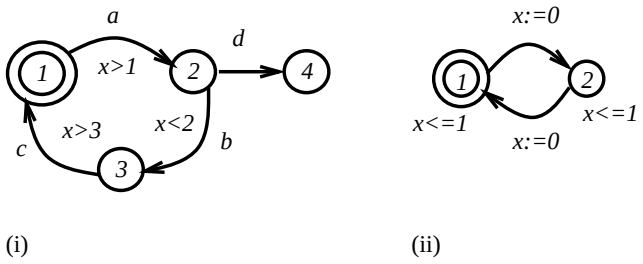


Figure 3.11: Non-SNZ Timelock-free Automata

3.6 Related Work

Here we comment on related work done on timelocks in timed automata and process calculi. We discuss later, in Chapter 8, studies on timelocks in deductive frameworks.

Timelocks in Timed Automata. Henzinger et al. [85] give a fixpoint algorithm to check for timelock-freedom. This algorithm is implemented in Kronos [170] via the λ_K formula (Section 3.5.1); Yovine [169] and Tripakis [156] present a number of case studies where λ_K is applied. Kronos, however, cannot detect Zeno runs.

Bouajjani et al. [36] implements a prototype tool to model-check an extension of TCTL with Timed Buchi Automata, where Zeno runs are detected during verification and avoided (i.e., they obtain a model-checking algorithm that is faithful to the semantics of TCTL and Timed Büchi Automata). This algorithm works on-the-fly over the simulation graph (similar to Uppaal's forward reachability graph). The authors refine these results in [159], and implement the tool Profounder to check emptiness of Timed Büchi Automata (using the simulation graph).

The model-checker Red can assert timelock-freedom via a TCTL formula (equivalent to Kronos' formula λ_K). In addition, Red is able to detect (and avoid) Zeno runs during verification [165]. However, unlike in [36, 159], Red implements fixpoint-based algorithms and verifies TCTL formulas with fairness assumptions. Similar to Kronos and Red, the model-checker TMV [146] asserts timelock-freedom (TCTL formula), but TMV cannot detect Zeno runs.

Tripakis [156, 157] gives an on-the-fly algorithm to detect timelocks that is based on nested reachability analysis. This algorithm is defined over a single automaton that must be syntactically modified before the analysis takes place, and denotes a sufficient-and-necessary check for timelock-freedom (it is similar to checking satisfiability of λ_K).

Braberman and Olivero [51] present I/O Timed Components, which are partitions of a timed automaton's actions as input, output or internal actions, such that composition is guaranteed to preserve timelock-freedom. However, the timed automata (components of the network) are assumed to be timelock-free, and no systematic way of deriving I/O Timed Components is given.

Bornot et al. [32, 33] introduce the concept of deadlines as progress conditions for timed automata. In addition, they discuss a number of composition strategies to preserve time reactivity.

Bowman et al. [43] report on time-actionlocks found in a timed automata specification of a lip-synchronisation protocol. The analysis was performed in Uppaal, and time-actionlocks were detected as deadlocks using the Uppaal formula `A[]not deadlock`. In [38], Bowman explains the difficulty of defining timeouts without introducing timelocks in the specification. Bowman [39] is the first in distinguishing between time-actionlocks and Zeno-timelocks, and gives new composition strategies for timed automata with deadlines (which address limitations in [32, 33]).

Tsoronis [160] introduces the leads-to property λ_U (Section 3.5.1), and uses it to guarantee timelock-freedom in a number of case studies in Uppaal. Time-actionlocks and Zeno-timelocks are formalised later by Su in [150] (our classification of Section 3.2 follows from Bowman and Su's work, although we present a simpler formalisation for Zeno-timelocks). Su also implements a strong non-Zenoness check over Uppaal specifications; however the prototype cannot handle specifications with certain loop configurations.

IF [50] is a validation environment for real-time systems, which relies on an intermediate representation language based on timed automata with deadlines, and supports front-ends for high-level specification languages (such as UML) and validation tools (such as SPIN [88] and CADP [68]). The intermediate language is asynchronous: Processes communicate by message passing, senders never block (messages are kept in a queue), and receivers block until the message is available. Time reactivity is achieved by enforcing urgency only over enabled actions (note, a blocked receiver is not enabled until the input message is available). In addition, IF supports dynamic process creation and destruction, and the use of dynamic priorities to restrict non-determinism in execution. IF verifies safety properties expressed by "observers" (observers are similar to test automata in Uppaal). However, the tool does implement checks for timelock-freedom or absence of Zeno runs.

Timelocks in Process Calculi. The problem of timelocks has been noted (and partially resolved) in the early timed concurrency theory work, which focused on timed process calculi. As a result, most timed process calculi only allow urgency to be applied to internal actions (external -or observable- actions are never urgent), enforcing the *as soon as possible* (asap) principle [138]. In process calculi with asap, internal actions are considered implicitly urgent and will be performed as soon as they are enabled.

This interpretation of urgent internal behaviour, together with a hiding operator, can be used to make synchronisation urgent without causing a time-actionlock. The hiding operator turns observable into internal actions. Synchronisation results in an external action only when it is successful; thus, this external action can be hidden and therefore made urgent. As a result, mismatched synchronisation never blocks time. However, other form of timelocks may still occur (including both time-actionlocks and Zeno-timelocks). For instance, Zeno runs and Zeno-timelocks may be caused by unguarded recursion.

Bowman and Gomez [45] show how timelocks and Zeno runs may arise in a timed extension of the LOTOS process algebra. Schneider [144] explores similar issues in Timed CSP, and gives a static check to guarantee timelock-freedom and absence of Zeno runs.

3.7 Summary

We discussed the nature of different classes of timelocks in timed automata specifications, namely time-actionlocks and Zeno-timelocks. We explained the problem of timelocks and Zeno runs for verification: we cannot rely on verification unless the specification if timelock-free, and in some cases, also free from Zeno runs.

Time-actionlocks can be prevented by construction in Timed Automata with Deadlines, but Zeno-timelocks and Zeno-runs can still occur. Prevention of timelocks and Zeno runs in Timed Automata, on the other hand, is not possible. Therefore, we focused on detection methods. We studied the nature of liveness properties in Uppaal and Kronos, and the strong non-Zenoness property [156].

A liveness formula can be verified in Uppaal that denotes absence of timelocks and Zeno runs, but the algorithm involves a demanding nested reachability analysis, and rules out timelock-free specifications where Zeno runs occur. In Kronos, a liveness formula can be checked that is sufficient-and-necessary for timelock-freedom, but verification requires fixpoint algorithms which can be computationally expensive, and the product automaton (in order to verify networks of automata) must be constructed beforehand. Another limitation of Kronos is that absence of Zeno runs cannot be checked.

Strong non-Zenoness is compositional and holds in most specifications, and implies absence of Zeno-timelocks. A consequence of this observation is that, if strong non-Zenoness hold, timelock-freedom corresponds precisely to deadlock-freedom. This makes strong non-Zenoness a very appealing property to check: Deadlock-freedom is usually asserted before any other property is verified; thus, in most cases, strong non-Zenoness suffices to guarantee timelock-freedom (thus avoiding the more demanding checks of Uppaal and Kronos).

However, one problem with the definition of strong non-Zenoness is that it implies the detection of non-simple loops, which can be computationally expensive. Another limitation is that many useful specifications, which are both free from timelocks and Zeno runs, cannot be recognised as safe by the compositional application of strong non-Zenoness. In addition, strong non-Zenoness is of no use if we want to guarantee timelock-freedom but disregarding, at the same time, possible occurrences of Zeno runs. We point out, also, that strong non-Zenoness has not been implemented in any model-checker.

The next two chapters (Chapters 4 and 5) show that we can improve on the limitations of strong non-Zenoness, and obtain more efficient and comprehensive checks. This chapter also serves as background for Chapter 8, where we present a deductive framework (Discrete Timed Automata) that exploits the semantics of urgency and synchronisation of Timed Automata with Deadlines, in order to prevent time-actionlocks by construction.

Chapter 4

Zeno-timelocks: Static Checks

Abstract. We present static checks to guarantee the absence of Zeno runs and Zeno-timelocks in timed automata. We refine the strong non-Zenoness property (Chapter 3), which results in a more efficient and comprehensive analysis of networks of timed automata. These improvements better exploit the relationship among simple loops, synchronisation and strong non-Zenoness. In addition, we define new static checks based on the syntax of invariants in simple loops, which guarantee absence of Zeno-timelocks regardless of the occurrence of Zeno runs.

This chapter follows from the analysis of Zeno-timelocks, Zeno runs, and the shortcomings of strong non-Zenoness given in Chapter 3. In Chapter 5, we complement the static checks discussed in this chapter with semantic checks based on reachability analysis, and comment on a tool that we have implemented which performs these checks over Uppaal specifications.

Organization. In Section 4.1 we show how the application of strong non-Zenoness can be improved. We introduce invariant-based checks in Section 4.2. In Section 4.3 we present conclusions and suggestions for further research.

4.1 Weakening Strong Non-Zenoness

According to Lemma 3.3 (Section 3.5.2), in order to ensure that a network is free from Zeno runs, we have to check that *every* loop (i.e., in every component of the network) is strongly non-Zeno (SNZ). Note that, as was originally stated, we have to consider both simple and non-simple loops. Considering that the number of simple loops may be exponential in the worst case (in a complete graph, the number of elementary circuits is exponential in the number of vertices [89]), it is not difficult to see how searching for both, simple and non-simple loops, may easily yield the method intractable.

Interestingly, it turns out that the premises of Lemma 3.3 can be weakened in two ways.

1. We just need to consider simple loops in the checks of strong non-Zenoness: a non-simple loop is SNZ if and only if at least one of the constituent simple loops is SNZ.

2. Synchronisation preserves strong non-Zenoness, in the sense that it does not matter which loop a SNZ-loop will actually synchronise with; in any case, the result will be a loop in the product automaton which *is SNZ*. Hence, not every simple loop in a network must necessarily be SNZ to guarantee that the network is free from Zeno runs.

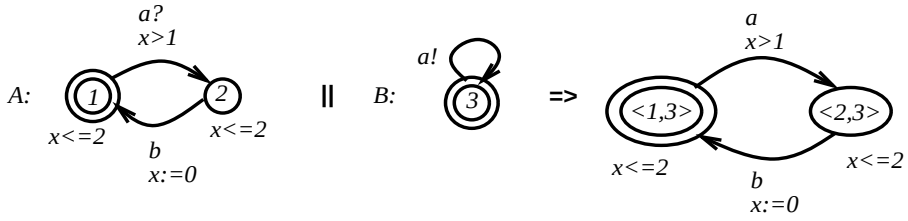


Figure 4.1: Composition Preserves Strong Non-Zenoness

Consider the network of Figure 4.1: the automaton A is SNZ, but B is not. According to Lemma 3.3 (i.e., the compositional results of strong non-Zenoness as originally worked out by Tripakis), this network cannot be considered safe (equivalently, the absence of Zeno runs cannot be confirmed): B contains a loop that is not SNZ. However, Lemma 3.3 is not taking into account that the behaviour of B is constrained (via synchronisation) by the behaviour of A; in other words, B can perform infinitely fast only if A is willing to do so. Therefore, because Zeno runs are not possible in A (A is SNZ), the network is free from Zeno runs. This can be confirmed if we examine the product automaton (right): the synchronisation of any loop (e.g., $\langle a! \rangle$ in B) with a SNZ-loop (e.g., $\langle a?, b \rangle$ in A), results only in SNZ-loops in the product (in this case, $\langle a, b \rangle$). These results are formalised by Lemma 4.1 and Theorem 4.2 below.

LEMMA **4.1.** *Let A be a timed automaton. If every simple loop in A is SNZ, then A is free from Zeno runs (and thus, A is free from Zeno-timelocks).*

Proof. Suppose, by contradiction, that every simple loop in A is SNZ but A is not free from Zeno runs. Let ρ be one such Zeno run. By definition, actions occur infinitely often in ρ, and so by Lemma 2.1 ρ covers some simple loop lp in A. However, because lp is SNZ, ρ must be divergent (by Lemma 3.1), which contradicts our assumption of ρ being a Zeno run. Hence, the lemma holds. □

Now, we can argue that a network is free from Zeno runs if the product automaton is free from Zeno runs (by construction, the product automaton is semantically equivalent to the original network). By Lemma 4.1, the product automaton is free from Zeno runs if every simple loop in it is SNZ. Interestingly, there is a nice structural dependency among simple loops in the product and simple loops in the network, which we can exploit to weaken the premises of Lemma 3.3.

By construction (see Section 2.1.1) a simple loop lp_π in the product must satisfy at least one of the following two conditions.

1. The loop lp_π contains the set of transitions of some completed simple loop in the network, lp_c (lp_π might, in addition, be composed of transitions which belong to other simple loops in the network). In particular, if a clock x is reset and bounded from below in lp_c, then so is it in lp_π.

2. The loop lp_π contains the set of transitions that are derived from the synchronisation of at least two matching simple loops in the network (lp_π might, in addition, be composed of transitions which belong to other simple loops in the network). Let lp_h be any of these matching half loops. If a clock x is reset and bounded from below in lp_h, then so is it in lp_π (remember: synchronisation unions reset sets, and conjoins guards).

Assume that every completed loop in a network is SNZ, and that, for every pair of matching loops in the network, *at least one* of these loops is SNZ. Necessarily, then, every loop in the product automaton must be SNZ (even when some half loops may not be SNZ). Effectively, we have weakened Lemma 3.3, but we can still guarantee the same result.

THEOREM **4.2.** *Let $|A = |\langle A_1, \ldots, A_n \rangle$ be a network of timed automata. Let $HL(|A)$ be the set of matching half loops (simple loops), and $CL(|A)$ the set of completed loops (simple loops) in the network, where*

$$\begin{aligned} HL(|A) = \{\, (lp, lp') \mid \exists\, i, j\, (1 \leq i \neq j \leq n)\,.\, lp \in Loops(A_i) \wedge lp' \in Loops(A_j) \wedge \\ \exists\, a? \in Act(lp).\, a! \in Act(lp') \,\} \\ CL(|A) = \{\, lp \mid \exists\, i\, (1 \leq i \leq n)\,.\, lp \in Loops(A_i) \wedge \forall\, a \in Act(lp)\,.\, a \in CAct \,\} \end{aligned}$$

If at least one loop in every pair in $HL(|A)$ is SNZ and every loop in $CL(|A)$ is SNZ, then the product automaton obtained from $|A$ is free from Zeno runs. Equivalently, the network $|A$ is free from Zeno runs (and thus, free from Zeno-timelocks).

Proof. Assume that at least one loop in every pair in $HL(|A)$ is SNZ and every loop in $CL(|A)$ is SNZ. Note that both sets, $HL(|A)$ and $CL(|A)$, contain only simple loops. Consider the structure of a simple loop in the product automaton, lp_π: it is either derived from some completed simple loop in the network, or from two or more matching simple loops in the network.

Case 1. The loop lp_π contains the set of transitions of some completed simple loop in the network, lp_c. In particular, the following holds.

$$\begin{aligned} \mathit{Resets}(lp_\pi) &\supseteq \mathit{Resets}(lp_c) \\ \mathit{Guards}(lp_\pi) &\supseteq \mathit{Guards}(lp_c) \end{aligned}$$

Let x be the clock that witnesses the strong non-Zenoness of lp_c (every loop in $CL(|A)$ is SNZ), i.e., x is reset and bounded from below in lp_c. It is not difficult to see that x is also reset and bounded from below in lp_π. Then, lp_π is SNZ.

Case 2. The loop lp_π contains the set of transitions that are derived from the synchronisation of at least two matching simple loops in the network. Let lp_1, lp_2 be two such matching loops, such that lp_1 is SNZ (at least one loop in every pair in $HL(|A)$ is SNZ). In particular, the following holds.

$$\begin{aligned} \mathit{Resets}(lp_\pi) &\supseteq \mathit{Resets}(lp') \cup \mathit{Resets}(lp'') \\ \mathit{Guards}(lp_\pi) &\supseteq \{g \mid l \xrightarrow{a,g,r} l' \in \mathit{Trans}(lp') \cup \mathit{Trans}(lp'')\} \cup \\ &\quad \{g_i \wedge g_j \mid l_i \xrightarrow{a?,g_i,r} l'_i \in \mathit{Trans}(lp') \wedge l_j \xrightarrow{a!,g_j,r} l'_j \in \mathit{Trans}(lp'')\} \end{aligned}$$

Let x be a clock that is reset and bounded from below in lp_1. Then, x is also reset and bounded from below in lp_π. Hence, lp_π is SNZ. Then, every simple loop in the product is SNZ, and so (by Lemma 4.1) the product is free from Zeno runs. Hence, the network is free from Zeno runs (and thus, free from Zeno-timelocks). □

Note that we have fundamentally enhanced the applicability of strong non-Zenoness. On the one hand, we have shown that absence of Zeno runs can be guaranteed by considering only simple loops, which drastically improves tractability. On the other hand, our (weakened) compositional application of strong non-Zenoness is able to recognise a broader and more useful class of specifications than was previously possible with Lemma 3.3.

Many specifications, and in particular, those that result from constraint-oriented designs [162, 30, 144], exhibit a design pattern similar to the one shown in Figure 4.1. Frequently, non-SNZ components are abstract components that define an overall behavior (e.g., where the main concern is the temporal ordering of events, rather than the exact timing of these events), which can be constrained by other, more specific SNZ components through parallel composition (synchronisation). Such specifications are common in the literature of real-time systems; for instance, in the train-gate-controller model of [22], the lip-synch protocol of [43], and the gear-box controller of [108].

4.2 Invariant-based Properties

Sometimes, when we are interested in detecting Zeno-timelocks but the occurrences of Zeno runs are irrelevant to verification, strong non-Zenoness may be too restrictive a property to check.

Consider the loop of Figure 4.2(i). This could represent a system that toggles between two states: in location 1, it can perform action a at any time and switch to location 2, but it can only remain there for at most 2 time units before switching back to location 1. This loop is not SNZ, and indeed Zeno runs may occur (which toggle between location 1 and 2 infinitely fast). However, this specification is timelock-free, and may be an acceptable abstraction of the system (e.g., at this stage, the relative speed between a and b may not be relevant).

Consider, now, the loop of Figure 4.2(ii). Here the system is known to remain a maximum of 3 time units in *both* locations, and no more than 2 time units in location 1, but the exact time spent in each location varies in every iteration. Again, this loop is not SNZ; nonetheless, we would argue that it is a natural specification for such behaviour. Note, in addition, the loop contains Zeno runs but is timelock-free: x is reset in the loop, so time may always pass up to 2 time units before the next iteration (thus, time can always diverge).

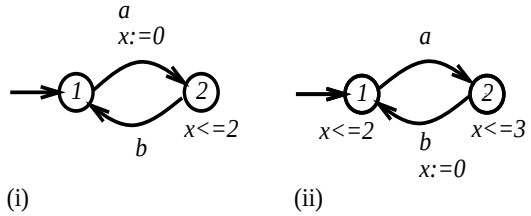


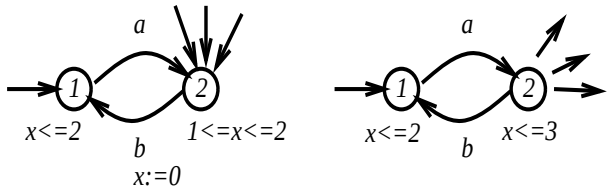
Figure 4.2: Invariant-based Properties

We can go one step further, and derive syntactic properties that characterise specifications such as those of Figure 4.2. If at least one location in the loop does not constrain time (e.g., location 1 in Figure 4.2(i)), or at least one invariant in the loop imposes upper bounds on clocks that are reset in the loop (e.g., location 1 in Figure 4.2(ii)), then such a loop cannot prevent time-divergence.[1]

Similarly, other properties can be derived that also consider location invariants. Note that our timed automata model of Chapter 2 permits lower-bounded invariants (even if these, for technical reasons, are not allowed in model-checkers such as Uppaal or Kronos). Consider the model of Figure 4.3(i), where it is known that location 2 cannot be entered from anywhere unless $v(x) > 1$; this can be conveniently modelled using a lower-bounded invariant. The loop $\langle a, b \rangle$ is not SNZ, but it does not cause a Zeno-timelock (in fact, it does not cause Zeno runs): location 2 cannot be entered unless at least 1 time unit passes between consecutive iterations.

[1] Originally [47], we called these properties "location non-urgency" and "reset non-urgency", respectively.

A different scenario can be observed in Figure 4.3(ii). The loop can be performed any number of times during the first 2 time units; however, at any time during the initial 3 time units the execution may abandon the loop via location 2. This loop is not SNZ and contains Zeno runs, but it does not cause Zeno-timelocks. Because of the difference between upper-bounds in locations 1 and 2, a valuation can be reached in location 2 that prevents b from being performed, i.e., a state can always be reached where no further loop iterations are possible. Note that, solely from the structure of the loop, we cannot guarantee that a time-actionlock does not occur (e.g., location 2 may not have enabled outgoing transitions when $v(x) = 3$).



(i) (ii)

Figure 4.3: More Invariant-based Properties

As we did before for Figure 4.2, we can extract general properties from Figure 4.3. If there is a clock that is reset in the loop, and bounded from below in some location of the loop, then that loop does not cause Zeno runs (e.g., location 2 in Figure 4.3(i)). If there is at least one location in the loop where all upper-bounds are strictly greater than the smallest upper bound for the corresponding clocks (e.g., in Figure 4.3(ii), the invariant of location 2 imposes an upper-bound of 3 to clock x, whose smallest upper bound is 2), then that loop does not cause Zeno-timelocks (it may, though, cause time-actionlocks).

In a sense, the properties illustrated by Figures 4.2 and 4.3 are "weaker" than strong non-Zenoness; they can guarantee the absence of Zeno-timelocks, but not the absence of Zeno runs. On the other hand, these properties consider the syntax of loop invariants and are able to recognise a class of safe specifications that are not SNZ.

We formally enunciate these *invariant-based* properties in Lemma 4.7; this is used later to derive a syntactic method to prove the absence of Zeno-timelocks in a given automaton (Theorem 4.8). We first introduce a number of auxiliary concepts and lemmas.

Informally, a loop that "causes" a Zeno-timelock determines the occurrence of *covering Zeno-timelocks*; otherwise the loop is said to be *inherently safe*.

LEMMA **4.3.** *Every state reachable from a Zeno-timelock is also a Zeno-timelock.*

Proof. This lemma follows from the definition of Zeno-timelocks (Section 3.2). □

LEMMA **4.4.** *Let l be a given automaton's location with invariant $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$. The following two conditions hold.*

1. *If l is reachable, then there exists a finite run ρ that remains in l while the invariant holds. Moreover, the last state in ρ is $s = [l, v]$, where $v(x_i) = c_i$, for some $1 \leq i \leq n$.*
2. *Time can pass in l only if $c_i > 0$, for all $1 \leq i \leq n$.*

Proof. This lemma is a consequence of the semantics of invariants in timed automata (see Section 2.1.2). □

Covering Zeno-timelocks. Let lp be a loop, and s a Zeno-timelock. We say that s *covers* lp if every finite run starting from s can be extended to a run that covers lp. Formally, a Zeno-timelock s covers lp if $\forall\, \rho \in \mathit{FiniteRuns}(s).\, \exists\, \rho' \in \mathit{CoveringRuns}(s, lp).\, \rho \subseteq \rho'$.

LEMMA **4.5.** *From any Zeno-timelock, another Zeno-timelock can be reached that covers some simple loop.*

Proof. Let s_0 be a Zeno-timelock, and s_1 be a Zeno-timelock reachable from s_0 (by Lemma 4.3, every state reachable from s_0 is a Zeno-timelock). Let ρ_1 be any Zeno run starting from s_1. By Lemma 2.1, ρ_1 must cover some simple loop lp_1 (i.e., ρ_1 visits lp_1 infinitely often). Suppose, by contradiction, that the lemma does not hold. In other words, there is no state s' reachable from s_1, and no simple loop lp', such that s' covers lp'. Then, after visiting lp_1 a finite number of times, ρ_1 must reach a state s_2, from which lp_1 cannot be visited again.

Because there are only finitely many simple loops to visit, we can apply our previous reasoning to postulate the existence of some state s_n, from which no simple loop can ever be visited again. This implies that no run starting from s_n can be extended to a Zeno run (once s_n is reached, no action can occur infinitely often). By definition, s_n cannot be a Zeno-timelock, which contradicts Lemma 4.3. Hence, our initial contradiction cannot be justified, and the lemma holds. □

COROLLARY **4.6.** *A Zeno-timelock occurs if and only if a Zeno-timelock occurs that covers a simple loop.*

Inherently Safe Loops. A (simple or non-simple) loop lp is *inherently safe* if there is no Zeno-timelock that covers lp (note, by Lemma 3.1, that any run that covers a SNZ-loop is necessarily divergent, and so SNZ-loops are inherently safe).

LEMMA **4.7.** *Let lp be a (simple or non-simple) loop, such that*

- *(Prop. 1) there is an invariant in lp where no clock is bounded from above; or*
- *(Prop. 2) there is an invariant in lp,* $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$*, where* $x_i \in Resets(lp)$ *and* $c_i > 0$ *for all* $1 \leq i \leq n$*; or*
- *(Prop. 3) there is a clock that is reset in lp and bounded from below in some invariant of lp; or*
- *(Prop. 4) there is an invariant in lp,* $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$*, where* $c_i > c_{\min}(x_i, lp)$ *for all* $1 \leq i \leq n$*.*

Then, lp is inherently safe.

Proof. Let lp be a loop that satisfies at least one of the properties stated by this lemma. Suppose, by contradiction, that there exists a Zeno-timelock, s, which covers lp. Consider the possible cases.

(Prop. 1) Let $l \in Loc(lp)$, s.t. no clock in $I(l)$ is bounded from above. Then, $I(l)$ cannot prevent time from passing. Hence, if l is reached, a divergent run (starting from l) is possible.

(Prop. 2) Let $l \in Loc(lp)$ s.t. $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, $x_i \in Resets(lp)$ and $c_i > 0$, for all $1 \leq i \leq n$. By definition, every finite run starting from s can be extended to a run that covers lp. Let $\rho(s)$ be a run that covers lp, s.t. every time it reaches l, it remains there while the invariant holds (Lemma 4.4). Every time that ρ iterates over lp, all the clocks in $I(l)$ are reset before l is reached. Because ρ remains in l while the invariant holds, every iteration has a delay of at least d time units, where $d = min(c_1, c_2, \ldots, c_n)$ (note that, $d > 0$ because $c_i > 0$, for all $1 \leq i \leq n$). Hence, ρ must be a divergent run.

(Prop. 3) Let $x \in Resets(lp)$, and $l \in Loc(lp)$ s.t. x is bounded from below in $I(l)$. Let t_1 be the transition in lp where x is reset, and t_2 be the transition in lp that is ingoing to l. Every time a run reaches l, it must be the case that $v(x) \geq c$, $(c \in \mathbb{N}, c > 0)$. Note that, at least c time units must pass between the execution of t_1 and t_2. Hence, any run that covers lp is divergent.

(Prop. 4) Let $l \in Loc(lp)$ s.t. $I(l) \triangleq \bigwedge_{i=1}^{n} x_i \leq c_i$, where $c_i > c_{\min}(x_i, lp)$ for all $1 \leq i \leq n$. By definition, every finite run starting from s can be extended to a run that covers lp. Let $\rho(s)$ be a run that covers lp, s.t. every time it reaches l, it remains there while the invariant holds (Lemma 4.4). Let y be a clock occurring in $I(l)$ (i.e., $y = x_i$, for some $1 \leq i \leq n$). Let $l' \neq l$ be

a location in lp where y is bounded from above by its smallest upper bound (i.e., $y \leq c_{\min}(y, lp)$ is a term in $I(l')$). We assumed that ρ covers lp, and that remains in l while the invariant holds. In particular, every time ρ leaves l, the value of y is such that $v(y) > c_{\min}(y, lp)$. Then, y must be reset before ρ reaches l' again (otherwise, the invariant in l' would prevent this location to be entered). But if this is the case, then every iteration has a delay of at least d time units, where $d = min(c_1, c_2, \ldots, c_n)$ (again, note that $d > 0$ because $c_i > 0$, for all $1 \leq i \leq n$). Hence, ρ must be a divergent run.

We proved that, if any of the four properties stated in the lemma are satisfied, and a Zeno-timelock s occurs that covers lp, then a divergent run must exists that starts from s. This contradicts the assumption that s is a Zeno-timelock. Hence, either s is not a Zeno-timelock, or it is a Zeno-timelock that does not cover lp: by definition, lp is inherently safe. □

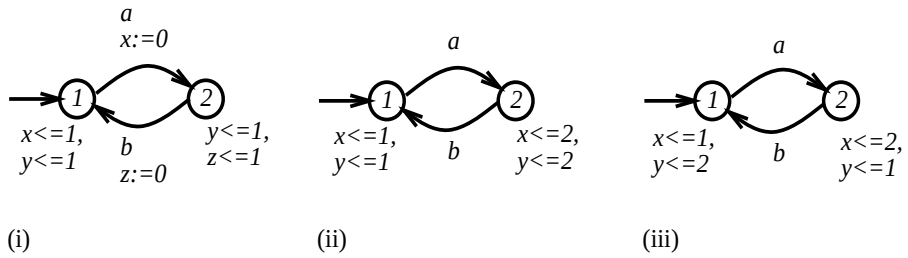


Figure 4.4: On the Nature of Property (4)

We have seen examples of loops that satisfy properties (1), (2), (3) and (4) in Figures 4.2(i) and (ii), and Figures 4.3(i) and (ii), respectively. Figure 4.4 further illustrates the check of property 4. The loop in Figure 4.4(i) does not satisfy property (4): every invariant imposes the same (smallest) upper bound on y ($c_{\min}(y, lp) = 1$). In particular, the state $s = [1, v]$ is a Zeno-timelock, where $v(y) = 1$ and $v(x) = v(z) = 0$.

On the other hand, the loop in Figure 4.4(ii) satisfies property (4) because all conjuncts in the invariant of location 2 refer to constants that are greater than the smallest upper bound (for every clock). Note that the difference between the upper bounds in locations 1 and 2 confirms that time is allowed to pass by at least 1 time unit in location 2 (if so, we will end up with a time-actionlock, but no Zeno-timelock can cover this loop).

Finally, the loop in Figure 4.4(iii) shows a slightly different arrangement of upper bounds, but does not satisfy property (4). Note, that there is no invariant where every clock is greater than its smallest upper bound. In fact, a Zeno-timelock occurs in $s = [1, v]$, where $v(x) = v(y) = 1$.

THEOREM **4.8.** *Let A be a timed automaton, where every simple loop in A either is SNZ or satisfies Lemma 4.7. Then, A is free from Zeno-timelocks.*

Proof. Assume that every simple loop in A either is SNZ, or it satisfies some condition enumerated in Lemma 4.7. Then, by Lemmas 3.1 and 4.7, no simple loop in A can be covered by a Zeno-timelock. By Corollary 4.6, A is free from Zeno-timelocks. □

The Nature of Invariant-based Properties. The application of Theorem 4.8 is static (thus, no model-checking is necessary), and only simple loops need to be considered. Invariant-based properties complement strong non-Zenoness to detect a broader class of safe specifications (these properties, invariant-based and strong non-Zenoness, do not imply each other).

The invariant-based properties (1), (2) and (4) guarantee the absence of Zeno-timelocks, but not the absence of Zeno runs. These properties are not compositional. For example, both automata in Figure 4.5(i) satisfy property (1), because locations 2 and 4 have *true*-invariants. However, composition results in a product automaton with a single loop, where a Zeno-timelock occurs at $s = [\langle 1, 3\rangle, v]$ with $v(x) = v(y) = 1$. Similarly, the component automata of Figure 4.5(ii) can be considered safe according to Theorem 4.8, because the loops satisfy property (4), but again a Zeno-timelock occurs at $s = [\langle 1, 3\rangle, v]$ with $v(x) = v(y) = 1$.

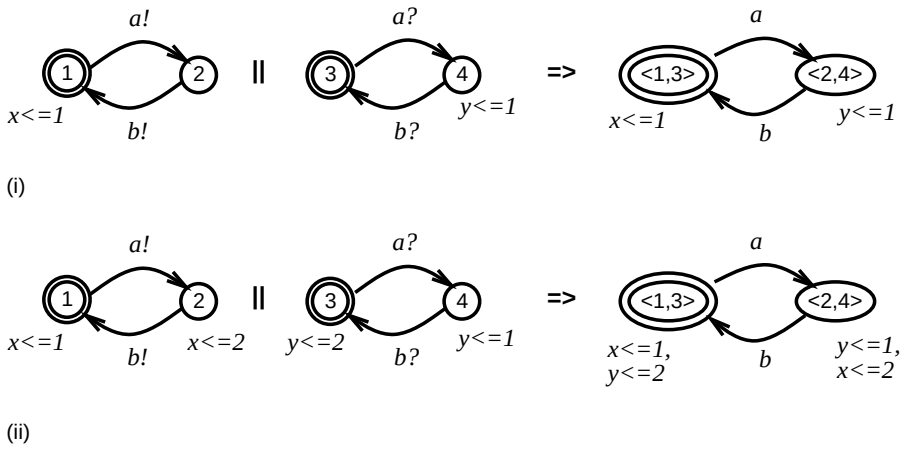


Figure 4.5: Noncompositional Syntactic Conditions

Property (3), on the other hand, does guarantee absence of Zeno runs (see proof of Lemma 4.7) and is compositional (the proof of this, which we omit here, is similar to that of Theorem 4.2).

All invariant-based properties represent a sufficient-only condition for absence of Zeno-timelocks (and so, by extension, Theorem 4.8 is also sufficient-only). For example, the automaton of Figure 4.6 is free from timelocks, but it does not satisfy Theorem 4.8: the loop does not satisfy any of the invariant-based properties (nor is it a SNZ-loop). The key point here

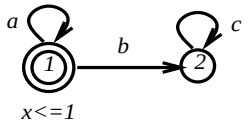


Figure 4.6: A Timelock-free Automaton (Theorem 4.8 is Sufficient-only)

is that even when Zeno runs do exist (e.g., the run starting in location 1 that remains there, performing an infinite number of a-transitions in 1 time unit), there is no state in the model that prevents divergent runs. Note that transition b is always enabled in location 1, and time is always allowed to diverge in location 2.[2]

4.3 Conclusions

The strong non-Zenoness property [156] is a condition on guards and resets in loops, which guarantees absence of Zeno runs in networks of timed automata. Strong non-Zenoness is static, compositional, holds in most specifications, and suffices to guarantee timelock-freedom (if the specification is deadlock-free). Thus, in most cases, a simple check for strong non-Zenoness avoids the more demanding verification of liveness properties, which characterise the support for timelock detection available in model-checkers (see Sections 3.5.1 and 3.5.2).

However, checking for strong non-Zenoness (according to the original definition) requires the detection of non-simple loops, which may be computationally expensive. Another limitation is that all loops, in every component of the network, are required to be SNZ to guarantee absence of Zeno runs. We offered a number of solutions to these problems.

- We refined the application of strong non-Zenoness to derive a more efficient and comprehensive check. We proved that, in order to guarantee absence of Zeno runs, (a) it is sufficient to check strong non-Zenoness just on simple loops (we proved that a non-simple loop is SNZ if and only if any of its constituent simple loops is SNZ), and (b) not every loop in the specification is required to be SNZ (we proved that synchronisation of a SNZ-loop with any other loop is safe). With these improvements, a class of systems that are free from Zeno runs, but where not every loop is SNZ, can now be positively analysed.

- We introduced new static checks to guarantee absence of Zeno-timelocks in a timed automaton, based on the syntax of invariants in simple loops. These checks are able to recognise specifications that are free from Zeno-timelocks, but which may not be SNZ.

[2]This is related to the notion of escape transitions, which we exploit in Section 5.1 to define sufficient-and-necessary checks.

Therefore, together with strong non-Zenoness, invariant-based properties define a more comprehensive check for absence of Zeno-timelocks.

These improvements suggest that strong non-Zenoness should be our first sanity check (i.e., to assert whether the specification is well-timed); if this check is inconclusive, then we can verify the liveness properties λ_U (in Uppaal) and λ_K (in Kronos), if we are only interested in guaranteeing timelock-freedom (Section 3.5.1). In addition, our strong non-Zenoness check simplifies verification in Red [165] and Profounder [159], which must run demanding algorithms to detect and avoid Zeno runs.

On the other hand, we may need to check timelock-freedom in specifications where Zeno runs occur, but where such runs do not affect the verification of our requirements (e.g., safety properties). In such situations, checking strong non-Zenoness (or verifying the property λ_U in Uppaal) will give inconclusive results. In response to these limitations of strong non-Zenoness and Uppaal's detection method, we proposed a number of invariant-based properties (Section 4.2), which guarantee absence of Zeno-timelocks but are insensitive to Zeno runs.

Kronos can detect timelock-freedom with the liveness formula λ_K, which is equivalent to the TCTL formula $\forall\Box\exists\Diamond_{=1}\, true$. This embodies a sufficient-and-necessary condition for timelock-freedom, and thus it is insensitive to the occurrences of Zeno runs. However, Kronos cannot detect absence of Zeno runs (thus, it would benefit from our strong non-Zenoness check), and TCTL is not available to most model-checkers (most tools can only perform reachability analysis). In contrast, our static checks are available to all timed automata specifications, and so they can be implemented in all model-checkers.

Nonetheless, depending on the specification at hand, the checks available in Uppaal or Kronos may be the best choice. For instance, the invariant-based conditions of Section4.2 require the product automaton to be built a priori; checking for timelock-freedom in Uppaal may be more efficient if a timelock is detected after exploring just a small part of the reachability graph (Uppaal's verification is on-the-fly).

Another advantage of Uppaal and Kronos is that verification is performed over richer specification languages. For instance, Uppaal's automata may include variables and richer reset conditions, while the strong non-Zenoness property has been defined over timed automata models where clocks are reset to zero and lower-bounds are constants.

Interestingly, the application of strong non-Zenoness could be further weakened, to account for more complex synchronisation scenarios. For example, Figure 4.7 shows three synchronising loops, where only one of them is SNZ. We can see (right) that synchronisation is safe. However, we cannot determine this if we only check the set of loops pairwise, as Theorem 4.2 suggests (Section 4.1). Instead, we could sacrifice some efficiency in favor of a more accurate check, which gathers the component loops that are necessary to complete synchronisation.

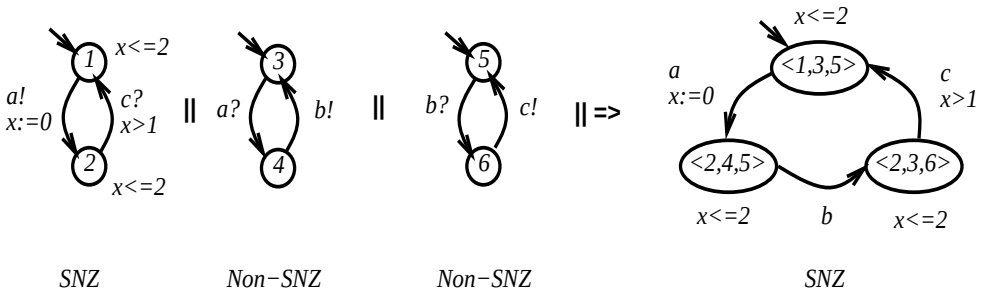


Figure 4.7: Exploiting Synchronisation in Strong Non-Zenoness

To conclude, we point out that the static checks presented in this chapter are sufficient-only, thus they may be inconclusive for certain safe specifications. In the following chapter (Chapter 5), we will present semantic checks based on reachability analysis, which denote sufficient-and-necessary conditions. In addition, we will comment on a tool that we have implemented, which performs our static and semantic checks over Uppaal specifications.

Chapter 5

Zeno-timelocks: Semantic Checks

Abstract. We present sufficient-and-necessary checks for the absence of Zeno-timelocks and Zeno runs in a timed automaton. These checks are based on reachability formulae that are derived from the structure of loops, which characterise the occurrence of Zeno-timelocks and Zeno runs in the loops. We propose a strategy to assert absence of Zeno runs and Zeno-timelocks in networks of timed automata, which combines the static checks of Chapter 4 (strong non-Zenoness and invariant-based properties) with the reachability-based properties discussed in this chapter. We show that reachability analysis needs to be performed only for those loops that static checks could not rule out as safe. Furthermore, we have implemented a tool that performs these checks over Uppaal specifications.

Organization. We develop the theory behind our semantic checks in Section 5.1. In Section 5.2 we introduce a reachability-based property that characterises the occurrence of Zeno-timelocks in a loop. Similarly, we characterise the occurrence of Zeno runs in Section 5.3. In Section 5.4, we propose a layered strategy to combine static and semantic checks. Section 5.5 describes a tool that we have implemented, which performs both static and semantic checks over Uppaal specifications. In Section 5.6, we use the CSMA/CD protocol as a case study. We conclude this chapter in Section 5.7, and comment on further research.

5.1 Preliminaries

Static checks, such as strong non-Zenoness and the invariant-based properties that we presented in Chapter 4, made evident that the structure of loops allows us to infer the absence of Zeno runs and Zeno-timelocks. However, specifications may be free from timelocks and Zeno runs even if static checks do not hold; in other words, syntax alone does not suffice to infer that Zeno runs or Zeno-timelocks *will* occur. We need to complement static checks (easier to implement, and

more efficient) with semantic, sufficient-and-necessary checks (more conclusive). In particular, we must address the following issues.

- **Simple vs. Non-simple Loops.** In general, properties that consider simple loops (e.g., strong non-Zenoness) check whether the structure of the loop prevents Zeno runs. However, even if Zeno runs are possible, this does not necessarily mean that Zeno-timelocks will occur. If we only consider simple loops, we cannot see what happens to Zeno runs when (and if) they abandon a loop: Zeno runs may reach a state outside the loop where time diverges (thus, a Zeno-timelock cannot occur), or they may remain "trapped" among simple loops where all actions are urgent (i.e., a Zeno-timelock has occurred, but the Zeno runs are confined to some non-simple loop). The interaction between Zeno-timelocks and loops is characterised by the concept of *locality*, which we will formalise in this section.

- **Time-divergence.** Zeno-timelocks may occur in runs where time cannot diverge, but where delays are always possible (e.g., time may pass by a series of decreasing delays $0.5 + 0.25 + 0.125 + \ldots$). To simplify our reachability-based properties, we assume that whenever a Zeno-timelock occurs, a valuation is eventually reached that remains constant, i.e., runs thereafter do not change the values of clocks. Hence, we restrict our analysis to loops that contain only *true*- or *right-closed* invariants. Right-closed invariants are represented by the following BNF,

  $$I ::= x \leq c \mid I \wedge I$$

  where I is a right-closed invariant, x is a clock, and $c \in \mathbb{N}$ is a constant.[1]

- **Synchronisation.** It seems impossible to obtain sufficient-and-necessary checks that, in addition, are compositional: unresolved synchronisation brings uncertainty to the analysis. Therefore, we must restrict our analysis to timed automata that contain only completed actions (thus, in order to analyse a network, the checks will be applied to the product automaton).

The remainder of this section introduces concepts and lemmas that support the definition of sufficient-and-necessary checks for Zeno-timelocks (Section 5.2) and Zeno runs (Section 5.3). Throughout this chapter, A denotes a timed automaton that contains only completed actions, and where all loops contain only *true*- or right-closed invariants.

[1] This is not a severe restriction: most specifications are given in terms of right-closed invariants only.

Local Runs. Let lp be a loop and ρ a run. We say that ρ is *local* to lp if it only visits transitions of lp. Formally, ρ is local to lp if $\mathit{Trans}(\rho) \subseteq \mathit{Trans}(lp)$. We use $\mathit{LocalRuns}(s, lp)$ to denote the set of all runs starting from s that are local to lp.

Local Zeno-timelocks. Let lp be a loop, and s a Zeno-timelock that covers lp. We say that s is *local* to lp if, once s is reached, only transitions in lp can be visited (note that, because s covers lp, lp can be visited infinitely often from s). Formally, a Zeno-timelock s is local to lp if $\mathit{Runs}(s) = \mathit{LocalRuns}(s, lp)$.

By way of example, Figure 5.1 (i) shows that the state $s = [1, v]$ $(v(x) = 1)$ is a Zeno-timelock local to the (non-simple) loop $\langle a, b, d, c, d\rangle$. Note that, if a Zeno-timelock is local to some loop lp, then it also covers lp, but the converse is not always true.

For example, in (i), $s = [1, v]$ $(v(x) = 1)$ is a Zeno-timelock that *covers* the simple loop $\langle c, d\rangle$, because every finite run starting from s can be extended to a run that visits c and d infinitely often. However, s is *not local* to $\langle c, d\rangle$; there are runs starting from s that visit a and b, which are not part of the loop. For the same reason, s is not local to the simple loop $\langle a, b, d\rangle$ either. In some specifications, then, Zeno-timelocks may occur that are only local to non-simple loops. In contrast, Figure 5.1 (ii) shows that $s' = [1, v']$ $(v'(x) = 2)$, is a Zeno-timelock local to the simple loop $\langle c, d\rangle$; once s' is reached, neither a nor b are enabled.

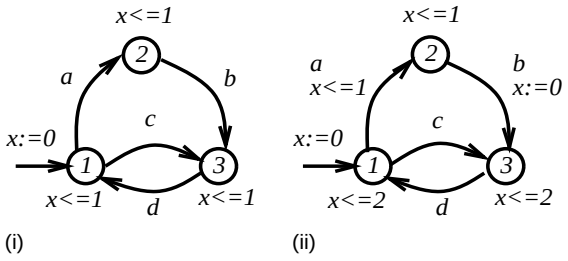


Figure 5.1: Simple Loops, Non-simple Loops and Local Zeno-timelocks

LEMMA **5.1.** *From any Zeno-timelock, another Zeno-timelock can be reached that is local to some (simple or non-simple) loop.*

Proof. Let s be a Zeno-timelock (remember, by Lemma 4.3, that any state reachable from s is also a Zeno-timelock). By Lemma 4.5, there exists a simple loop, lp_1, and a Zeno-timelock, s_1, that covers lp_1 and is reachable from s. By definition, every finite run starting from s_1 can be extended to a run that covers lp_1. Given s_1 and lp_1, only two cases are possible. An analysis of each case follows.

Case 1. There exists s'_1 reachable from s_1, s.t. runs starting from s'_1 visit only transitions in lp_1 (i.e., every run starting from s'_1 is local to lp_1). By definition, s'_1 is a Zeno-timelock local to lp_1. Hence, we have proved our claim.

Case 2. Case 1 is not true; i.e., from every state that is reachable from s_1, some transition outside lp_1 can always be visited. In fact, because lp_1 can be visited infinitely often, there must exist s_2 reachable from s_1, and a set of transitions $T_2 \supset \mathit{Trans}(lp_1)$, s.t. every transition in T_2 can be visited infinitely often, from any state that is reachable from s_2. Note that T_2 represents a non-simple loop lp_2, s.t. $\mathit{Trans}(lp_2) = T_2$. Again, the same two cases are possible for s_2 and lp_2, and we apply the same reasoning.

Note that, every time that *Case 1* does not hold, a bigger non-simple loop can be built (according to *Case 2*). But there are only finitely many transitions to try. Thus, our reasoning can only be applied a finite number of times, and *Case 1* must eventually hold for some state s_n and non-simple loop lp_n, where

$$\mathit{Trans}(lp_n) \supset \mathit{Trans}(lp_{n-1}) \supset \ldots \supset \mathit{Trans}(lp_1)$$

This state s_n is a Zeno-timelock local to lp_n, and s_n is reachable from s. Together, s_n and lp_n justify the lemma. □

COROLLARY **5.2.** *A Zeno-timelock occurs if and only if a Zeno-timelock occurs that is local to a (simple or non-simple) loop.*

Converged Zeno-timelocks and Maximal Valuations. Let $s = [l, v]$ be a Zeno-timelock. We say that s is a *converged* Zeno-timelock if no valuation, other than v, is reachable from s. Formally, a Zeno-timelock $s = [l, v]$ is a converged Zeno-timelock if $\forall\, l', v'.\, (s \stackrel{*}{\Longrightarrow} [l', v']) \;\Rightarrow\; (v' = v)$. In addition, we say that v is *maximal* w.r.t. $\mathit{Runs}(s)$.

Figure 5.1(i) serves to illustrate the concept of converged Zeno-timelocks. Every state that is reachable in $\langle a, b, d, c, d\rangle$ is a Zeno-timelock, but only those where $v(x) = 1$ are converged Zeno-timelocks (no clock can ever change its value after $v(x) = 1$).

LEMMA **5.3.** *From any Zeno-timelock, another Zeno-timelock is reachable from which no further delay is possible.*

Proof. Let s be a Zeno-timelock. Suppose, by contradiction, that every state that is reachable from s allows some delay (although, time cannot diverge). By definition of Zeno-timelock, every finite run starting from s can be extended to a Zeno run. In addition, all locations that are reachable from s must have a right-closed invariant (*true*-invariants would imply the existence

of divergent runs, by Lemma 4.7). Therefore, it is possible to build a Zeno run, ρ, that starts in s and remains in every location it visits while the invariant holds (Lemma 4.4).

Since there are only finitely many locations to visit, there must exist a location l that is visited infinitely often by ρ, s.t. that some delay is possible whenever l is entered. By Lemma 4.4, and because ρ remains in l while the invariant holds, there is a term $y \leq c$ in $I(l)$, s.t. $v(y) = c$ $(c \in \mathbb{N},\ c > 0)$ every time that ρ leaves l. Note that once l is left, ρ must reset all clocks in $I(l)$ before it can visit l again (including the clock y). As ρ visits l infinitely often, it must accumulate an infinite number of delays $d > 0,\ d \in \mathbb{N}$; i.e., it is a divergent run (this contradicts our assumption that ρ is a Zeno run).

We proved that if s is a Zeno-timelock, some state s' must exist that is reachable from s, where time cannot pass any further (by Lemma 4.3, s' is itself a Zeno-timelock). Hence, the lemma holds. □

LEMMA **5.4.** *From any Zeno-timelock, a converged Zeno-timelock is reachable.*

Proof. Let s be a Zeno-timelock. By Lemmas 5.1 and 5.3, there exists a Zeno-timelock s' reachable from s, which is local to some loop lp, and where no further delay is possible. Consider the set $R \subseteq \mathit{Resets}(lp)$ of all clocks that are reset by any run starting from s'. Note that, R is finite and a state s'' must exist, which is reachable from s', such that every clock in R has been reset at least once before s'' is reached.

In our timed automata model, a clock value can change only as a result of delays (the value increases) or resets (the value is set to zero). This observation, and the fact that further time passing is not possible from s', has two consequences. First, once a clock has been reset after s' is reached, its value becomes permanently zero. In particular, the following holds,

$$\forall\, x \in R, l, v.\, ((s'' \stackrel{*}{\Longrightarrow} [l, v]) \Rightarrow v(x) = 0)$$

Secondly, if a clock in the automaton is never reset after s' is reached, then its value remains constant (it maintains the value it had when s' was reached). In particular, the following holds,

$$\forall\, x \notin R, l_1, l_2, v_1, v_2.\, ((s'' \stackrel{*}{\Longrightarrow} [l_1, v_1] \stackrel{*}{\Longrightarrow} [l_2, v_2]) \Rightarrow v_1(x) = v_2(x))$$

Hence, once s'' is reached, the valuation remains constant. By definition, s'' is a converged Zeno-timelock, and the lemma holds. □

COROLLARY **5.5.** *Any state reachable from a converged Zeno-timelock is also a converged Zeno-timelock.*

Converged Zeno-timelocks denote valuations with some particular features, which makes them easier to identify. For some loop in the automaton, we want to determine whether a converged Zeno-timelock *may* occur, which is local to this loop. Let us refer to such loops as *Zeno loops*.

Zeno Loops And Maximal Valuations. We say that a loop lp is a *Zeno loop* if there exists a state s reachable in lp, s.t. once s is reached, lp can be covered by local runs, but none of these runs can pass time. Formally, lp is a Zeno loop if there exists a reachable state $s = [l, v]$, where $l \in \mathit{Loc}(lp)$, s.t. $\mathit{LocalRuns}(s, lp) \cap \mathit{CoveringRuns}(s, lp) \neq \emptyset$, and v is maximal w.r.t. $\mathit{LocalRuns}(s, lp)$. We refer to such v as a maximal valuation of lp.[2]

The syntactic structure of Zeno loops plays a role in the reachability of maximal valuations. Indeed, if lp is a Zeno loop and v is a maximal valuation of lp, the following conditions hold.

1. v satisfies all invariants and guards of lp (lp can be visited infinitely often).
2. $v(x) = 0$, for every clock x that is reset in lp (once v is reached, no clock can ever decrease).
3. v reaches at least one upper bound in every invariant of lp (once v is reached, no clock can ever increase).

By way of example, Figure 5.2(i) shows a Zeno loop, $\langle b, c\rangle$ where a number of converged Zeno-timelocks may occur. For instance, $s = [2, v]$ ($v(x) = v(y) = 1$, $v(z) = 0$) is a converged Zeno-timelock that is reached if transition a is performed as soon as possible.

On the other hand, the converged Zeno-timelock $s' = [2, v']$ ($v'(x) = 1$, $v'(y) = 2$, $v'(z) = 0$) is reached if a was performed as late as possible. Note that, in this loop, the possible maximal valuations are represented by the set $\{\, v \mid v(x) = 1 \wedge 1 \leq v(y) \leq 2 \wedge v(z) = 0 \,\}$. In general, many different maximal valuations may be reachable in a loop; hence different converged Zeno-timelocks may be local to the same loop.

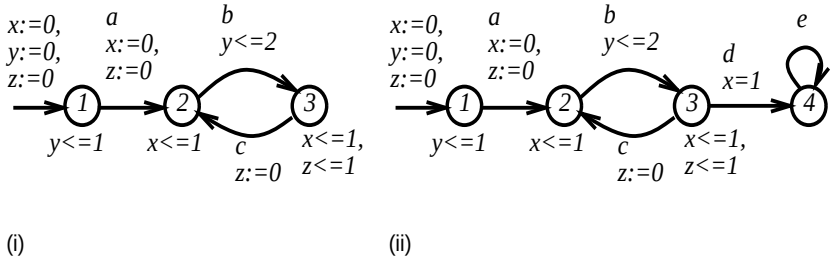


Figure 5.2: Zeno loops, Converged Zeno-timelocks and Maximal Valuations

[2]Note that, if v is a maximal valuation of lp, then $s' = [l', v]$ is reachable, for any $l' \in \mathit{Loc}(lp)$.

LEMMA **5.6.** *Let lp be a Zeno loop; $s = [l, v]$ a state reachable in lp; and v a maximal valuation of lp. Every finite run in $LocalRuns(s, lp)$ can be extended to an infinite run in $LocalRuns(s, lp)$, and every infinite run in $LocalRuns(s, lp)$ is a Zeno run.*

Proof. By definition of Zeno loop, there is a run in $LocalRuns(s, lp)$ that covers lp. Because v is maximal w.r.t. $LocalRuns(s, lp)$, no such local run can ever reach other valuation than v. Therefore, every finite run can be extended to a run that visits just the transitions of lp, infinitely often (i.e., an infinite local run). Again, because v is maximal, no location in lp can ever pass time (if reached from a transition in lp). Hence, every infinite run in $LocalRuns(s, lp)$ is a Zeno run. □

Zeno loops are responsible for Zeno runs that cover the loop; these runs are possible once a maximal valuation has been reached in the loop. Zeno-timelocks may occur only if maximal valuations are reachable, but the converse does not necessarily hold; for instance, maximal valuations may enable divergent runs, or may denote time-actionlocks (in neither case a Zeno-timelock would occur). However, if a maximal valuation does not represent a Zeno-timelock local to the loop, then it must enable some transition outside the loop. This motivates the definition of *escape transitions*.

Escape transitions. Let lp be a loop in A. We will say that a transition $l \xrightarrow{a,g,r} l'$ is an *escape transition* of lp if $l \in Loc(lp)$ and $l \xrightarrow{a,g,r} l' \notin Trans(lp)$. We use $Esc(lp)$ to denote the set of escape transitions from lp.

Figure 5.2(ii) shows that $\langle b, c\rangle$ is a Zeno loop, with maximal valuations in $\{v \mid v(x) = 1 \wedge 1 \leq v(y) \leq 2 \wedge v(z) = 0\}$. Transition d is an escape transition that is enabled by any maximal valuation of the loop. Zeno loops and maximal valuations do not necessarily determine the existence of Zeno-timelocks: any run visiting the loop $\langle b, c\rangle$ can be extended to a divergent run that visits e infinitely often.

If a maximal valuation is reached and escape transitions are not enabled at this point, a Zeno-timelock occurs. On the other hand, Zeno-timelocks may occur even if escape transitions are enabled by maximal valuations. Consider again Figure 5.1 (i), and the Zeno loop $\langle c, d\rangle$. Transition a is an escape transition from this loop, which is enabled by any of its maximal valuations (which satisfy $v(x) = 1$). Therefore, there is no Zeno-timelock that is local to the loop $\langle c, d\rangle$. However, a Zeno-timelock occurs that is local to $\langle a, b, d, c, d\rangle$.

THEOREM **5.7.** *Let lp be a loop in A. There exists a converged Zeno-timelock $s = [l, v]$, local to lp, if and only if lp is a Zeno loop, v is a maximal valuation of lp, and there are no escape transitions of lp that are enabled by v.*

Proof. ($\Rightarrow$) Assume that $s = [l, v]$ is a converged Zeno-timelock, local to lp. By definition, no run in $Runs(s)$ can ever reach a valuation other than v; there exists at least one run in $Runs(s)$ that covers lp; and every run in $Runs(s)$ visits only transitions of lp. Hence, lp is a Zeno loop; v is a maximal valuation of lp; and no escape transition can be enabled by v.

($\Leftarrow$) Assume that lp is a Zeno loop; v is a maximal valuation of lp; and there are no escape transitions from lp that are enabled by v. By Lemma 5.6, every finite local run (starting from s) can be extended to an infinite local run, and every such infinite run is a Zeno run. Because escape transitions are not enabled by v, every run starting from s must be local to lp, and v is the only valuation that can be reached. By definition, s is a converged Zeno-timelock, local to lp. Hence, the lemma holds. □

COROLLARY **5.8.** *A Zeno-timelock occurs in A if and only if there is a Zeno loop lp in A, s.t. some maximal valuation of lp is reachable that does not enable any escape transition of lp.*

Corollary 5.8 (and Theorem 5.7) represents a sufficient-and-necessary condition for the occurrence of Zeno-timelocks. However, this check depends on finding loops, maximal valuations, and escape transitions. In order to derive a practical method, a feasible way to obtain maximal valuations must be found. In what follows, we show that this can be done through reachability analysis.

5.2 Zeno-timelocks and Reachability

The following results relate maximal valuations and escape transitions with reachability formulae (incidentally, these formulae can be verified in Uppaal). As before, we assume a single automaton (A) that contains only completed actions, and where all invariants in loops are either *true*-invariants or right-closed invariants. In addition, in the definitions and results that follow, we do not necessarily assume simple loops, but we assume that the loops under consideration cannot be considered safe according to the static checks of Chapter 4 (i.e., the loops are not SNZ-loops and do not satisfy any invariant-based property in Lemma 4.7).

Let lp be a loop, and $Loc(lp) = \{l_1, \ldots, l_n\}$. Let $\prod = Clocks(l_1) \times \ldots \times Clocks(l_n)$. The formula $\Theta(x, l)$ (where $x \in Clocks(lp)$ and $l \in Loc(lp)$) holds whenever x has reached its smallest upper bound,[3] and such a bound is enforced by the invariant of l.

$$\Theta(x,l) \triangleq \begin{cases} x = c_{\min}(x, lp) & \text{if } x \leq c_{\min}(x, lp) \text{ occurs in } I(l) \\ \mathit{false} & \text{otherwise} \end{cases}$$

[3] $c_{\min}(x, lp)$ is the smallest upper bound for x in any invariant of lp (this was formally defined in Section 2.1.1).

Using $\Theta(x, l)$, the formula $sub(lp)$ denotes a valuation that has reached at least one smallest upper bound in every location of lp.[4]

$$sub(lp) \triangleq \bigvee_{(x_1,\ldots,x_n)\in\prod} \bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

Using $sub(lp)$, the formula $\alpha(lp)$ denotes a maximal valuation of lp,

$$\begin{aligned} \alpha(lp) \triangleq \quad & \bigwedge_{l\in Loc(lp)} I(l) \\ \wedge \quad & \bigwedge_{g\in Guards(lp)} g \\ \wedge \quad & \bigwedge_{y\in Resets(lp)} y = 0 \\ \wedge \quad & sub(lp) \end{aligned}$$

By way of example, we show below the values of $\Theta(X, L)$ and $\alpha(lp)$, for $lp = \langle a, b, c, d\rangle$ in Figure 5.3(i).

$\Theta(X, L) =$

clock X / location L	1	2	3	4
t	$\Theta(t, 1) = false$	*false*	*false*	$t = 0$
x	$x = 1$	*false*	*false*	*false*
y	$y = 2$	*false*	$y = 2$	*false*
z	*false*	$z = 2$	*false*	*false*
w	*false*	*false*	$w = 1$	*false*

$$\begin{aligned} \alpha(lp) = \quad & (x \le 1 \wedge y \le 2) \wedge (z \le 2 \wedge y \le 3) \ \wedge \ (y \le 2 \wedge w \le 1) \wedge (t \le 0) \\ \wedge \quad & (z > 1 \wedge y = 2) \\ \wedge \quad & (t = 0 \wedge w = 0) \\ \wedge \quad & (\ (x = 1 \wedge z = 2 \wedge y = 2 \wedge t = 0) \ \vee \ (x = 1 \wedge z = 2 \wedge w = 1 \wedge t = 0) \vee \\ & (y = 2 \wedge z = 2 \wedge y = 2 \wedge t = 0) \ \vee \ (y = 2 \wedge z = 2 \wedge w = 1 \wedge t = 0)\ ) \end{aligned}$$

Note that once a valuation that satisfies $\alpha(lp)$ is reached, and provided the execution does not leave the loop, every location can be visited (first conjunct), every transition can be performed (second conjunct), and clock values cannot decrease (third conjunct), or increase (fourth conjunct). Equivalently, $\alpha(lp)$ characterises the maximal valuations of lp. This is proved by the following two lemmas.

[4] We have assumed that lp does not satisfy any of the invariant-based properties in Lemma 4.7; in particular, it does not satisfy property (4). This ensures that in every location of lp, at least one clock must be compared against its smallest upper bound, and so $sub(lp)$ is well-defined.

LEMMA **5.9.** *Let lp be a loop, and v a valuation that satisfies sub(lp). For any location $l \in Loc(lp)$ s.t. $s = [l, v]$ is reachable, time cannot pass any further in l.*

Proof. Let $Loc(lp) = \{l_1, \ldots, l_n\}$ and $\prod = Clocks(l_1) \times \ldots \times Clocks(l_n)$. By definition, if v satisfies $sub(lp)$, then v satisfies

$$\bigvee_{(x_1,\ldots,x_n)\in\prod} \bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

In turn, v satisfies at least one disjunct,

$$\bigwedge_{i=1}^{n} \Theta(x_i, l_i)$$

for some $(x_1, \ldots, x_n) \in \prod$. By definition of $\Theta(x_i, l_i)$, this conjunction is satisfied if v has reached at least one upper bound in every invariant of lp. Hence, for any $l \in Loc(lp)$ s.t. $s = [l, v]$ is reachable, time cannot pass any further in l (by semantics of invariants). □

Let A be a timed automaton, l a location in A, and ϕ a state formula. We say that the formula $\texttt{EF}(A.l \wedge \phi)$ is satisfiable, if a state $s = [l, v]$ is reachable in some execution of A, where $v \models \phi$ (i.e, the standard interpretation of satisfiability for CTL formulae). We denote this, $s \models \texttt{EF}(A.l \wedge \phi)$.

LEMMA **5.10.** *Let lp be a loop in A, and $s = [l, v]$ be a reachable state (for some $l \in Loc(lp)$ and valuation v). Then, $s \models \texttt{EF}(A.l \wedge \alpha(lp))$ if and only if lp is a Zeno loop and v is a maximal valuation of lp.*

Proof. Assume that $s = [l, v] \models \texttt{EF}(A.l \wedge \alpha(lp))$. Then, by definition of $\alpha(lp)$, v satisfies every invariant and guard of lp, and $v(y) = 0$ for every y that is reset in lp. Therefore, once s is reached, lp can be visited infinitely often. Because the effect of resets in lp has been considered in v, clock values cannot decrease in local runs. Moreover, because $v \models sub(lp)$, clock values cannot increase in local runs (by Lemma 5.9). By definition, v is a maximal valuation of lp and lp is a Zeno loop.

Now, for the converse, assume that lp is a Zeno loop, and that v is a maximal valuation of lp. By definition, $s = [l, v]$ is reachable, $LocalRuns(s, lp) \cap CoveringRuns(s, lp) \neq \emptyset$, and v is maximal w.r.t. $LocalRuns(lp)$. Therefore, once s is reached, lp can be covered by local runs, and v remains constant in all such runs. Necessarily, (a) v must satisfy every invariant and guard of lp, (b) $v(y) = 0$ for every clock y that is reset in lp, and (c) clock values cannot increase in local runs, and so v reaches at least one upper bound in every invariant of lp (i.e., for every

$l' \in Loc(lp)$, $v(x) = c$, where $x \leq c$ is a term in $I(l')$). Thus, in order to prove that $v \models \alpha(lp)$, we just need to prove that this upper bound is the smallest upper bound for some clock in $I(l)$ (this will allow us to prove that $v \models sub(lp)$). We accomplish this as follows.

Suppose, by contradiction, that $v(x) = c$ and $x \leq c$ is a term in $I(l)$, where $c > c_{\min}(x, lp)$ (for some $l \in Loc(lp)$). There is, of course, another location $l' \in Loc(lp)$, where $x \leq c_{\min}(x, lp)$ is a term in $I(l')$. We have assumed that local runs can cover lp, without changing the valuation. However, l' cannot be entered once v has been reached, because $v(x) > c_{\min}(x, lp)$. Therefore, v must reach at least one smallest upper bound in every location of lp. By definition, $v \models sub(lp)$, and so we conclude that $s = [l, v] \models \mathtt{EF}(A.l \wedge \alpha(lp))$. □

According to Lemma 5.10, the satisfiability of $\alpha(lp)$ determines that lp is a Zeno loop, and that maximal valuations are reachable in lp. However, Zeno-timelocks cannot be guaranteed to occur in lp unless we also check that escape transitions are not enabled by any of these maximal valuations (Theorem 5.7).

Let $t \triangleq l \xrightarrow{a,g,r} l'$ denote an escape transition, and v a valuation. As we know, v enables t if $v \models g$ and $r(v) \models I(l')$ (i.e., the invariant in the target location holds from v, after resets have been performed). By definition, $r(v)(x) = v(x)$ if $x \notin r$, and $r(v)(x) = 0$ if $x \in r$.

In our timed automata model, invariants do not impose lower bounds, thus resets cannot invalidate invariants. We can safely claim that $r(v) \models I(l')$ if and only if v satisfies all conjuncts in $I(l')$ that do not refer to clocks in r.

Correspondingly, we define the formula $Target(l', r)$ to extract those conjuncts in $I(l')$ that do not refer to clocks in r. Then, with $Target(l', r)$ as an auxiliary formula, we define the formula $IsEnabled(g, r, l')$ to check whether a transition is enabled (where g, r and l' are the guard, reset set, and target location, respectively).

$$Target(l', r) \triangleq \{\, x \leq c \mid x \leq c \text{ occurs in } I(l') \text{ and } x \notin r \,\}$$

$$IsEnabled(g, r, l') \triangleq g \wedge \bigwedge_{conj \in Target(l', r)} conj$$

Let lp be a loop, and $Esc(lp) = \{l_1 \xrightarrow{a_1,g_1,r_1} l'_1, \ldots, l_n \xrightarrow{a_n,g_n,r_n} l'_n\}$ be the set of escape transitions of lp. We define $\beta(lp)$, which checks whether the current valuation enables some $t \in Esc(lp)$.

$$\beta(lp) \triangleq \textstyle\bigwedge_{i=1}^{n} \neg\, IsEnabled(g_i, r_i, l'_i)$$

Now, with $\alpha(lp)$ and $\beta(lp)$, we can use reachability analysis to characterise (precisely) the Zeno-timelocks local to lp. This is formalised in Theorem 5.11.

THEOREM **5.11.** *Let lp be a loop in A, and $s = [l, v]$ be a reachable state (for some $l \in Loc(lp)$ and valuation v). Then, $s \models \texttt{EF}(A.l \wedge \alpha(lp) \wedge \beta(lp))$ if and only if s is a converged Zeno-timelock local to lp.*

Proof. By definition, a valuation satisfies $\beta(lp)$ if and only if it does not enable any escape transition from lp. By Lemma 5.10, for any $l \in Loc(lp)$, $\texttt{EF}(A.l \wedge \alpha(lp))$ is satisfiable if and only if the state $s = [l, v]$ is reachable; lp is a Zeno loop; and v is a maximal valuation of lp. In turn, by Theorem 5.7, this can only happen if and only if s is a converged Zeno-timelock local to lp (by CTL semantics, $\texttt{EF}(A.l \wedge \alpha(lp) \wedge \beta(lp))$ holds if and only if $s = [l, v]$ is reachable, and $v \models \alpha(lp) \wedge \beta(lp)$). Hence, the lemma holds. □

COROLLARY **5.12.** *Let A be a timed automaton. A Zeno-timelock occurs in A is there is some loop lp s.t. $\texttt{EF}(A.l \wedge \alpha(lp) \wedge \beta(lp))$ is satisfiable for any $l \in Loc(lp)$.*

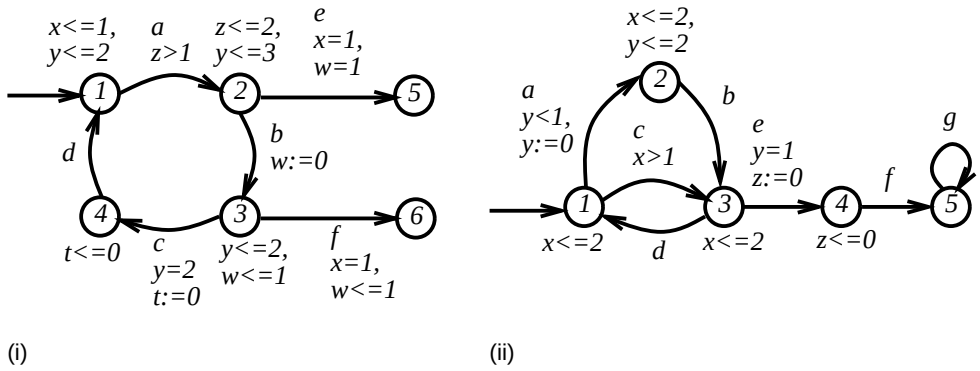


Figure 5.3: Zeno loops and Escape transitions

Consider the loop $lp = \langle c, d \rangle$ in Figure 5.3(ii). Formulae $Esc(lp)$, $\alpha(lp)$ and $\beta(lp)$ are shown below (expressions have been simplified).

$$\begin{aligned} Esc(lp) &= \{1 \xrightarrow{a,\ y<1,\ \{y\}} 2,\ 3 \xrightarrow{e,\ y=1,\ \{z\}} 4\} \\ \alpha(lp) &= x = 2 \\ \beta(lp) &= \neg\,(y < 1 \wedge x \leq 2) \wedge \neg\,(y = 1) \end{aligned}$$

Depending on the reachable valuations, $\langle c, d \rangle$ may or may not contain a local Zeno-timelock. For example, $\texttt{EF}(A.1 \wedge \alpha(lp) \wedge \beta(lp))$ is satisfiable if any state in $\{[1, v] \mid v(y) > 1\}$ is reachable. If so, a converged Zeno-timelock occurs, $s = [1, v],\ v(x) = 2,\ v(y) > 1$, with

$$s \models \texttt{EF}(A.1 \wedge x = 2 \wedge \neg\,(y < 1 \wedge x \leq 2) \wedge \neg\,(y = 1))$$

Note that, escape transitions a and e are not enabled ($v(y) > 1$).

On the other hand, $\langle c, d\rangle$ does not contain a (local) Zeno-timelock if any state in $\{[1, v] \mid v(x) > 1 \wedge v(y) = 0\}$ is reachable. When $v(x) = 2$ is reached, $v(y) < 1$ necessarily holds, and a is enabled by any maximal valuation of the loop. In addition, any reachable state in $\{[1, v] \mid v(x) > 1 \wedge v(y) = 0\}$ is a Zeno-timelock local to $\langle a, b, d, c, d\rangle$ (e is not enabled); and no state in $\{[1, v] \mid v(x) > 1\}$ is a Zeno-timelock local to $\langle a, b, d\rangle$ (c is enabled).

5.3 Zeno Runs and Reachability

Interestingly, a reachability formula can be derived that is sufficient-and-necessary to characterise the occurrence of Zeno runs. Note that, for a Zeno run to cover a particular loop, a valuation must be reached that (a) satisfies all invariants in the loop; (b) satisfies all the guards in the loop; and (c) assigns zero to every clock that is reset in the loop. Such a valuation can be represented by the formula $\gamma(lp)$,

$$\begin{aligned}\gamma(lp) \triangleq \quad & \bigwedge\nolimits_{l \in Loc(lp)} I(l) \\ \wedge \quad & \bigwedge\nolimits_{g \in Guards(lp)} g \\ \wedge \quad & \bigwedge\nolimits_{y \in Resets(lp)} y = 0\end{aligned}$$

Unlike Zeno-timelocks, Zeno runs are not related to maximal valuations or escape transitions. This is very convenient: we can derive a reachability-based property that is not restricted to right-closed invariants, and which can be analysed exclusively on simple loops (by Lemma 2.1, any Zeno run must cover some simple loop). This is formalised by Theorems 5.13 and 5.14.

THEOREM **5.13.** *Let lp be a loop in A. For any $l \in Loc(lp)$, the reachability formula*

$$\texttt{EF}(A.l \wedge \gamma(lp))$$

is satisfiable if and only if there exists a Zeno run that covers lp.

Proof. ($\Rightarrow$) Assume that $\texttt{EF}(A.l \wedge \gamma(lp))$ is satisfiable. Then, a state $s = [l, v]$ is reachable where $v \models \gamma(lp)$. By definition of $\gamma(lp)$, v satisfies every invariant and guard of lp, and $v(y) = 0$ for every y that is reset in lp. Therefore, once v is reached, there exists a run that covers lp without ever changing v. By definition, such a run is a Zeno run (with 0-delay) that covers lp.

($\Leftarrow$) Let ρ be a Zeno run that covers some loop lp. Note, that ρ must reach a state from which it only visits locations and transitions of lp (and does so infinitely often). Every location of lp can be visited infinitely often, and invariants only impose upper-bounds; hence, a valuation v can be reached that satisfies all invariants of lp.

Now, let Y be the set of clocks (possibly empty) that are bounded from below in any guard of lp. Necessarily, no clock in Y is reset in any transition of lp (otherwise, ρ would be a divergent

run). Therefore, and because every transition of lp can be visited infinitely often, we can choose v so that (a) it satisfies all lower bounds imposed by guards of lp, and (b) $v(x) = 0$ whenever x is reset or bounded from above in any guard of lp.

We proved that there exists a reachable valuation, v, which satisfies all invariants and guards of lp, and which assigns zero to every clock that is reset in lp. In other words, $\texttt{EF}(A.l \wedge \gamma(lp))$ is satisfiable for any $l \in Loc(lp)$. □

THEOREM **5.14.** *Let A be a timed automaton. A Zeno run occurs in A if and only if there is a simple loop lp that satisfies $\texttt{EF}(A.l \wedge \gamma(lp))$, for any $l \in Loc(lp)$.*

Proof. ($\Rightarrow$) By Lemma 2.1, if a Zeno run occurs, then this run covers some simple loop lp. By Theorem 5.13, $\texttt{EF}(A.l \wedge \gamma(lp))$ is satisfiable for any $l \in Loc(lp)$.

($\Leftarrow$) This is a consequence of Theorem 5.13. □

5.4 A Layered Strategy to Detect Zeno-timelocks

Here we show how our static checks (Chapter 4) and reachability-based properties can be efficiently combined to detect Zeno-timelocks. This strategy should be applied only after the strong non-Zenoness check (Section 4.1), and only if that check identifies unsafe loops.

We give an algorithm that receives a single automaton as input, A, and returns a set of loops where local Zeno-timelocks occur, $L_S \cup L_{NS}$ (see steps 3 and 4). The automaton A is assumed to contain only completed actions, and invariants in A are either *true*- or right-closed (in general, A denotes the product automaton of the network at hand).

The algorithm consists of a number of steps, where each step discards the loops in A that are guaranteed not to cause Zeno-timelocks. High-level operations such as set operations, loop detection and reachability analysis, are assumed to be primitive. The algorithm is as follows.

1. $L_0 = \mathit{SimpleLoops}(A) \setminus \{\ lp \mid lp \in \mathit{SimpleLoops}(A),\ \mathit{and\ lp\ is\ SNZ\ or\ satisfies\ Lemma\ 4.7}\ \}$
2. $L_1 = L_0 \setminus \{\ lp \mid lp \in L_0,\ \mathit{and}\ \texttt{EF}(A.l \wedge \alpha(lp))\ \mathit{is\ not\ satisfiable}\ \}$
3. $L_S = \{\ lp \mid lp \in L_1,\ \mathit{and}\ \texttt{EF}(A.l \wedge \alpha(lp) \wedge \beta(lp))\ \mathit{is\ satisfiable}\ \}$
4. $L_{NS} = \mathit{findZTs}(L_1 \setminus L_S, A)$

The first step applies Theorem 4.8 to identify simple loops that do not contain a Zeno-timelock. This results in a set of unsafe loops, L_0. The second step removes simple loops in L_0 where maximal valuations cannot be reached (i.e., those loops that are not Zeno loops). This leaves a set of simple loops L_1. In this way, we may reduce the number of loops to consider in

the following steps (which are more demanding). For example, we can avoid checking escape transitions in loops that cannot even reach a maximal valuation.

The third step identifies simple loops in L_1 that contain local Zeno-timelocks. These loops may or may not participate in non-local Zeno-timelocks; however, they have already been identified as a source of Zeno-timelocks, so we do not need to consider them any further.

The final step identifies non-simple loops that contain Zeno-timelocks. These non-simple loops result from simple loops in $L_1 \setminus L_S$. This is realised by *findZTs*(), shown in Figure 5.4.[5] This function systematically looks for non-simple loops (which contain only simple loops in $L_1 \setminus L_S$), and checks (using the reachability formula) for the occurrence of Zeno-timelocks.

The description of *findZTs*() assumes the following definitions. We define $lp \cup lp' = lp''$, where lp and lp' are s.t. $Loc(lp) \cap Loc(lp') \neq \emptyset$; and lp'' results from joining lp and lp' through their common locations. Equivalently, lp'' is s.t. $Loc(lp'') = Loc(lp) \cup Loc(lp')$, and $Trans(lp'') = Trans(lp) \cup Trans(lp')$. In addition, we used $lp \subseteq lp'$ to denote that $lp = lp'$ or $\exists\, lp''.\, lp \cup lp'' = lp'$.

Example. Figure 5.5 depicts a non-trivial automaton (which could represent a product automaton) that serves to illustrate how the algorithm works. We have identified six simple loops (in boldface),

$$SimpleLoops(A) = \{\ \texttt{lp1}, \texttt{lp2}, \texttt{lp3}, \texttt{lp4}, \texttt{lp5}, \texttt{lp6}\ \}$$

In the first step, `lp5` is the only loop that is recognised to be inherently safe (in fact, `lp5` is SNZ). This loop does not need to be considered any further, and so,

$$L_0 = \{\ \texttt{lp1}, \texttt{lp2}, \texttt{lp3}, \texttt{lp4}, \texttt{lp6}\ \}$$

The second step finds that, of all loops in L_0, `lp6` is the only one that is not a Zeno loop: location 9 is not even reachable (location 8 can only be reached if $v(x) > 1$), which in turn disables transition p). As a result of the second step, we get:

$$L_1 = \{\ \texttt{lp1}, \texttt{lp2}, \texttt{lp3}, \texttt{lp4}\ \}$$

The third step confirms that none of these loops spawns a local Zeno-timelock: for each of these loops, an escape transition is always enabled (once a maximal valuation has been reached). We get, then, $L_S = \emptyset$. Now, it is possible that Zeno-timelocks occur in non-simple loops. The last

[5] Parameters are passed by-value.

```
Function findZTs(L, A)
begin
    Z ← ∅;
    while L ≠ ∅ do
      Choose lp ∈ L;
      L ← L \ {lp};
      Z ← Z ∪ getZTsFrom(lp, L, A, Z);
    end
    return Z;
end.

Function getZTsFrom(lp, L, A,ZTLoops)
begin
    Aux ← ∅;
    while ∃ lp′ ∈ L. Loc(lp) ∩ Loc(lp′) ≠ ∅ do
      L ← L \ {lp′};
      lp″ ← lp ∪ lp′;
      if ∄ vlp ∈ ZTLoops. lp″ ⊆ vlp then
        Choose l ∈ Loc(lp″);
        if EF(A.l ∧ α(lp″) ∧ β(lp″)) then
          Aux ← Aux ∪ {lp″};
        else
          Aux ← Aux ∪ getZTsFrom(lp″, L, A,ZTLoops);
        end
      end
    end
    return Aux;
end.
```

Figure 5.4: Detecting Zeno-timelocks in Non-simple Loops

step confirms that Zeno-timelocks occur in the following non-simple loops.

$$nslp_1 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp3} \text{ (if } a \text{ is performed)}$$
$$nslp_2 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp4} \text{ (if } c \text{ is performed)}$$

Actually, the non-simple loop $nslp_3 = \texttt{lp1} \cup \texttt{lp2} \cup \texttt{lp3} \cup \texttt{lp4}$ also contains a Zeno-timelock (if b is performed). However, the algorithm does not return this combination, because it includes non-simple loops that are known to contain Zeno-timelocks. This can be confirmed by inspecting the behaviour of function *getZTsFrom*(), which looks for bigger combinations (in terms of number of simple loops) only if the current one does not contain a Zeno-timelock. In addition, combinations such as $\texttt{lp2} \cup \texttt{lp3}$ or $\texttt{lp2} \cup \texttt{lp4}$ are not verified, as they are part of non-simple loops that contain Zeno-timelocks (we use the variable *ZTLoops* in *getZTsFrom*(), to keep track of visited loops).

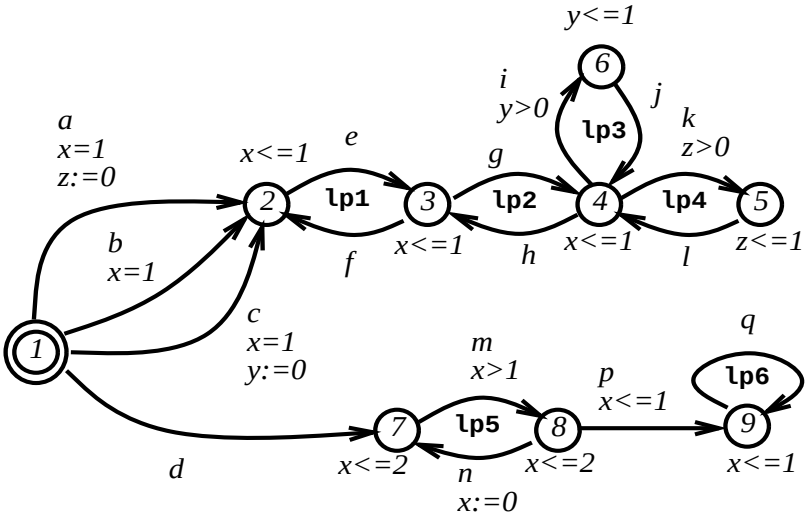


Figure 5.5: Detecting Zeno-timelocks

Discussion: Complexity. The detection of loops and reachability analysis may bring exponential complexity in worst-case scenarios;[6] however, our algorithm works on simple loops as much as possible and attempts to reduce, at every step, the set of loops to check.

Note, as well, that detection of non-simple loops does not occur as such, but rather non-simple loops are "constructed" on demand in the last step of the algorithm, from a reduced set of loops. In addition, model-checkers can usually perform reachability analysis very efficiently, even for complex specifications. Moreover, we would expect that for many timelock-free specifications, the static analysis performed in step 1 (i.e., checking for strong non-Zenoness and invariant-based properties) will suffice, and reachability analysis will not be required.

5.5 A Tool to Detect Zeno-timelocks

We implemented a tool that performs our proposed static and semantic checks over a subset of Uppaal specifications. The tool is implemented in Java, and accepts the XML format of Uppaal networks (thus, the user can benefit from Uppaal's GUI to construct the model). The tool is able to analyse the subset of Uppaal specifications that corresponds to our timed automata model augmented with urgent and committed locations (see Section 2.2.3). We have implemented Tarjan's algorithm [152] to partition graphs in strongly connected components, and the cycle enumeration algorithm of Szwarcfiter and Lauer [151] for loop detection.

[6]The size of the product automaton can be exponential in the size of the network's components (i.e., number of locations and transitions). The number of loops in a given automaton can be exponential in the size of the automaton [89]. The worst-case complexity of reachability analysis is linear in the size of the automaton, and exponential in the number of clocks and the size of the constants in clock constraints [8].

Initially, the tool performs a strong non-Zenoness check over the input network (Theorem 4.2 in Section 4.1). If this compositional analysis is inconclusive, the tool constructs the product automaton and performs the analysis described in Section 5.4.

Every simple loop in the product automaton is then checked for strong non-Zenoness and invariant-based properties (Lemma 4.7 and Theorem 4.8 in Section 4.2). For those loops that are found unsafe according to these properties, the tool computes and checks (interfacing with Uppaal's verifier) the reachability-based property of Theorem 5.11 (Section 5.1).[7]

Finally, if after the previous steps the analysis is inconclusive for some loops, the tool constructs and checks non-simple loops. Non-simple loops are derived only from those simple loops where no conclusive information could be obtained (Section 5.4).

We conclude this section with some remarks regarding urgent and committed locations. Note that, locations are not taken into account in the check of strong non-Zenoness (Theorem 4.2), because this property relies only on the syntax of guards and reset sets (Section 4.1). Therefore, urgent and committed locations are irrelevant to this check, and the syntactic transformation of Section 2.2.3 is not necessary. On the other hand, the transformation is necessary to check invariant-based and reachability-based properties.

5.6 Case Study: Timelocks in a CSMA/CD Protocol

The CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol controls the transmission of data between stations sharing a common medium, and is characteristic of Ethernet networks (LANs). The following description mainly follows [149].

A station wishing to transmit a frame first listens to the medium to determine if another transmission is in progress. If the medium is idle, the station begins to transmit; otherwise the station continues to listen until the medium is idle, then it begins to transmit immediately. It may happen that two or more stations begin to transmit at about the same time. If this happens, there will be a collision and the data from both transmissions will be garbled and not received successfully. If such a collision is detected during transmission, the station transmits a brief jamming signal (to ensure that all stations know that there has been a collision) and then it ceases transmission. After transmitting the jamming signal, the station waits a random amount of time and then attempts to retransmit the frame.

Collisions can only occur when more than one station start transmitting within a short time (the propagation delay). If a station attempts to transmit a frame, and there are no collisions during the time it takes for the leading edge of the packet to propagate to the farthest station, then there will be no collision because all other stations are aware of the transmission (i.e., the

[7] We plan to include the check for Zeno runs (Section 5.3) in the next version of the tool.

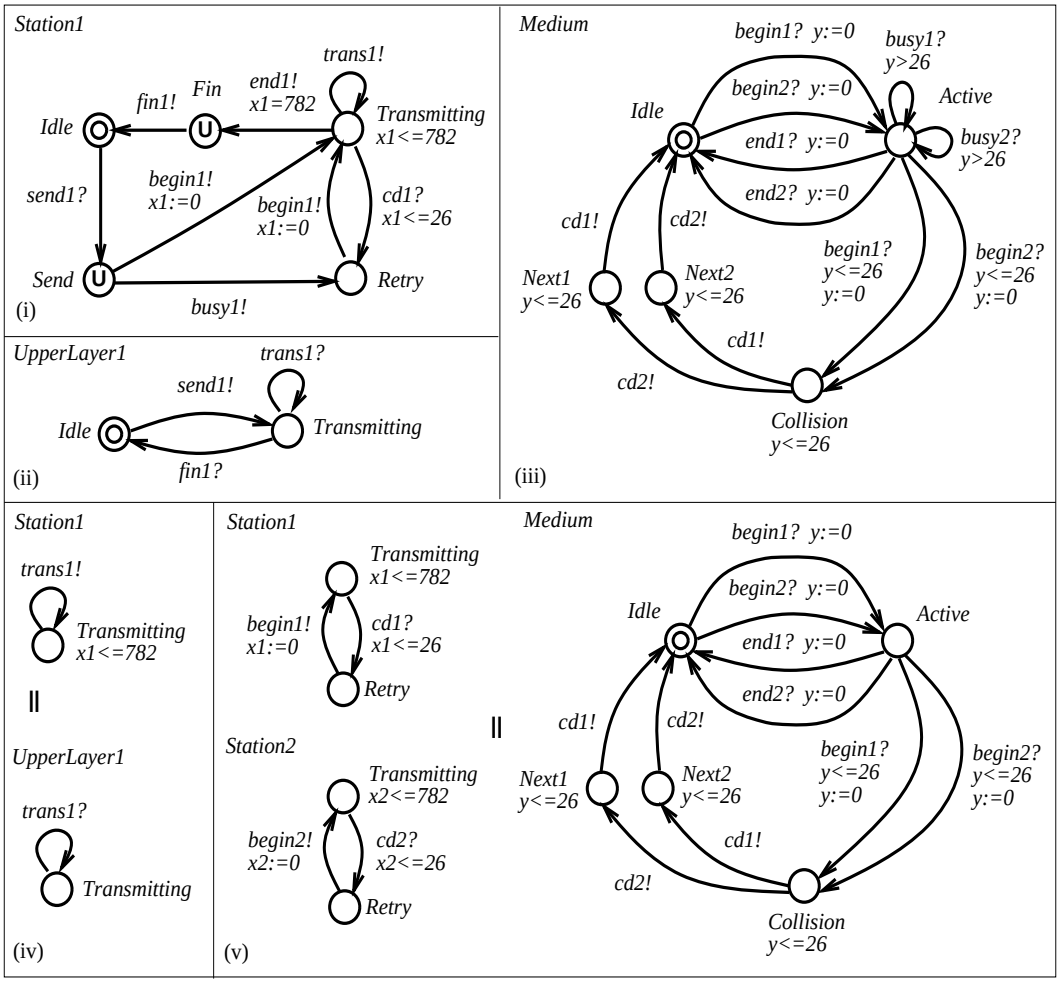


Figure 5.6: An Uppaal Model For The CSMA/CD Protocol

medium will be found busy). Note, as well, that the time needed to detect a collision is no greater than twice the propagation delay.

Figures 5.6(i) and (iii) show part of an Uppaal specification for the protocol. We have considered only two stations, *Station1* (Figure 5.6(i)) and *Station2* (similar to (i) modulo renaming). The main role of *Station1* is to model the transmission of frames, and the retransmission of frames in the case of a collision. The automaton *Medium* (Figure 5.6 (iii)) models the state of the medium; this includes collision detection and broadcasting of the jamming signal. Both *Station1* and *Medium* have temporal constraints derived from either the end-to-end propagation delay (26 μs.), or the frame transmission-time (782 μs.). We have included the automaton *UpperLayer1* (Figure 5.6 (ii)) to model a client layer that uses the protocol service in the station (*UpperLayer2* is similar). It simply provides frames to the protocol layer, and acknowledges ongoing transmission and successful termination.

Automaton *Station1* starts in *Idle*, waiting for *UpperLayer1* to send a new frame (*send1?*). If this happens, *Station1* moves to *Send*, which is an urgent location. The station may find that either the medium is idle, and so the transmission of the new frame can start immediately (*begin1!*), or that the medium is busy, and so the station has to wait (*busy1!*). Location *Transmitting* denotes that a transmission has started. Transmission of a complete frame takes 782 μs. Immediately after ending a transmission (location *Fin*), a signal *fin1!* is sent to the upper layer. Ongoing transmission is also signaled to the upper layer (*trans1!*).

A collision with another station may occur in *Transmitting*, in which case the jamming signal *cd1?* is detected. The guard $x1 \leq 26$ denotes that no collision can occur after 26 μs. have passed since a station started sending a frame. Location *Retry* denotes that a collision indeed occurred and that the station is waiting to attempt a retransmission (*begin1!*). The station remains in *Retry* if a retransmission attempt finds a busy medium (note that, *begin1?* is not enabled in such a situation, because $v(x1) > 26$).

The *Medium* starts in *Idle*, waiting for stations to begin their transmissions (*begin1?*/*begin2?*); then it moves to *Active* and clock y is reset. *Active* denotes that a station is currently using the medium. In *Active*, the value of y denotes the time elapsed since the station started its transmission. Transitions *busy1?*/*busy2?* denote that stations can already acknowledge that the medium is busy and thus, that new transmissions are not yet possible. The guard $y > 26$ in *busy1?*/*busy2?* denotes that a second station can only acknowledge a busy medium after 26 μs. have passed since the first station started its transmission (the propagation delay).

Collision denotes that a collision has happened, and that the jamming signal is about to reach the stations. The *Medium* moves from *Active* to *Collision* through *begin1?*/*begin2?* happening at $v(y) \leq 26$, i.e., a second station has started transmitting a frame before it could acknowledge that the medium is busy. In *Collision*, y denotes the time elapsed since a collision occurred; note that y is reset when the second transmission begins while *Medium* is in *Active* (to simplify matters, we have assumed that a collision occurs as soon as this second transmission begins). The sequences *cd1!-Next2-cd2!* and *cd2!-Next1-cd1!* model the jamming signal reaching *Station1* and *Station2*, in any order. Moreover, the invariants $y \leq 26$ in *Collision* and *Next1*/*Next2* indicate that the jamming signal reaches the stations not later than 26 μs. after the collision.

5.6.1 Checking Strong Non-Zenoness

Here we explain how the automata *UpperLayer1* and *UpperLayer2* disguise a time-actionlock in the specification, making it undetectable to Uppaal. Interestingly, this hidden time-actionlock results in a Zeno-timelock, which our tool helps to detect.

We begin by asserting deadlock-freedom using the formula `A[]not deadlock`, which Uppaal finds satisfiable. We then use our tool (Section 5.5) to discover that a number of unsafe loops,

which may cause Zeno-timelocks. These loops correspond to the interaction between *Station1* and *UpperLayer1* (Figure 5.6(iv)), between *Station2* and *UpperLayer2* (not shown), and between *Station1*, *Station2* and *Medium* (Figure 5.6(v)).

By way of example, we describe below the completed loops that result from synchronising the unsafe loops of Figure 5.6(v). We use $l_1 l_2 l_3$ to denote a location in the product automaton where l_1, l_2 and l_3 are respectively locations in *Station1*, *Medium* and *Station2*. We use *R*, *I*, *T*, *A*, *C*, *N1* and *N2* to denote locations *Retry*, *Idle*, *Transmitting*, *Active*, *Collision*, *Next1* and *Next2*, respectively. Locations in *UpperLayer1* and *UpperLayer2* are omitted (these have *true* invariants, so they are irrelevant to our analysis). Internal actions *begin1*, *begin2*, *cd1* and *cd2* result from synchronisation between the corresponding half actions.

$$lp_1 = \langle\ RIR, begin1, TAR, begin2, TCT, cd1, RN2T, cd2, RIR\ \rangle$$
$$lp_2 = \langle\ RIR, begin2, RAT, begin1, TCT, cd1, RN2T, cd2, RIR\ \rangle$$
$$lp_3 = \langle\ RIR, begin1, TAR, begin2, TCT, cd2, TN1R, cd1, RIR\ \rangle$$
$$lp_4 = \langle\ RIR, begin2, RAT, begin1, TCT, cd2, TN1R, cd2, RIR\ \rangle$$

These loops correspond to situations in which stations continue to retransmit their frames too soon, therefore colliding again after every attempt. These loops are not SNZ (note in Figure 5.6(v) that clocks are reset in the component loops, but not bounded from below), and Zeno runs occur that correspond to sequences of collisions and retransmissions. However, time can always pass in location *RIR* because invariants in *Retry* and *Idle* are *true*. This location is included in every loop lp_1 to lp_4, and so these loops are inherently safe (by Lemma 4.7).

Now we focus on the unsafe loop in *Station1* (Figure 5.6(iv)). A Zeno-timelock occurs in location *Transmitting* (Figure 5.6(i)) if *trans1!* is the only enabled transition when time stops. If this Zeno-timelock occurs, a deadlock occurs if *trans1!* cannot longer induce Zeno runs. Effectively, Uppaal detects a deadlock in the resulting specification, after we modify *trans1!* by adding an arbitrary lower bound (we added a new local clock *z1* to *Station1*, guarded *trans1!* with $z1 = 1$, and added the reset $z1 := 0$; we did the same for *Station2*).

This deadlock is caused by an error in the guard of *cd1?* in *Station1*, $v(x1) \leq 26$ (a similar error is present in *Station2*).[8] This guard denotes that collisions cannot occur after 26 μs. have passed since *Station1* started transmitting a frame; however, 26 μs. happens to be too small an upper bound for collision detection, as the following scenario illustrates.

1. *Station1* starts transmitting a frame and moves to location *Transmitting*; *Medium* moves to *Active*.

2. *Station2* starts transmitting a frame just before 26 μs have passed since *Station1* started

[8] Yovine highlighted the same error in [169].

transmitting. At this point, *Station1* remains in *Transmitting*, *Station2* has changed to *Transmitting*, and *Medium* has changed to *Collision*. Note, also, that $v(x1) \leq 26$ and $v(x2) = v(y) = 0$.

3. Based on the previous observations, the values of y and $x1$ are such that $v(y) \leq 26$ and $v(x1) \leq 52$, while the automaton remains in *Collision*. In particular, if $26 < v(x1) \leq 52$, *cd1!* in *Collision* will not be able to synchronise with *cd1?* in *Station1*. Transition *cd2!* can still be performed to reach *Next1*, but *cd1!* cannot be performed from *Next1* either. Thus, no action is enabled while *Medium* remains in *Next1*. Furthermore, the invariant $y \leq 26$ in *Next1* also prevents time from diverging, causing a time-actionlock when $v(y) = 26$.

This time-actionlock shows that the guard $x1 \leq 26$ in *cd1?* (in *Station1*) should be modified to account for a bigger delay, i.e., it should be $x1 \leq 52$. After a transmission has started, the jamming signal could be detected up to 52 μs later, that is, twice the propagation delay [149].

This (hidden) time-actionlock resulted in a Zeno-timelock in the original specification (i.e., before *trans1!* was modified). When *Medium* is in *Collision* and $v(y) = 26$, and *Station1* and *Station2* are in *Transmitting*, *trans1!* (*trans2!*) will be performed infinitely often while time is prevented from passing (synchronisation with *UpperLayer1*/*UpperLayer2* is always possible).

If we correct the specifications of *Station1* and *Station2* ($x1 \leq 52$ in *cd1?* and $x2 \leq 52$ in *cd2?*), we can verify that the specification is free from actionlocks (and thus free from time-actionlocks). In turn, because now a time-actionlock no longer arises, the loop *trans1!* (Figure 5.6(i)) does not cause a Zeno-timelock. Time will not be prevented from passing in *Next1*/*Next2* (Figure 5.6(iii)), thus the system will evolve normally. After a collision occurs, the stations move from *Transmitting* to *Retry*; i.e., *trans1!* in *Transmitting* is no longer enabled.

Summary. We started our analysis of the CSMA/CD protocol by asserting deadlock-freedom, using the formula `A[]not deadlock` in Uppaal. The specification was found to be deadlock-free, and thus, also free from time-actionlocks. We then checked strong non-Zenoness (compositionally) using our tool. The tool found a number of potentially unsafe loops (pairs of synchronising loops where no loop was SNZ), which we examined by hand.

We found that Zeno-timelocks occurred in some of these loops. The Zeno loops ($\langle$*trans1!*$\rangle$ in *Station1* and $\langle$*trans2!*$\rangle$ in *Station2*) witnessed Zeno runs due to under-specification; we thus added lower bounds to these loops and found that Zeno-timelocks were hiding time-actionlocks (once the loops were modified, time-actionlocks could be detected in Uppaal as deadlocks, using the formula `A[]not deadlock`). Finally, the system was corrected to remove the time-actionlocks (this also prevented Zeno-timelocks).

5.6.2 Checking Invariant- and Reachability-based Properties

In the previous section we used our tool to check for strong non-Zenoness in the network. Strong non-Zenoness was found unsatisfiable, and the tool revealed a number of potentially unsafe loops. We examined these unsafe loops by hand, and concluded that some of them were indeed the cause of Zeno-timelocks. This section shows that our tool can discover such Zeno-timelocks automatically, at the expense of a more demanding analysis.

Specifically, the tool constructs the product automaton, and applies strong non-Zenoness, invariant-based properties (see Lemma 4.7 and Theorem 4.8), and reachability-based properties (see Theorem 5.11 and Corollary 5.12).

Figure 5.7 depicts the product automaton for the network of Figure 5.6. Location vectors are given in the form $l_1l_2l_3l_4l_5$, where $l_1 \in$ *UpperLayer1*, $l_2 \in$ *Station1*, $l_3 \in$ *Medium*, $l_4 \in$ *Station2* and $l_5 \in$ *UpperLayer2*. Here, *I*, *S*, *T*, *A*, *C*, *R*, *F*, *N1* and *N2* denote, respectively, *Idle*, *Send*, *Transmitting*, *Active*, *Collision*, *Retry*, *Fin*, *Next1* and *Next2*.

Recall the analysis of the previous section: Zeno-timelocks (if any occur) may only be caused by composition of the following half loops.

1. $\langle$*trans1*!$\rangle$ in *Transmitting* (*Station1*) || $\langle$*trans1*?$\rangle$ in *Transmitting* (*UpperLayer1*)
2. $\langle$*trans2*!$\rangle$ in *Transmitting* (*Station2*) || $\langle$*trans2*?$\rangle$ in *Transmitting* (*UpperLayer2*)

These half loops, in turn, result in the following set of completed loops in the product automaton (Figure 5.7).

1. $\langle$*trans1*$\rangle$ in *TTAII*, *TTAST*, *TTART*, *TTCTT*, *TTN1RT* and *TTAFT*.
2. $\langle$*trans2*$\rangle$ in *IIATT*, *TSATT*, *TRATT*, *TTCTT*, *TRN2TT* and *TFATT*.

Effectively, our tool constructed the product automaton and found that these are the only loops that do not satisfy Lemma 4.7 (i.e., none of them satisfies any invariant-based property). Next, the tool derived and checked (via Uppaal) the reachability-based property for Zeno-timelocks (Theorem 5.11). The α- and β-formulae are shown in Tables 5.1 and 5.2. Table 5.3 summarises the analysis realised in Uppaal (formulae are given in Uppaal syntax; `P` is the name assigned to the product automaton; only loops involving *trans1* are shown). Satisfiability results show that all loops are Zeno loops, i.e., they satisfy $\texttt{EF}(\texttt{P}.l \wedge \alpha(lp))$. However, only $\langle$*trans1*$\rangle$ in *TTN1RT* and $\langle$*trans2*$\rangle$ in *TRN2TT* cause Zeno-timelocks, because these are the only loops that satisfy $\texttt{EF}(\texttt{P}.l \wedge \alpha(lp) \wedge \beta(lp))$

This confirms the scenarios and static analysis of the previous section. Finally, by applying the last step of the algorithm of Section 5.4, the tool found that none of these simple loops can be combined into non-simple loops (i.e., non-simple loops in the product are safe). In addition,

trans1 in l	$\alpha(lp)$
TTAII	`P.x1==782`
TTAST	`P.x1==782 or P.u2==0`
TTART	`P.x1==782`
TTCTT	`P.x1==782 or P.x2==782 or P.y==26`
TTN1RT	`P.x1==782 or P.y==26`
TTAFT	`P.x1==782 or P.u2==0`

Table 5.1: α-formulae

trans1 in l	$\beta(lp)$
TTAII	`not P.x1==782 and not P.x1<=782`

TTAST	`not P.x1==782 and`
	`not (P.y<=26 and P.x1<=782) and`
	`not (P.y>26 and P.x1<=782)`

TTART	`not P.x1==782 and not P.y<=26`
TTCTT	`not (P.x1<=26 and P.y<=26 and P.x2<=782) and` `not (P.x2<=26 and P.y<=26 and P.x1<=782)`
TTN1RT	`not P.x1<=26`
TTAFT	`not (P.x1==782 and P.u2<=0 and P.u1<=0) and` `not (P.x1<=782)`

Table 5.2: β-formulae

we used the tool to check the corrected CSMA/CD protocol, and confirmed that is free from deadlocks and Zeno-timelocks (and thus, timelock-free).

A Note on Uppaal's Check: The Liveness Property λ_U. We also checked the property λ_U (Section 3.5.1), but the results were inconclusive. There are several loops in the specification that induce Zeno runs (e.g., lp_1 of Section 5.6.1), which invalidate λ_U. However, these loops do not cause Zeno-timelocks. In fact, the CSMA/CD specification contains Zeno runs even after we remove the timelocks. Thus, if verification is not affected by Zeno runs (e.g., for safety properties), checking λ_U in Uppaal cannot distinguish safe from unsafe specifications.

5.7 Conclusions

We presented semantic checks to guarantee the absence of Zeno-timelocks and Zeno-runs in a timed automaton. As we have mentioned before, this is equivalent to timelock-freedom in specifications that are deadlock-free.

The semantic checks are based on reachability formulae that are derived from the syntax of unsafe loops (those loops in the automaton that do not satisfy strong non-Zenoness or

`trans1` *in* l	`EF`$(P.l \wedge \alpha(lp))$	`EF`$(P.l \wedge \alpha(lp) \wedge \beta(lp))$
TTAII	`SAT.`	`NOT SAT.`
TTAST	`SAT.`	`NOT SAT.`
TTART	`SAT.`	`NOT SAT.`
TTCTT	`SAT.`	`NOT SAT.`
TTN1RT	`SAT.`	`SAT.`
TTAFT	`SAT.`	`NOT SAT.`

Table 5.3: α+β-analysis

the invariant-based properties of Chapter 4). We proved that these checks are sufficient-and-necessary, provided the invariants in the loops of the automaton satisfy a syntactic constraint (invariants must be *true* or right-closed). Thus, semantic checks for Zeno-timelocks can be conclusive in specifications where strong non-Zenoness, invariant-based properties, and the property λ_U in Uppaal (Section 3.5.1) do not hold.

We exploit the complementary nature of static and semantic checks in a layered detection strategy: static checks are applied first, and semantic checks are applied as required (i.e., if the previous analysis has been inconclusive). We implemented a tool that realises this strategy, which performs the static and semantic checks over Uppaal specifications. Furthermore, we have used the tool to facilitate the analysis of timelocks in a specification of the CSMA/CD protocol (a well-known benchmark for real-time formalisms).

Compared with timelock detection in model-checkers (Uppaal, Kronos and Red), which is based on liveness properties, our semantic checks are based on simple reachability analysis, and thus can be implemented in all model-checkers. Moreover, our checks for Zeno-timelocks are insensitive to Zeno runs, which make Uppaal's check inconclusive (Section 5.6).

Currently, our theory do not consider the effect of data variables in the occurrence of Zeno runs and Zeno-timelocks. Thus, model-checkers are able to guarantee timelock-freedom for a bigger class of specifications (e.g., Uppaal's specification language permits integer and array variables, and non-zero resets). In addition, Uppaal's verification is on-the-fly, and so this may be more efficient than our semantic checks if timelocks are found before the whole reachability graph is explored (we need to construct the product automaton).

We plan to extend our static and semantic checks, in order to consider data variables, non-zero resets, urgent channels and other modelling facilities available in Uppaal. Our initial experience with the tool is encouraging, and more case studies will contribute to improve the implementation.

This chapter concludes Part II, where we introduced the theory of Timed Automata and explained some its limitations (Chapter 2), and where we offered solutions to the problem of timelocks and Zeno runs. In Part III, we will use WS1S and MONA (Chapter 6) to explore

solutions to other limitations of timed automata and model-checkers, regarding expressiveness issues in branching-time requirements languages (Chapter 7) and data support (Chapter 8). The reader will find timelocks revisited in Chapter 8, where a deductive framework is presented that rules out time-actionlocks by construction.

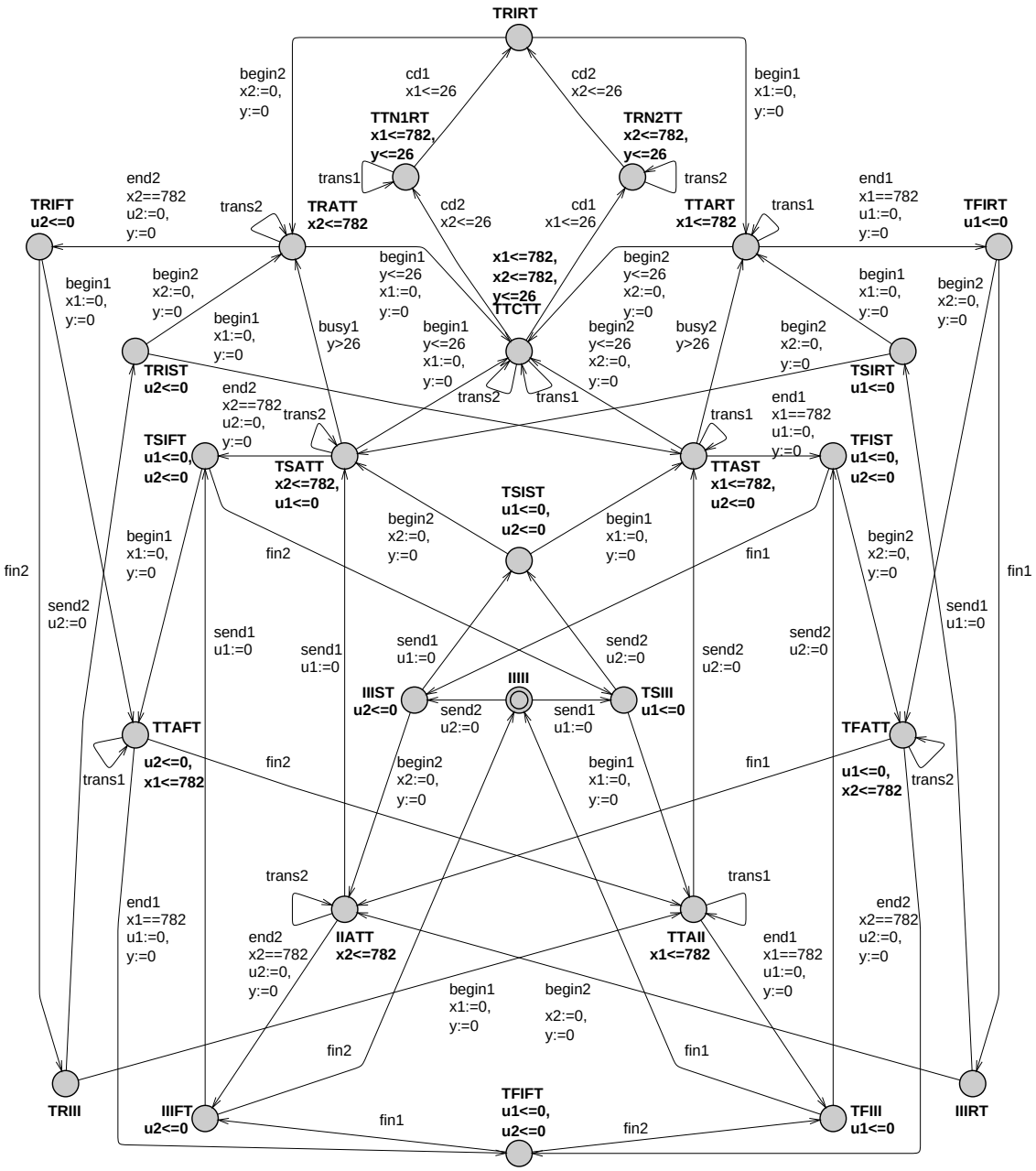


Figure 5.7: CSMA/CD (Product Automaton)

Part III

WS1S and MONA

Chapter 6

WS1S and MONA

Abstract. The Weak Monadic Second-order Theory of One Successor (WS1S) is a decidable logic with a semantics tied to arithmetic, which has the expressive power of regular languages on finite words. MONA is an efficient implementation of a decision procedure for WS1S. Despite the worst-case non-elementary complexity of WS1S, MONA has been successfully applied to many nontrivial problems, e.g., hardware verification, controller synthesis and theorem proving.

This is the background chapter for Part III, where we explore the use of WS1S and MONA in real-time settings. We are motivated by limitations in timed automata and model-checkers (Chapter 2). Chapter 7 will show that MONA can be used as an efficient decision procedure for PITL, an interval temporal logic where certain requirements are easier to express than in Uppaal and other model-checkers. Chapter 8 will introduce Discrete Timed Automata, a deductive framework where WS1S is used as the assertion language, and MONA serves as tool support for invariance proofs. Discrete Timed Automata support unbounded data domains, and specifications are time reactive by construction.

Organization. We give an overview of WS1S and MONA in Section 6.1. We present the syntax and semantics of MONA specifications (as required in this thesis) in Section 6.2. In Section 6.3 we comment on the decision procedure implemented by MONA. We conclude with a summary in Section 6.4.

6.1 Introduction

The Weak Monadic Second-order Theory of One Successor (WS1S) [54, 66, 155] is a decidable logic with a semantics tied to arithmetic. WS1S has the expressive power of regular languages on finite words, which makes decidability non-elementary [114]. Despite being a non-elementary

logic, the tool MONA[1] [96] demonstrated that WS1S could be used in practice. MONA is an efficient implementation of a decision procedure for WS1S, which has been successfully applied in a number of nontrivial problems, from hardware, protocol and software verification [12, 148, 65, 117], to computational linguistics, controller synthesis and theorem proving [118, 142, 131, 11]. MONA is also part of many verification tools [137, 130, 13, 134].

A specification in MONA represents a WS1S formula; MONA translates this formula into an equivalent deterministic finite automaton: the models (counterexamples) of the formula are represented by paths in the automaton leading to accepting (rejecting) states. MONA returns the shortest model (or counterexample) in terms of the free variables of the formula.

By way of example, Figures 6.1 and 6.2 show a MONA specification that "calculates" the multiples of 5 from 0 to 20, and the result of the analysis. MONA finds that the WS1S formula is satisfiable, where `M5={0,5,10,15,20}` is the shortest model, and `M5={}` is the shortest counterexample. The output also shows the string interpretation for `M5` (we will discuss this interpretation in Section 6.3).

```
% declaration section

const FROM = 0;
const TO = 20;
var2 M5;

pred cons(var2 Set, var1 s1,s2) =
         s1 in Set & s2 in Set & s1<s2 &
         ~ ex1 s':s' in Set & s1<s' & s'<s2;

% formula section

FROM = min M5 & TO = max M5 & all1 m1,m2:cons(M5,m1,m2) => m1+5 = m2;
```

Figure 6.1: A MONA Specification

The automata-based decision procedure for WS1S dates back to [54, 66]; it is in the efficient implementation of the translation to automata where MONA stands out. In MONA, the automaton's transition relation is compactly encoded using Binary Decision Diagrams (BDDs) [53, 52]; this, and other optimisations (e.g., the identification of isomorphisms in the source formula) have proved to be the ingredients of MONA's success [97].

MONA provides the user with different information about the translation process, such as the size of the automata and BDDs, the time spent in the process, optimisations performed and intermediate results (e.g., partial automata for subformulae). Additionally, MONA offers libraries to directly manipulate automata and BDDs without interfacing with a WS1S formula.

[1] `http://www.brics.dk/mona`

```
ANALYSIS
A counter-example of least length (0) is:
M5        X
M5 = {}

A satisfying example of least length (21) is:
M5        X 10000100000100001000001
M5 = {0,5,10,15,20}

Total time:  00:00:00.02
```

Figure 6.2: Result of Analysis for the MONA specification of Figure 6.1

6.2 MONA – Syntax and Semantics

WS1S is a logic interpreted over natural numbers ($\mathbb{N}$), and finite sets of natural numbers ($\mathcal{F}(\mathbb{N})$). The syntax of WS1S can be characterised by the following BNF.

$$\phi ::= X(x) \mid succ(x, y) \mid \neg\,\phi \mid \phi \ \wedge\ \psi \mid \exists\, x.\, \phi \mid \exists\, X.\ \phi$$

where ϕ and ψ are WS1S formulae, x and y are first-order variables, and X is a second-order variable. Given I an interpretation that maps first-order variables to natural numbers ($I(x) \in \mathbb{N}$), and second-order variables to finite sets of natural numbers ($I(X) \in \mathcal{F}(\mathbb{N})$), the semantics of WS1S is defined as follows.

$$\begin{array}{lll}
I \models X(x) & \text{iff} & I(x) \in I(X) \\
I \models succ(x, y) & \text{iff} & I(x) + 1 = I(y) \\
I \models \neg\,\phi & \text{iff} & I \not\models \phi \\
I \models \phi \ \wedge\ \psi & \text{iff} & I \models \phi \text{ and } I \models \psi \\
I \models \exists\, x.\, \phi & \text{iff} & I[n/x] \models \phi \text{ for some } n \in \mathbb{N} \\
I \models \exists\, X.\, \phi & \text{iff} & I[N/X] \models \phi \text{ for some } N \in \mathcal{F}(\mathbb{N})
\end{array}$$

where $I[u/v]$ denotes substitution, i.e., $I[u/v](v) = u$ and $I[u/v](x) = I(x)$ for all $x \neq v$.

The language of MONA specifications augments WS1S with syntactic sugar.[2] The following BNF describes the fragment of MONA specifications which are relevant to this thesis. A complete syntax, and other features available in MONA, can be found in [96].

[2] Actually, MONA also implements a decision procedure for WS2S [155], a generalisation of WS1S that is interpreted over finite binary trees. In this thesis we are concerned just with WS1S, so the subset of MONA specifications that we will use is expressively equivalent to WS1S.

program ::= [*declaration* `;`]$^{?}$ ϕ

declaration ::= `const` *constname* `=` k |
[`var0`|`var1`|`var2`] [*varwhere*]$^{\oplus}$ |
`pred` *predname* [`(` [*par*]$^{\odot}$ `)`]$^{?}$ `=` ϕ

ϕ ::= `true` | `false` | b | `(` ϕ `)` | $\sim \phi$ | ϕ_1 [`&`|`|`|`=>`|`<=>`] ϕ_2 |
[`ex1`|`ex2`|`all1`|`all2`] [*varwhere*]$^{\oplus}$`:`ϕ |
t_1 [`~=`|`=`|`<`|`>`|`<=`|`>=`] t_2 |
T `=` T | T_1 `sub` T_2 | t [`in`|`notin`] T | `empty (`T`)` |
predname `(` [t|T|ϕ]$^{\odot}$ `)`

t ::= p | `(` t `)` | k | t [`+`|`-`] k | [`max`|`min`] T

T ::= P | `(` T `)` | `{` [t]$^{\odot}$ `}` | `empty` | T [`+`|`-`] k | T_1 [`union`|`inter`|`\`] T_2

par ::= [`var0`|`var1`|`var2`] [*varwhere*]$^{\oplus}$
varwhere ::= v [`where` ϕ]$^{?}$

where ϕ, ϕ_1 and ϕ_2 are MONA formulae; *constname* is a constant identifier; k is a constant integer expression; v is a variable identifier (Boolean, first-order or second-order variable); b, p and P are identifiers of Boolean, first-order and second-order variables (resp.); *predname* is a predicate identifier; t, t_1 and t_2 are first-order terms; and T, T_1 and T_2 are second-order terms. The meta-operator $[X|Y]$ denotes either X or Y; $[X]^{+}$ denotes one or more occurrences of X; $[X]^{?}$ denotes an optional occurrence of X; $[X]^{\oplus}$ denotes one or more occurrences of X separated by commas; and $[X]^{\odot}$ denotes zero or more occurrences of X separated by commas.

A MONA specification consists of a declaration section, in which free variables are introduced, and a formula section, where the WS1S formula itself is defined (naturally, the declaration section may be omitted if no free variables, constants or predicates, occur in the formula). Boolean (`var0`), first-order (`var1`) and second-order (`var2`) variables can be declared, as well as integer constants (`const`) and predicates (`pred`), which act as formula "templates".[3]

Logic connectives include negation (`~`), conjunction (`&`), disjunction (`|`) and implication (`=>`). First-order terms can be combined using relational operators (e.g. `t1>=t2`), addition of constant values (`t+`k) and quantification (`ex1 t:`φ, `all1 t:`φ).

[3]Other data types can be derived as well, such as enumerations, records and bounded arrays [148, 29].

Second-order terms can be subject to operators that retrieve the minimum and maximum element in the set (`min T`, `max T`), test for membership (`t in T`), set inclusion (`T1 sub T2`), quantification (`ex2 T:`φ, `all2 T:`φ) and other typical operations such as intersection (`inter`), difference (\) and union (`union`).

Restrictions can also be added to variables, e.g., the expression `v where v in {0,1}` constrains the value of the variable `v` to be in the set $\{0, 1\}$ (this restriction is taken into account when checking the satisfiability of formulae where `v` occurs). Note that, addition is possible but is restricted to constants; on the other hand, general addition, multiplication and set complementation are not possible in WS1S (these will make the logic undecidable).

6.3 From MONA Specifications To Finite Automata

MONA implements a decision procedure for WS1S based on a translation from WS1S formulae to deterministic finite automata [54, 66]. This can be explained in terms of the following simplified WS1S syntax, which does not include first-order variables (although, this syntax is as expressive as the original set of operators [155]).

$$\phi \ ::= \sim \phi \mid \phi \ \texttt{\&} \ \psi \mid \texttt{ex2}\, P : \phi \mid P \ \texttt{sub}\ Q \mid P \texttt{=} Q \setminus R \mid P \texttt{=} Q\texttt{+1}$$

where ϕ denotes a WS1S formula, and P, Q and R denote second-order variables. A string interpretation can be given to finite sets of natural numbers; to the finite set P corresponds any string $w = w_0 w_1 \ldots w_n \in \mathbb{B}^*$, $\mathbb{B} = \{0, 1\}$, such that $P = \{\, i \mid w_i = 1 \wedge 0 \leq i \leq n\}$.

For example, the set $P = \{0, 1, 3\}$ can be interpreted as the string $w = 1101$. Note that, $P = \{0, 1, 3\}$ can actually be interpreted as any string $w' \in 11010^*$. The semantics are then extended such that a formula with k variables is interpreted over strings $w \in (\mathbb{B}^k)^*$. For example, $P = \{0, 1, 2\}$ and $Q = \{1, 2, 3\}$ can be interpreted as the string $w = (10)(11)(11)(01)$,

$$\begin{array}{lcccc} P\text{-track} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ Q\text{-track} & & & & \\ & 0 & 1 & 2 & 3 \end{array}$$

Given a string w, the value of a second variable P is encoded as $w(P) = \{i \mid$ the i-th bit in the P-track of w is $1\}$. Satisfiability of WS1S formulae is therefore defined in terms of binary strings, as follows.

$$
\begin{array}{lll}
w \models \sim \phi & \text{iff} & w \not\models \phi \\
w \models \phi \texttt{ \& } \psi & \text{iff} & w \models \phi \text{ and } w \models \psi \\
w \models \texttt{ex2 } P\texttt{:}\phi & \text{iff} & \text{exists } w' \text{ s.t. } w' \approx_P w \text{ and } w' \models \phi \\
w \models P \texttt{ sub } Q & \text{iff} & w(P) \subseteq w(Q) \\
w \models P \texttt{ = } Q \setminus R & \text{iff} & w(P) = w(Q) \backslash w(R) \\
w \models P \texttt{ = } Q\texttt{+1} & \text{iff} & w(P) = \{\, i+1 \mid i \in w(Q) \,\}
\end{array}
$$

where $w' \approx_P w$ if w' is the shortest string that interprets every variable in the formula as w does, save possibly for P (i.e., $w'(Q) = w(Q)$, for all $Q \neq P$). Note that w' may be longer than w; in general, $w' = w.(0,\ldots,0,\mathtt{x}_P,0,\ldots,0)^*$, where $\mathtt{x}_P$ denotes that the value of the bit in the P-track is irrelevant. Given this interpretation, a WS1S formula ϕ defines a language of satisfying strings $\mathcal{L}(\phi) = \{w \mid w \models \phi\}$, which corresponds to its models.

A deterministic finite automaton A_ϕ, $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$, can be constructed inductively on the structure of ϕ. Basic automata are constructed to accept the models of each atomic subformula (P `sub` Q, P `=` $Q \setminus R$ and P `=` Q`+1`). These automata are depicted in Figure 6.3.[4]

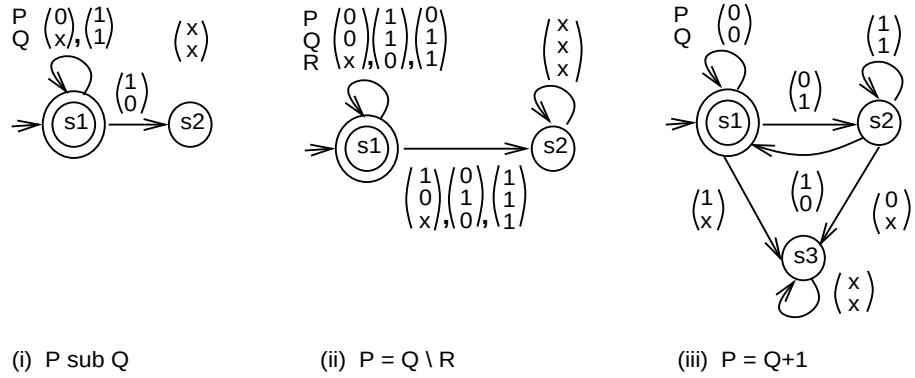
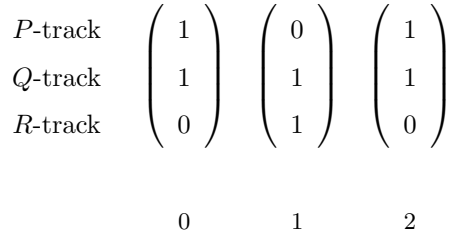


Figure 6.3: Basic Automata for Atomic Formulae (Simplified WS1S)

Consider, for example, the automaton for P `=` $Q \setminus R$ in Figure 6.3(ii). The automaton accepts the model $P = \{0, 2\}$, $Q = \{0, 1, 2\}$ and $R = \{1\}$, which corresponds to a string w leading to an accepting state (`s1`), where w is of the form:

[4]The initial state (double circle) is the only accepting state. To simplify the presentation, transitions are labelled with multiple letters.



Boolean operators correspond to well-known operations on automata. Negation ($\sim \phi$) translates to automata complementation ($\complement A_\phi$), and conjunction (ϕ `&` ψ) is implemented by the product automaton ($A_\phi \times A_\psi$).

Quantification (`ex2` P`:`ϕ) transforms the base automaton, A_ϕ, in a number of steps. First, any state for which a path exists that leads to an accepting state via $(0, \ldots, 0, \mathtt{x}_P, 0, \ldots, 0)^*$, is made accepting. This accounts for the fact that the strings satisfying `ex2` P `:`ϕ may be shorter than those satisfying ϕ. Then, for any state s and any pair of outgoing transitions from s which only differ in the P-track, (s, l, s') and $(s, \overline{l}, s'')$, a pair of complementary transitions are added to the automaton, (s, l, s'') and $(s, \overline{l}, s')$. This non-determinism represents the different values of P which may satisfy ϕ.

Finally, the resulting automaton is determinised and minimised (quantification, then, may cause an exponential increase in the automaton size due to determinisation).

MONA implements the conceptual translation discussed above, but a number of syntactic transformations are first applied to the formula to reduce it to a simplified language (this language is similar to the one we have presented here, although it contains more atomic subformulae, e.g., P `=` Q `union` R). The implementation considers, as well, some technical issues such as the interpretation of first-order variables as second-order variables, the interpretation of Boolean variables, and checking satisfiability of formulae under restrictions. A discussion of these problems (and the solutions implemented in MONA) is beyond the scope of this thesis; we refer the reader to [95, 96].

Meyer [114] showed that the time and space for translating WS1S formulae to automata, in the worst case, is non-elementary in the nesting of quantifiers and negation (state explosion may occur as a result of automata determinisation). Because this complexity cannot be avoided in the worst case (non-elementary complexity has been proved to be a lower-bound for the translation), MONA focused instead on improving the performance of the translation algorithm in the general case (under the hypothesis that the worst-case scenario would not arise frequently in practice). Efficient data-structures, algorithms and different optimisations were included for this purpose [97]. In the following section we elaborate on the use of BDDs in MONA to represent automata. Other optimisations are explained in detail in [97].

6.3.1 Encoding Deterministic Finite Automata with BDDs

BDDs are a well-known data structure to represent boolean functions [53, 52]. In particular, MONA uses *shared*, *multiterminal* BDDs [81, 94] to efficiently encode the automata that result from translating WS1S formulae.

BDDs in MONA encode just the automaton's transition relation; states, instead, are modelled explicitly. In a shared, multiterminal BDD, a vector represents the states of the automaton. Every element in this vector points to a direct acyclic graph that encodes the outgoing transitions of the corresponding state. Let V_1, V_2, ..., V_k be an enumeration of the variables in the formula. The graph is ordered in levels, such that nodes in level k are labelled with V_k, and point to two other nodes (via 0- and 1-transitions) in some lower levels $k', k'' > k$. Leaf nodes (at the bottom) point to elements in the state vector: these denote the "next state" of the encoded automaton's transitions. Figure 6.4 shows the BDD representation for the DFA corresponding to $P = Q \setminus R$ (see the corresponding automaton in Figure 6.3(ii)).

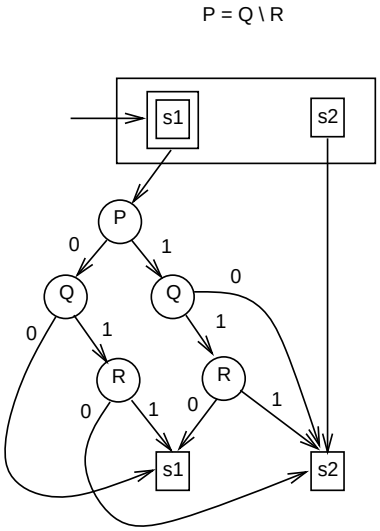


Figure 6.4: A BDD Encoding of DFAs

By way of example, consider the transition `(s1,110,s1)` in the automaton of Figure 6.3(ii); this corresponds to a letter that has a 1 in the P- and Q- tracks, and a 0 in the R-track. This is represented in the BDD by the path,

$$\texttt{s1} \longrightarrow P \xrightarrow{1} Q \xrightarrow{1} R \xrightarrow{0} \texttt{s1}$$

In addition, note in the graph that `s2` is linked directly to itself, without any reference to internal (i.e., variable) nodes. Effectively, the automaton of Figure 6.3(ii) remains in `s2` no matter the current string; because values in the P-, Q- and R- tracks are irrelevant, references to the internal nodes were removed (in BDD-terms, redundant tests were removed). The BDD

shows, as well, that transitions `(s1,110,s1)` and `(s1,111,s2)` share a common subpath (in BDD-terms, isomorphic subtrees were identified, and duplicates removed),

$$\texttt{s1} \longrightarrow P \xrightarrow{1} Q \xrightarrow{1} R$$

By using BDDs, MONA is able to compactly encode the large alphabets arising in DFAs. As we have illustrated previously, a WS1S formula with k variables is interpreted on strings in $\mathbb{B}^k$. There are, therefore, 2^k different letters for the DFA to read, and so the size of the DFA's transition table is exponential on the number of variables in the source formula. A BDD representation of transitions, on the other hand, is usually much more succinct.

Let us conclude this section by pointing out that BDDs are not exempt from exponential blow-ups, if a bad variable ordering prevents compression. Unfortunately, the problem of finding an optimal ordering is difficult (NP-complete). Some ordering heuristics have proved useful in other BDD applications [141, 132, 3, 4], but their effect on WS1S formulae has not been investigated yet. Currently, MONA does not include any ordering heuristics; the order chosen is that in which variables are declared in the source file.

6.4 Summary

We presented WS1S, a decidable logic with the expressive power of regular language on finite words. Despite the non-elementary complexity of this logic, MONA implements an efficient decision procedure which has been successfully applied in many non-trivial problems.

MONA translates a WS1S formula into an equivalent finite automaton, where models are represented by paths from the initial to an accepting state (rejecting state for counterexamples). MONA optimises the translation in different ways (e.g., by identifying common sub-expressions), and compactly represents the automaton using BDDs.

In the following chapters, we will see how WS1S and MONA help us to address limitations of timed automata and model-checkers (Chapter 2). Chapter 7 will show that MONA can be used as an efficient decision procedure for PITL, an interval temporal logic where certain requirements are easier to express than in Uppaal and other model-checkers. Chapter 8 will introduce Discrete Timed Automata, a deductive framework where WS1S is used as the assertion language, and MONA serves as tool support for invariance proofs. Discrete Timed Automata support unbounded data domains, and specifications are time reactive by construction.

Chapters 7 and 8 also reveal shortcomings that stem from the restricted arithmetic of WS1S (limitations in expressive power), and the lack of variable-ordering heuristics in MONA (possible state explosion).

Chapter 7

MONA: A Decision Procedure for Propositional ITL

Abstract. Propositional Interval Temporal Logic (PITL) is a linear-time logic with the expressive power of regular languages on finite words. Time constraints, sequences and iteration of events, and past computations, can be intuitively expressed in PITL. Thus, PITL overcomes expressiveness limitations of branching-time requirements languages, as adopted by Uppaal and other real-time model-checkers (Chapter 2). However, PITL's decision procedure has a non-elementary worst-case complexity, which has hampered the development of efficient tool support and further application of PITL in practice.

In this chapter, we show that MONA (Chapter 6) can be used as an efficient decision procedure for PITL, despite the logic's complexity. We develop a translation from PITL to WS1S (and thus, to MONA specifications), and implement this translation in a tool, PITL2MONA, which acts as a front-end to MONA. In this way, PITL2MONA+MONA can be considered an efficient implementation of an automata-based decision procedure for PITL.

Organization. In Section 7.1 we give an overview of Interval Temporal Logic. We introduce the logic PITL in Section 7.2. In Section 7.3, we present a WS1S semantics for PITL formulae, which we use to obtain a translation from PITL to MONA (this translation is proven correct in Appendix C). We introduce the tool PITL2MONA in Section 7.4, and evaluate its performance on a number of examples. Section 7.5 evaluates PITL as a specification language for real-time systems. Section 7.6 discuss related work. In Section 7.7 we draw conclusions and suggest further research.

7.1 Interval Temporal Logic

Interval Temporal Logic (ITL) [120, 122, 125, 128, 126][1] is a linear-time temporal logic that is interpreted over finite state sequences (called intervals).[2] ITL has been applied in different problems, such as specification and verification of hardware devices [78, 120, 121, 76], temporal logic programming [122, 64, 99, 77] and refinement [56], specification of multimedia documents [41] and human computer interaction [37, 42]. Infinite-time extensions to ITL have been proposed in [124, 64, 128] (however, in this thesis, we restrict our analysis to finite-time ITL).

Our interest in ITL comes from its natural notation and expressiveness. Operators such as chop ($;$), projection ($proj$) and chop-star (*) support sequential composition, repetitive behaviour and refinement [122, 123, 124, 64]. High-level operators can be defined in ITL to denote iteration, conditional tests and assignments, as used in imperative programming languages; thus, ITL lends itself to execution [122, 99].

Furthermore, ITL can express quantitative time constraints; the operator $len(n)$ holds in intervals with n states, thus providing an abstraction to quantify discrete time (when moving from one state in the interval to the next denotes time passing by 1 time unit). This allows ITL to specify real-time systems; e.g., Bowman et al. [41] have exploited this feature of ITL in the specification of multimedia documents and related requirements.

ITL can be seen as an alternative notation which overcomes certain expressiveness limitations of conventional point-based temporal logics, such as CTL [58] and LTL [111]. One drawback of branching-time logics (such as CTL or TCTL [6]) is that formulae do not represent requirements in an intuitive manner, and are difficult to understand for non-logic experts [161, 14].

For instance, in model-checkers such as Uppaal [22] and Kronos [169] (whose requirements languages are based on CTL and TCTL, respectively), it is difficult (and sometimes, impossible) to express requirements that refer to sequences and iteration of events, or past computations (Section 2.4). This affects our confidence in verification, e.g., we may assert the satisfiability of formulae that do not represent the intended requirements.

On the other hand, linear-time logics (such as LTL with past operators) are more intuitive than those based on branching-time, but sequences and iteration of events are difficult to express.

These limitations of point-based logics have motivated researchers to consider requirements languages with the power of regular expressions and past operators [166, 136, 140, 161]. Although, in general, past operators do not add expressive power, they facilitate requirements that refer to past events; for instance, LTL formulae with past operators can be exponentially more succinct than equivalent formulae with just future operators [113].

[1] See also `www.cse.dmu.ac.uk/STRL/ITL//itlhomepage.html`.

[2] This chapter refers to *local* ITL, in which propositional variables are satisfiable on a given interval if and only if they are satisfiable at the initial state [120].

In this chapter we consider a rich propositional subset of ITL over finite intervals, PITL, which has the expressive power of regular languages over finite words [128]. PITL includes the *len* operator, finite quantification (i.e., quantification over propositional variables), (future) operators such as until, chop, chop-star (*) and projection, and past operators such as previous ($\ominus$), chop in the past ($\overset{\sim}{;}$) and since (*since*). These operators permit a natural representation of sequence, iteration and past computations (we formally present PITL in Sections 7.2 and 7.2.2).

However, PITL's decision procedure has a non-elementary worst-case complexity [120], which has hampered the development of efficient tool support and further application of PITL in practice. Here we use WS1S and MONA (Chapter 6) to address this problem.

We show that MONA can be used as an efficient decision procedure for PITL, despite its complexity. We develop a translation from PITL to WS1S (and thus, to MONA specifications), and implement this translation in a tool, PITL2MONA, which acts as a front-end for MONA (Sections 7.3 and 7.4). In this way, PITL2MONA+MONA can be considered an efficient implementation of an automata-based decision procedure for PITL. Furthermore, we show that our implementation outperforms a tableau-based decision procedure for a similar propositional subset of ITL (Section 7.4.3). In addition, we assess the strengths and weaknesses of PITL in real-time settings (Section 7.5).

7.2 PITL – Syntax and Semantics

7.2.1 PITL$^+$: Future Operators

This section presents syntax and semantics for the future subset of PITL (PITL$^+$); past operators will be considered in Section 7.2.2. The syntax of PITL$^+$ is given by the following BNF,

$$P ::= \quad v \mid \mathit{false} \mid \neg P \mid P \vee Q \mid \exists v.\, P \mid \mathit{empty} \mid \bigcirc P \mid P\ ;\ Q \mid P^* \mid P\ \mathit{until}\ Q \mid P\ \mathit{proj}\ Q$$

where P is a PITL$^+$ formula and $v \in \mathbb{V}$, where $\mathbb{V}$ is a set of propositional variables. Operators (*;*), (*) and (*proj*) are usually called, respectively, chop, chop-star and projection. PITL$^+$ is semantically interpreted over finite, non-empty state sequences, called intervals. Let $\mathcal{I}$ denote the set of all intervals; an interval $\sigma \in \mathcal{I}$ has the form,

$$\sigma_0 \sigma_1 \ldots \sigma_{|\sigma|}$$

where $|\sigma| > 0$ denotes the length of an interval and σ_i denotes the i-th state of σ. One-state intervals are referred to as *empty* intervals; by convention, the length of an interval is the number of states minus one, and so the length of empty intervals is zero.

We use σ^i and ${}^i\sigma$ to denote the i-th prefix and suffix of σ, respectively.

$$\begin{aligned} \sigma^i &\triangleq \sigma_0 \dots \sigma_i \\ {}^i\sigma &\triangleq \sigma_i \dots \sigma_{|\sigma|} \end{aligned}$$

Intervals describe the behaviour of variables in time. Every state is mapped to a set of variables in $\mathbb{V}$, precisely the set of variables that are true at that state; we use $v \in \sigma_i$ to denote that $v \in \mathbb{V}$ is true at state σ_i. Given two intervals σ and σ', and a variable $v \in \mathbb{V}$, we use $\sigma' \approx_v \sigma$ to denote that the intervals are similar except maybe for the behaviour of v:

$$\sigma' \approx_v \sigma \;\text{ iff }\; |\sigma'| = |\sigma| \text{ and for all } v' \neq v \text{ and } 0 \leq i \leq |\sigma|,\, v' \in \sigma'_i \Leftrightarrow v \in \sigma_i$$

The semantics of PITL$^+$ is given by the following definitions,

$$\begin{array}{lll}
\sigma \models v & \text{iff} & v \in \sigma_0 \\
\sigma \not\models \mathit{false} & & \\
\sigma \models \neg P & \text{iff} & \sigma \not\models P \\
\sigma \models P \vee Q & \text{iff} & \sigma \models P \text{ or } \sigma \models Q \\
\sigma \models \exists v.\, P & \text{iff} & \text{exists } \sigma' \text{ s.t. } \sigma' \approx_v \sigma \text{ and } \sigma' \models P \\
\sigma \models \mathit{empty} & \text{iff} & |\sigma| = 0 \\
\sigma \models \bigcirc P & \text{iff} & |\sigma| > 0 \text{ and } {}^1\sigma \models P \\
\sigma \models P \;;\; Q & \text{iff} & \text{exists } k \in \mathbb{N} \text{ s.t. } \sigma^k \models P \text{ and } {}^k\sigma \models Q \\
\sigma \models P^* & \text{iff} & |\sigma| = 0 \text{ or exists a sequence } k_0, k_1, \dots, k_m \in \mathbb{N} \text{ s.t.} \\
 & & k_0 < k_1 < \dots < k_m,\, k_0 = 0,\, k_m = |\sigma| \text{ and} \\
 & & \text{for all } 0 \leq i < m,\, {}^{k_i}(\sigma^{k_{i+1}}) \models P \\
\sigma \models P \textit{ until } Q & \text{iff} & \text{exists } k \in \mathbb{N} \text{ s.t. } {}^k\sigma \models Q \text{ and for all } j \in \mathbb{N}, j < k,\, {}^j\sigma \models P \\
\sigma \models P \textit{ proj } Q & \text{iff} & |\sigma| = 0 \text{ or exists a sequence } k_0, k_1, \dots, k_m \in \mathbb{N} \text{ s.t.} \\
 & & k_0 < k_1 < \dots < k_m,\, k_0 = 0,\, k_m = |\sigma|, \text{ and} \\
 & & \text{for all } 0 \leq i < m,\, {}^{k_i}(\sigma^{k_{i+1}}) \models P \text{ and } \sigma_{k_0}\sigma_{k_1} \dots \sigma_{k_m} \models Q
\end{array}$$

where $\sigma \in \mathcal{I}$ is an interval; $\models$ is the satisfiability relation (as usual, a PITL$^+$ formula is valid if it is satisfiable in every possible interval); $v \in \mathbb{V}$ is a propositional variable; and P, Q are PITL$^+$ formulae.[3]

[3]Our definitions for chop-star and projection are slightly different from those of Moszkowski (see e.g., [122, 127]). It turns out that both semantics are equivalent; however, the translation to WS1S is simplified if we rule out empty iterations. We prove this equivalence only for chop-star (see Theorem C.1 in appendix C); the proof for projection can be derived in the same way.

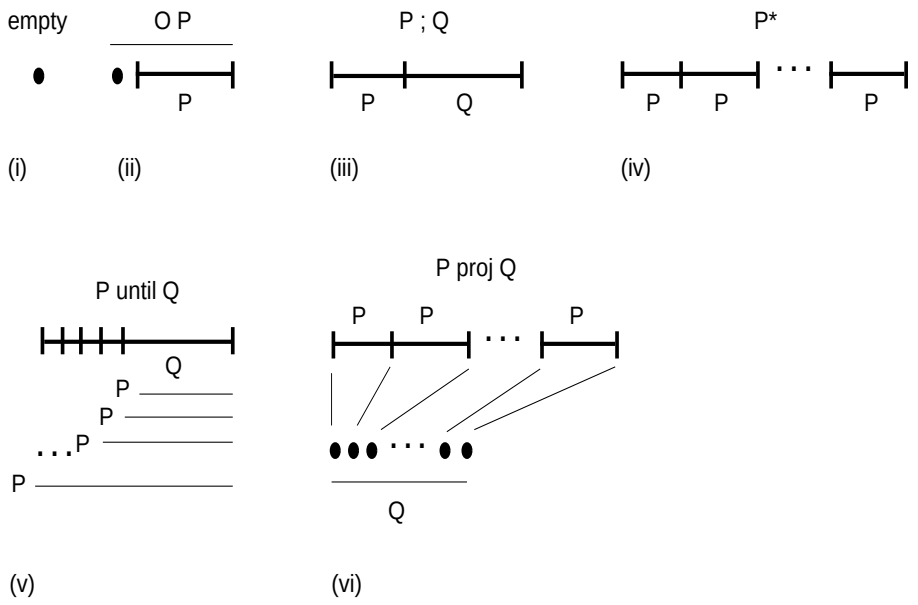


Figure 7.1: PITL$^+$ operators

The semantics of PITL$^+$ operators can be depicted as in Figure 7.1; below, we offer some intuitive explanations.

(i) *empty* is true on any 1-state interval.

(ii) $\bigcirc P$ is true on σ if P is true on the suffix ${}^1\sigma$. Note, the next operator $\bigcirc$ is given a *strong* interpretation, in which $\bigcirc P$ fails on empty intervals. A weak interpretation (satisfiable in any empty interval) can be derived, which results in the *weak-next* operator $\textcircled{w} P \triangleq \neg \bigcirc \neg P$.

(iii) P *;* Q is true on σ if σ can be split at some state σ_k, such that P is true on the prefix σ^k, and Q is true on the suffix ${}^k\sigma$. Note, chop is a "fusion" variant of the concatenation operator for regular expressions: σ^k and ${}^k\sigma$ share a common state, σ_k.

(iv) P^* is true on σ if either σ is an empty interval, or a sequence of natural numbers $0 = k_0 < k_1 < \ldots < k_m = |\sigma|$ can be found that splits σ into a sequence of subintervals ${}^{k_i}(\sigma^{k_{i+1}})$, $0 \leq i < m$, where P is true on each subinterval. By definition, P^* is satisfiable on empty intervals, for any PITL$^+$ formula P (even for $P \triangleq \mathit{false}$).

(v) P *until* Q is true on σ if σ can be split at some state σ_k, such that Q is true on the suffix ${}^k\sigma$, and P is true on every previous suffix ${}^j\sigma$, $j < k$. Note, P *until* Q holds trivially in σ if Q does, regardless of P.

(vi) P *proj* Q is true on σ if either σ is an empty interval, or a sequence of natural numbers

$0 = k_0 < k_1 < \ldots < k_m = |\sigma|$ can be found that splits σ into a sequence of subintervals ${}^{k_i}(\sigma^{k_{i+1}})$, $0 \leq i < m$, where P is true on each subinterval, and Q is true on the interval formed by "glueing" together the subintervals' end points, i.e., Q is true on $\sigma' = \sigma_{k_0}\sigma_{k_1}\ldots\sigma_{k_m}$.

A wealth of other useful operators can be derived from this set, as shown below.

$true$	$\triangleq \neg\, false$	
$P \wedge Q$	$\triangleq \neg(\neg P \vee \neg Q)$	
$P \Rightarrow Q$	$\triangleq \neg P \vee Q$	
$P \equiv Q$	$\triangleq (P \Rightarrow Q) \wedge (Q \Rightarrow P)$	
$\Diamond\ P$	$\triangleq true\ ;\ P$	(true in some suffix)
$\Box\ P$	$\triangleq \neg\Diamond\neg P$	(true in all suffixes)
$more$	$\triangleq \neg\, empty$	(non-empty interval)
$skip$	$\triangleq \bigcirc empty$	(two-state interval)
$\boxed{\text{m}}\ P$	$\triangleq \Box(more \Rightarrow P)$	(true in all suffixes, save perhaps at the last state)
$\diamondsuit{\scriptstyle \text{a}}\ P$	$\triangleq true\,;P\,;true$	(true in some subinterval)
$\boxed{\text{a}}\ P$	$\triangleq \neg\diamondsuit{\scriptstyle \text{a}}\neg P$	(true in all subintervals)
$fin(P)$	$\triangleq true\,;(empty \wedge P)$	(true at last state)
$halt(P)$	$\triangleq (\boxed{\text{m}}\neg P)\,;(empty \wedge P)$	(only true at last state)
$len(n)$	$\triangleq \underbrace{\bigcirc\bigcirc\cdots\bigcirc}_{n\ times} empty$	(interval of length n)
$len_{\geq}(n)$	$\triangleq len(n)\,;true$	(interval of length $\geq n$)
$len_{>}(n)$	$\triangleq len(n)\,;more$	(interval of length $> n$)
$len_{\leq}(n)$	$\triangleq \neg\, len_{>}(n)$	(interval of length $\leq n$)
$len_{<}(n)$	$\triangleq \neg\, len_{\geq}(n)$	(interval of length $< n$)

Interestingly, chop can be used to describe sequential composition, and "for" and "while" loops can be defined in terms of chop-star and projection.

$$\begin{array}{ll} while\ P\ do\ Q & \triangleq (P \wedge Q)^* \wedge halt(\neg P) \\ for\ n\ times\ do\ P & \triangleq P\ proj\ len(n) \end{array}$$

where P, Q are arbitrary ITL formulae and n is a natural number. [4] Intuitively, the construct

$$\mathit{while}\ P\ \mathit{do}\ Q$$

holds in any interval that can be divided in any number of consecutive subintervals, where both P and Q hold in each subinterval, and P does not hold in the last state of interval. Likewise, the construct,

$$\mathit{for}\ n\ \mathit{times}\ \mathit{do}\ P$$

holds in any interval that can be divided in n consecutive subintervals, where P holds in each subinterval. Note, as well, that chop-star and until can be defined in terms of projection and quantification, respectively.

$$\begin{array}{ll} P^* & \triangleq\ P\ \mathit{proj}\ \mathit{true} \\ P\ \mathit{until}\ Q & \triangleq\ \exists\, v.\, (\Box(v \equiv P) \wedge (\boxed{\mathrm{m}}\, v)\,;Q) \end{array}$$

Examples: Future Operators. Consider the following interval,

$$\sigma = a\,\{a,b\}\,a\,b = \sigma_0 \ldots \sigma_3$$

where $\mathbb{V} = \{a, b\}$, $|\sigma| = 3$ and $\sigma_0 = \{a\}$, $\sigma_1 = \{a, b\}$, $\sigma_2 = \{a\}$ and $\sigma_3 = \{b\}$. The following formulae (i)-(iii) are satisfiable over σ, while (iv)-(vi) are not.

$$\begin{array}{llllll} \text{(i)} & a\,;b & \text{(ii)} & a^* & \text{(iii)} & a\ \mathit{proj}\ \boxed{\mathrm{m}}\, a \\ \text{(iv)} & (a \wedge \mathit{empty})\,;b & \text{(v)} & \mathit{len}(2)^* & \text{(vi)} & a\ \mathit{proj}\ \Box a \end{array}$$

The formula (i) holds because a holds in σ^1 and b holds in ${}^1\sigma$. In contrast, (iv) does not hold because the conjunct is forcing the prefix to contain just the initial state, where b does not hold.

The formula (ii) holds because σ can be split into any number of consecutive subintervals, say $\sigma_0\sigma_1$, $\sigma_1\sigma_2$ and $\sigma_2\sigma_3$, where each subinterval is a model for a (i.e., a is true in the first state of every subinterval). Compare this with (v), which does not hold because there is no way to split σ in any number of consecutive 3-state subintervals.

The formula (iii) holds because σ can be split into consecutive subintervals where a is true in the first state, say $\sigma_0\sigma_1$ and $\sigma_1\sigma_2\sigma_3$, and the interval resulting from glueing together the subintervals' boundaries, i.e., $\sigma' = \sigma_0\sigma_1\sigma_3$, satisfies $\boxed{\mathrm{m}}\, a$ (note, in this case, $\sigma' = aab$). Note that projection (as is also the case with chop-star) does not force any particular way of splitting the base interval; however, the resulting (projected) interval inherits the base interval's first and

[4] In the "while" loop, P will typically denote a *point formula*, i.e., the satisfiability of P can be determined totally by considering just the first state of the interval.

last states. Therefore, every interval projected from σ will have a being false in the last state, and so $\Box a$ cannot hold in any such interval. The difference between (iii) and (vi) is that $\boxed{\text{m}}\, a$ (unlike $\Box a$) does not require a to hold in the last state.

7.2.2 PITL: Extending PITL$^+$ With Past Operators

The logic PITL extends PITL$^+$ with three past operators, *previous* ($\ominus$), *since* and *chop in the past* ($\overset{\sim}{;}$). These operators are based on those of Bowman et al. [41], adapted to the propositional setting. The syntax of PITL is given by,

$$\begin{aligned} P ::= \quad & v \mid \mathit{false} \mid \neg P \mid P \vee Q \mid \exists v.\, P \\ & \mid \mathit{empty} \mid \bigcirc P \mid P \;;\; Q \mid P^* \mid P \;\mathit{until}\; Q \mid P \;\mathit{proj}\; Q \\ & \mid \ominus P \mid P \;\overset{\sim}{;}\; Q \mid P \;\mathit{since}\; Q \end{aligned}$$

To accommodate past operators, we will interpret PITL formulae over pairs of the form (σ, j), where $\sigma \in \mathcal{I}$ and $0 \leq j \leq |\sigma|$. The state σ_j is the *current state*, which splits σ into a *history* prefix, σ^{j-1} (if $j > 0$), and a *future* suffix, ${}^{j}\sigma$. As in [41], the history prefix denotes the sequence of past states, and we consider the current state to be part of the future suffix. The semantics of past operators is as follows.

$$\begin{array}{lll} (\sigma, j) \models \ominus P & \text{iff} & j > 0 \text{ and } (\sigma, j-1) \models P \\ (\sigma, j) \models P \;\overset{\sim}{;}\; Q & \text{iff} & \text{exists } k \in \mathbb{N}, 0 \leq k \leq j \text{ s.t.} \\ & & (\sigma, \sigma_k) \models P \text{ and } ({}^{k}\sigma, j-k) \models Q \\ (\sigma, j) \models P \;\mathit{since}\; Q & \text{iff} & \text{exists } k \in \mathbb{N},\ 0 \leq k \leq j \text{ s.t. } (\sigma, k) \models Q \text{ and} \\ & & \text{for all } r,\ k < r \leq j,\ (\sigma, r) \models P \end{array}$$

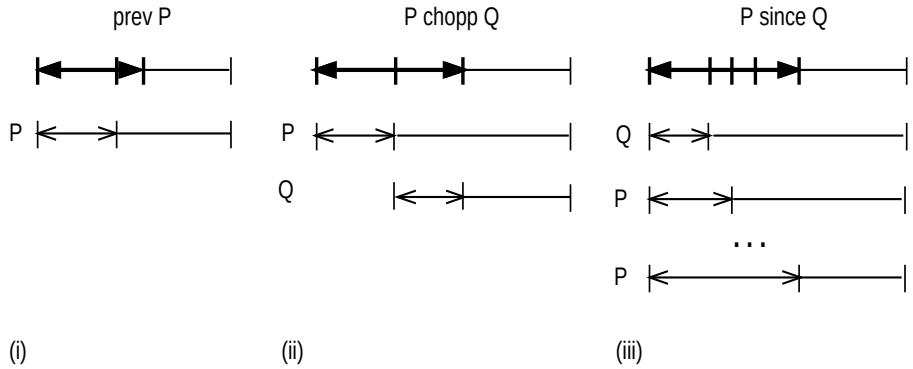


Figure 7.2: Past operators: (i) previous, (ii) chop in the past, and (iii) since

The behaviour of past operators is illustrated in Figure 7.2 (the history prefix is delimited by double arrows); next, we offer informal explanations.

(i) $\ominus P$ holds in some interval σ if it holds in σ by moving one state into the past;

(ii) $P \stackrel{\sim}{;} Q$ holds in some interval σ if P holds in some past suffix of σ, and Q holds in σ with a truncated history, starting at P's current state. Note, chop in the past splits the history of σ very much in the same way that chop splits the future.

(iii) P *since* Q holds in some interval σ if Q holds in σ, or Q holds in some past suffix of σ and P holds in every past suffix of σ between Q and σ's current states.

The semantics of future operators w.r.t. a pair (σ, j) is given below. This interpretation on pairs is conservative w.r.t. the semantics on intervals given in Section 7.2 for future operators, in the following sense: for any PITL formula P where past operators do not occur, and any pair (σ, j), then $(\sigma, j) \models P$ iff ${}^{j}\sigma \models P$. This completes, together with that of past operators, the semantics for PITL.

$$
\begin{array}{lll}
(\sigma, j) \models v & \text{iff} & v \in \sigma_j \\
(\sigma, j) \not\models \mathit{false} & & \\
(\sigma, j) \models \neg P & \text{iff} & (\sigma, j) \not\models P \\
(\sigma, j) \models P \vee Q & \text{iff} & (\sigma, j) \models P \text{ or } (\sigma, j) \models Q \\
(\sigma, j) \models \exists v.\ P & \text{iff} & \text{exists } \sigma' \text{ s.t. } \sigma' \approx_v \sigma \text{ and } (\sigma', j) \models P \\
(\sigma, j) \models \mathit{empty} & \text{iff} & j = |\sigma| \\
(\sigma, j) \models \bigcirc P & \text{iff} & j < |\sigma| \text{ and } (\sigma, j+1) \models P \\
(\sigma, j) \models P\ ;\ Q & \text{iff} & \text{exists } k \in \mathbb{N}, j \leq k \leq |\sigma| \text{ s.t. } (\sigma^k, j) \models P \text{ and } (\sigma, k) \models Q \\
(\sigma, j) \models P^* & \text{iff} & j = |\sigma| \text{ or exists a sequence } k_0, k_1, \ldots, k_m \in \mathbb{N} \text{ s.t.} \\
 & & k_0 < k_1 < \ldots < k_m,\ k_0 = j,\ k_m = |\sigma| \text{ and} \\
 & & \text{for all } 0 \leq i < m,\ (\sigma^{k_{i+1}}, k_i) \models P \\
(\sigma, j) \models P \ \mathit{until}\ Q & \text{iff} & \text{exists } k \in \mathbb{N}, j \leq k \leq |\sigma| \text{ s.t. } (\sigma, k) \models Q \text{ and} \\
 & & \text{for all } r \in \mathbb{N}, j \leq r < k,\ (\sigma, r) \models P \\
(\sigma, j) \models P \ \mathit{proj}\ Q & \text{iff} & j = |\sigma| \text{ or exists a sequence } k_0, k_1, \ldots, k_m \in \mathbb{N} \text{ s.t.} \\
 & & k_0 < k_1 < \ldots < k_m,\ k_0 = j,\ k_m = |\sigma|, \text{ and} \\
 & & \text{for all } 0 \leq i < m,\ (\sigma^{k_{i+1}}, k_i) \models P \text{ and} \\
 & & (\sigma_0 \ldots \sigma_{k_0} \sigma_{k_1} \ldots \sigma_{k_m}, k_0) \models Q
\end{array}
$$

The behaviour of future operators (Figure 7.3) does not take into account the history prefix, but the semantics of past operators must consider the possible expansion of this history. On the one hand, past operators may occur under the scope of future operators; on the other hand, when future operators are evaluated, the history prefix is extended to match the corresponding current state.

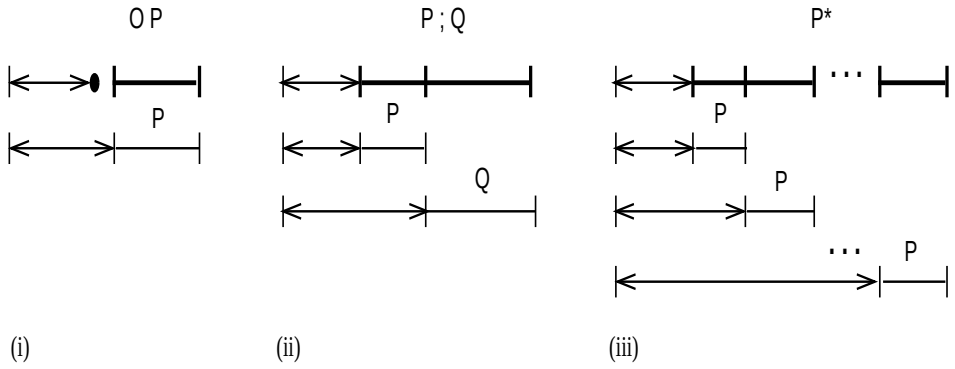


Figure 7.3: Future operators and history: (i) next, (ii) chop and (iii) chop-star

Examples: Past Operators. Consider a pair $(\sigma, 3)$, where

$$\sigma = baa \,|\, abab = \sigma_0 \ldots \sigma_6$$

The "|" symbol splits σ into a history prefix, σ^2, and a future suffix, ${}^3\sigma$ (which includes the current state, σ_3). The following formulae are satisfiable over σ.

$$\text{(i) } \ominus a \quad \text{(ii) } b \stackrel{\sim}{;} \mathit{true} \quad \text{(iii) } a \mathit{\ since\ } b \quad \text{(iv) } \Box(b \Rightarrow \ominus a)$$

The formula (i) is true in $(\sigma, 3)$ because it is true in σ by moving one state into the past: the current state is σ_3, thus, moving one state into the past yields the past suffix ${}^2\sigma$, which is a model for a. The formula (ii) holds in $(\sigma, 3)$ because b is true in a past suffix of σ. In this case, the *true*-term has the purpose of splitting the history of σ in order to "find" b in the past. This shows, then, how the operator $\diamondsuit\!\!\!\!-$ (eventually in the past) can be defined in terms of chop in the past,

$$\diamondsuit\!\!\!\!- P \triangleq P \stackrel{\sim}{;} \mathit{true}$$

the formula (iii) holds in $(\sigma, 3)$ because it holds in every past suffix of σ, after b and up to the current state (σ_3). The formula (iv) is an example of interaction between future and past operators. The pair (σ, j) is a model for this formula, because every b-suffix in the future is immediately preceded by an a-suffix. Note, in contrast, that a pair $(\sigma, 0)$ would not be a model for $\Box(b \Rightarrow \ominus a)$ because b is true in σ_0 (the first state), and thus, $\ominus a$ is trivially false (although a weak version of $\ominus$ could be derived).

7.3 A WS1S Semantics for PITL Formulae

Our goal is to obtain a translation from any PITL formula P to a corresponding WS1S formula P' with the same meaning, in the sense that P' is satisfiable w.r.t. WS1S semantics if and only if P is satisfiable w.r.t. PITL semantics. This is accomplished by representing PITL intervals as finite sets in $\mathbb{N}$ (i.e., the semantics of WS1S), and expressing the semantics of PITL operators as WS1S formulae.

7.3.1 Intervals as Finite Sets of Natural Numbers

Intervals in PITL (w.r.t. a set of propositional variables $\mathbb{V}$) are finite state-sequences, and every state σ_i can be interpreted as a set of propositional variables: if a variable is included in the set, $v \in \sigma_i$, then that variable is *true* in σ_i.

From a slightly different point of view, intervals can be thought of as mapping every variable in $\mathbb{V}$ to a finite (possibly empty) set of natural numbers; every number corresponds to the index of a state where that variable is *true*. For instance, if V is the set that encodes v in this way, and $v \in \sigma_i$, then $i \in V$. Given that WS1S encodes finite sets of natural numbers as second-order variables, we use second-order variables to encode PITL intervals in WS1S.

Formally, we define a mapping from PITL intervals and propositional variables to finite sets of natural numbers, encoded as WS1S second-order variables. Let $\sigma \in \mathcal{I}$ be an interval w.r.t. a finite set of propositional variables $\mathbb{V}$; $\mathbb{V}_2$ a finite set of WS1S second order variables; and $h : \mathbb{V} \to \mathbb{V}_2$ a renaming bijection. We define a (total) mapping $[\![\ ]\!]_\sigma : \mathbb{V}_2 \to \mathcal{F}(\mathbb{N})$ that assigns a valuation to second-order variables (a finite set of natural numbers), according to the behaviour of propositional variables in σ. The mapping is defined as follows,

$$[\![V]\!]_\sigma \triangleq \{i \mid \exists\, v \in \mathbb{V}.\ V = h(v) \wedge v \in \sigma_i\}$$

For example, if $\mathbb{V} = \{a, b\}$, $\mathbb{V}_2 = \{A, B\}$, $h = \{(a, A), (b, B)\}$, and σ is such that,

$$\sigma = \{a\}_0\, \{a, b\}_1\, \{a, b\}_2\, \{b\}_3\, \{b\}_4$$

then, $[\![\ ]\!]_\sigma = \{\, (A, \{0, 1, 2\}), (B, \{1, 2, 3, 4\})\, \}$.

7.3.2 Interpreting PITL in WS1S

We have shown that intervals can be represented in WS1S. If a WS1S semantics is possible for PITL operators (Section 7.2.2), then WS1S must be expressive enough to characterise (a) subintervals (b) the current state in a given interval (so history and future subintervals are properly

distinguished), and (c) the creation of intervals out of nonconsecutive states (so projection can be defined). In this section, we show that WS1S is expressive enough to model these concepts.

Consider the semantics of a PITL formula P w.r.t. a pair (σ, j). Depending on the operators occurring in P, the satisfiability of P may depend on the satisfiability of some subformula Q w.r.t. some (σ', k), where σ' is a subinterval of σ and $0 \leq k \leq |\sigma'|$. Note that, if Q is under the scope of a projection operator, then σ' may not be a sequence of consecutive states.

For instance, let $(\sigma, 0)$ and P be defined as follows.

$$\begin{aligned} \sigma &= c\,a\,b\,a\,b\,a\,b\,a\,c = \sigma_0 \ldots \sigma_8 \\ P &\triangleq \langle\!\langle a \rangle\!\rangle (\ (a \wedge \mathit{len}(2))\ \mathit{proj}\ (\mathit{skip}\,;Q\,;\mathit{skip})\ ) \end{aligned}$$

With this setting, the satisfiability of P in $(\sigma, 0)$ is ultimately determined by the satisfiability of Q in $(\sigma'', 2)$, where $\sigma'' = \sigma_0\sigma_1\sigma_3\sigma_5$ (thus, $\sigma''_2 = \sigma_3$ denotes the current state). We can justify this claim as follows.

- By the semantics of $\langle\!\langle a \rangle\!\rangle$, $(\sigma, 0) \models \langle\!\langle a \rangle\!\rangle P'$ implies that P' is satisfiable in some subinterval of σ, where $P' \triangleq (a \wedge \mathit{len}(2))\ \mathit{proj}\ P''$. Then, it must be the case that,

  $$(\sigma^j, i) \models (a \wedge \mathit{len}(2))\ \mathit{proj}\ P''$$

  for some $0 \leq i \leq j \leq 8$.

- By the semantics of *proj*, if $(\sigma^j, i) \models (a \wedge \mathit{len}(2))\ \mathit{proj}\ P''$ holds, then ${}^i\sigma^j$ can be split into consecutive models of $(a \wedge \mathit{len}(2))$. Now, these models must be 3-state subintervals of σ, where a is true in the first state. From the structure of σ, then,

  $$(\sigma^7, 1) \models (a \wedge \mathit{len}(2))\ \mathit{proj}\ P''$$

  because ${}^1\sigma^7 = \mathit{abababa}$. Moreover, $P'' \triangleq (\mathit{skip}\,;Q\,;\mathit{skip})$ must be satisfiable in the interval that results from "glueing" together the end-points of models of $(a \wedge \mathit{len}(2))$. Thus,

  $$(\sigma', 1) \models (\mathit{skip}\,;Q\,;\mathit{skip})$$

  where $\sigma' = \sigma_0\sigma_1\sigma_3\sigma_5\sigma_7 = \mathit{caaaa}$.

- By the semantics of *skip* and chop, if $P'' \triangleq \mathit{skip}\,;Q\,;\mathit{skip}$ is satisfiable in $(\sigma', 1)$, then $\sigma_1\sigma_3\sigma_5\sigma_7$ can be split into three consecutive subintervals, where the first $(\sigma_1\sigma_3)$ and last $(\sigma_5\sigma_7)$ subintervals are 2-state intervals; and the middle interval $(\sigma_3\sigma_5)$ is a model for Q.

We conclude that P is satisfiable in $(\sigma, 0)$ if and only if $(\sigma'', 2)$ is a model for Q, where $\sigma'' = \sigma_0\sigma_1\sigma_3\sigma_5$.

The previous example shows that, in order to determine the satisfiability of a PITL formula w.r.t. some interval σ, we need to refer to a sequence of states which are not necessarily consecutive in σ, say,

$$\sigma_{s_0}\sigma_{s_1}\ldots\sigma_{s_m}$$

for some set of indices $S = \{s_0, s_1, \ldots, s_m\}$. In the general case, we must be able to identify a pair (σ', j') in this sequence, where

$$\sigma' = \sigma_{s_k}\sigma_{s_{k+1}}\ldots\sigma_{s_{k+j'}}\ldots\sigma_{s_{k+|\sigma'|}}$$

for some $k \in \mathbb{N}$ $(0 \leq k + |\sigma'| \leq m)$. Let $i, j, n \in S\,(i \leq j \leq n)$ s.t. $i = s_k$, $j = s_{k+j'}$ and $n = s_{k+|\sigma'|}$. Intuitively, σ' is a subinterval constructed from those states in σ represented by indices in S between i and n, and where j points to the current state (i.e., $\sigma'_0 = \sigma_i$, $\sigma'_{j'} = \sigma_j$ and $\sigma'_{|\sigma'|} = \sigma_n$). Correspondingly, We introduce the function $\mathit{subinterval}(i, j, n, S, \sigma)$, defined s.t. $\mathit{subinterval}(i, j, n, S, \sigma) = (\sigma', j')$, where (σ', j') is as mentioned above.

Formally, we define a new satisfiability relation for PITL formulae, $\models_W$, in terms of tuples of the form $(i, j, n, S, \mathcal{M})$, where $S \subseteq \mathbb{N}$, $i, j, n \in S\,(i \leq j \leq n)$ and $\mathcal{M} : \mathbb{V}_2 \to \mathcal{F}(\mathbb{N})$ maps second-order variables to finite sets of natural numbers. This WS1S interpretation of PITL is equivalent to the interval-based semantics, in the sense given by Lemma 7.1 and Theorem 7.2.

LEMMA **7.1.** *For any $\sigma \in \mathcal{I}$, σ' a subinterval of σ (not necessarily contiguous), $0 \leq j' \leq |\sigma'|$, and PITL formula P, the following holds,*

$$(\sigma', j') \models P \;\; \textit{iff} \;\; (i, j, n, S, \mathcal{M}) \models_W P$$

where $S \subseteq \{0, 1, \ldots, |\sigma|\}$, $i, j, n \in S\,(i \leq j \leq n)$, $\mathit{subinterval}(i, j, n, S, \sigma) = (\sigma', j')$ and $\mathcal{M} = [\![\ ]\!]_\sigma$.

Proof. See Appendix C. □

THEOREM **7.2.** *For any $\sigma \in \mathcal{I}$ and PITL formula P, the following holds,*

$$(\sigma, 0) \models P \;\; \textit{iff} \;\; (i, j, n, S, \mathcal{M}) \models_W P$$

where $i = j = 0$, $n = |\sigma|$, $S = \{0, 1, \ldots, |\sigma|\}$ and $\mathcal{M} = [\![\ ]\!]_\sigma$.

Proof. Follows from Lemma 7.1. □

Before we give the formal definition of $\models_W$, we define the predicate $cons(Set, s_1, s_2)$, and the notation $\mathcal{M}' \approx_V \mathcal{M}$. The predicate $cons(Set, s_1, s_2)$ denote that s_2 is the smallest element of Set that is bigger than s_1.

$$cons(Set, s_1, s_2) \triangleq s_1, s_2 \in Set \wedge s_1 < s_2 \wedge \nexists s' \in Set.\, s_1 < s' < s_2$$

We use $\mathcal{M}' \approx_V \mathcal{M}$ to denote that the two mappings $\mathcal{M}'$ and $\mathcal{M}$ are similar except maybe for the second-order variable V. Formally, for second-order variables V and V',

$$\mathcal{M}'(V') = \begin{cases} \mathcal{M}(V') & \text{if } V' \neq V \\ N \in \mathcal{F}(\mathbb{N}) & \text{if } V' = V \end{cases}$$

The satisfiability relation is defined as follows (where v is a propositional variable, and $h(v)$ the corresponding second-order variable).

$(i, j, n, S, \mathcal{M}) \models_W v$	iff	$j \in \mathcal{M}(h(v))$
$(i, j, n, S, \mathcal{M}) \not\models_W \mathit{false}$		
$(i, j, n, S, \mathcal{M}) \models_W \neg P$	iff	$(i, j, n, S, \mathcal{M}) \not\models_W P$
$(i, j, n, S, \mathcal{M}) \models_W P \vee Q$	iff	$(i, j, n, S, \mathcal{M}) \models_W P$ or $(i, j, n, S, \mathcal{M}) \models_W Q$
$(i, j, n, S, \mathcal{M}) \models_W \exists v.\, P$	iff	exists $\mathcal{M}'$ s.t. $\mathcal{M}' \approx_{h(v)} \mathcal{M}$ and $(i, j, n, S, \mathcal{M}') \models_W P$
$(i, j, n, S, \mathcal{M}) \models_W \mathit{empty}$	iff	$j = n$
$(i, j, n, S, \mathcal{M}) \models_W \bigcirc P$	iff	$j < n$ and exists $k \in \mathbb{N}$ s.t.
		$cons(S, j, k)$ and $(i, k, n, S, \mathcal{M}) \models_W P$
$(i, j, n, S, \mathcal{M}) \models_W P\ ;\ Q$	iff	exists $k \in S,\ j \leq k \leq n$ s.t.
		$(i, j, k, S, \mathcal{M}) \models_W P$ and $(i, k, n, S, \mathcal{M}) \models_W Q$
$(i, j, n, S, \mathcal{M}) \models_W P^*$	iff	$j = n$ or exists $k_0, k_1, \ldots, k_m \in S$ s.t.
		$k_0 < k_1 < \ldots < k_m$, $k_0 = j$, $k_m = n$, and
		for all $r,\ 0 \leq r < m$, $(i, k_r, k_{r+1}, S, \mathcal{M}) \models_W P$
$(i, j, n, S, \mathcal{M}) \models_W P \mathit{\ until\ } Q$	iff	exists $k \in S,\ j \leq k \leq n$ s.t.
		$(i, k, n, S, \mathcal{M}) \models_W Q$ and
		for all $r \in S,\ j \leq r < k$, $(i, r, n, S, \mathcal{M}) \models_W P$

$$
\begin{array}{lll}
(i,j,n,S,\mathcal{M}) \models_W P \ \mathit{proj}\ Q & \text{iff} & j = n \text{ or exists } S' \subseteq S \text{ s.t.} \\
 & & S' = \{i, \ldots, k_0, k_1, \ldots, k_m\} \text{ and} \\
 & & \text{for all } t,\ i \leq t \leq k_0,\ t \in S \Leftrightarrow t \in S' \text{ and} \\
 & & k_0 < k_1 < \ldots < k_m,\ k_0 = j,\ k_m = n, \text{ and} \\
 & & \text{for all } r,\ 0 \leq r < m, \\
 & & (i, k_r, k_{r+1}, S, \mathcal{M}) \models_W P \text{ and } (i,j,n,S',\mathcal{M}) \models_W Q \\
(i,j,n,S,\mathcal{M}) \models_W P \ \tilde{;}\ Q & \text{iff} & \text{exists } k \in S,\ i \leq k \leq j \text{ s.t.} \\
 & & (i,k,n,S,\mathcal{M}) \models_W P \text{ and } (k,j,n,S,\mathcal{M}) \models_W Q \\
(i,j,n,S,\mathcal{M}) \models_W P \ \mathit{since}\ Q & \text{iff} & \text{exists } k \in S,\ i \leq k \leq j \text{ s.t.} \\
 & & (i,k,n,S,\mathcal{M}) \models_W Q \text{ and} \\
 & & \text{for all } r \in S,\ k < r \leq j,\ (i,r,n,S,\mathcal{M}) \models_W P \\
(i,j,n,S,\mathcal{M}) \models_W \ominus P & \text{iff} & i < j \text{ and exists } k \in S \text{ s.t.} \\
 & & \mathit{cons}(S,k,j) \text{ and } (i,k,n,S,\mathcal{M}) \models_W P
\end{array}
$$

7.3.3 Translating PITL to MONA

The satisfiability relation, $\models_W$ (see Section 7.3.2), gives an interpretation for PITL formulae in terms of natural numbers and finite sets of natural numbers; quantification over both numbers and sets; relational operations $(<, \leq)$; and set operations, such as set inclusion and tests for membership. All these can be expressed in WS1S; thus, the semantics of a PITL formula can be translated to a MONA specification.

The translation follows the inductive definition of $\models_W$ on tuples $(i, j, n, S, \mathcal{M})$, and can be characterised by the function $\mathit{trans}(i, j, n, S, P)$ (see Table 7.1). The parameters, i, j and n, will be instantiated with either natural numbers or first-order variables; S will be instantiated with second-order variables; and P will represent the PITL formula to translate.

The function *trans*() returns a MONA formula, P_M, such that P_M is satisfiable (w.r.t. WS1S semantics) if and only if P is satisfiable (w.r.t. PITL semantics). Moreover, the model found by MONA for P_M will be readily interpreted as a model for P (respectively, counterexample).

The final outcome of the translation algorithm is a MONA specification, P_S, which includes the necessary declarations to support the formula P_M ($P_M = \mathit{trans}(0, 0, \texttt{length}, \texttt{S}, P)$). This is shown below, in Figure 7.4.

P	$trans(i, j, n, S, P)$
v	j `<=` n `&` j `in` $h(v)$
false	`false`
$\neg P$	$\sim trans(i, j, n, S, P)$
$\exists v.\, P$	`ex2` $h(v)$`:` $trans(i, j, n, S, P)$
$\bigcirc P$	j `<` n `&` `ex1 k:cons(S,j,k)` `&` $trans(i, \mathtt{k}, n, S, P)$
$P \vee Q$	$trans(i, j, n, S, P)$ `|` $trans(i, j, n, S, Q)$
empty	j `=` n
$P\,;Q$	`ex1 k:k in` S `&` j `<=` `k & k <=` n `&` $trans(i, j, \mathtt{k}, S, P)$ `&` $trans(i, \mathtt{k}, n, S, Q)$
P^*	`ex2 K:K sub` S `& min K =` j `& max K =` n `&` `all1 k1,k2:cons(K,k1,k2) =>` $trans(i, \mathtt{k1}, \mathtt{k2}, P)$
P *until* Q	`ex1 k:k in` S `&` j `<=` `k & k <=` n `&` $trans(i, \mathtt{k}, n, S, Q)$ `&` `all1 k1:k1 >=` i `& k1 < k =>` $trans(i, \mathtt{k1}, n, S, P)$
P *proj* Q	`ex2 K:K sub` S `& min K =` j `& max K =` n `&` `all1 k1,k2:cons(K,k1,k2) =>` $trans(i, \mathtt{k1}, \mathtt{k2}, S, P)$ `&` $trans(i, j, n, \mathtt{K}, Q)$
$P \stackrel{\sim}{;} Q$	`ex1 k:k in` S `& k >=` i `& k <=` j `&` $trans(i, \mathtt{k}, n, S, P)$ `&` $trans(\mathtt{k}, j, n, S, Q)$
P *since* Q	`ex1 k:k in` S `&` i `<= k & k <=` j `&` $trans(i, \mathtt{k}, n, S, Q)$ `&` `all1 r:r in` S `& k < r & r <=` j `=>` $trans(i, \mathtt{r}, n, S, P)$
$\ominus P$	i `<` j `&` `ex1 k:k in` S `&` `cons(`S`,k,`j`)` `&` $trans(i, \mathtt{k}, n, S, P)$

Table 7.1: A function to translate PITL to MONA formulae

```
var1 length;
var2 V1,V2,...,Vm;
pred cons(var2 Set, var1 s1,s2) =
         s1 in Set & s2 in Set & s1<s2 &
         ~ ex1 s':s' in Set & s1<s' & s'<s2;
ex2 S: (all1 s:  s<=length <=> s in S) & PM;
```

Figure 7.4: P_S, the MONA specification for the PITL formula P

Let us comment on the declarations of P_S.

- Let $\mathbb{V}$ denote the propositional variables (both free and quantified) occurring in P, and assume that the sets of free and quantified variables are disjoint. Let $\mathbb{V}_2$ be a set of second-order variables, and $h : \mathbb{V} \to \mathbb{V}_2$ the corresponding renaming bijection. Let $\mathbb{V}_2^f \subseteq \mathbb{V}_2$, $\mathbb{V}_2^f = \{V_1, V_2, \ldots, V_m\}$, denote the second-order variables corresponding to free variables in P. Then, P_S declares $\mathbb{V}_2 \cup \{\mathtt{length}\}$ as free variables, where `length` is a first-order variable encoding the length of the model/counterexample for P_M (as found by MONA).

- The predicate *cons*(), as used in $\models_W$, is translated directly into MONA, and declared as part of P_S. Also note that the mapping $\mathcal{M}$ will be represented implicitly by the models of P_M. Finding a model for P_M (i.e., using MONA to check for satisfiability of P_M) can

be thought of as a constraint solving problem, where MONA finds the minimum sets of values for $\mathbb{V}_2 \cup \{\texttt{length}\}$ that satisfy the constraints expressed by P_M.

- At the top level, the satisfiability of P is checked against the pair $(\sigma, 0)$. Correspondingly, the function $trans(i, j, n, S, P')$ is instantiated with $i = j = 0$ (i.e., the current state is the initial state); $n = \texttt{length}$ (i.e., the final state is the last state of σ); $S = \{0, 1, \ldots, \texttt{length}\}$ (i.e., the interval is σ itself, and S must contain all the states of σ); and $P' = P$. Note that, P_S represents S as a quantified variable `S`, which is constrained to include all elements between 0 and `length`.

By way of example, Figures 7.5 and 7.6 show the translation of $P \triangleq \bigcirc A \,;\, \bigcirc B$ to a MONA formula P_M, and the final MONA specification P_S, respectively.

$\rightarrow$ $trans(0, 0, \texttt{length}, \texttt{S}, \bigcirc A \,;\, \bigcirc B)$
$\rightarrow$ `ex1 k:k in S & k>=0 & k<=length &` $trans(0, \texttt{k}, \bigcirc A)$ `&` $trans(\texttt{k}, \texttt{length}, \bigcirc B)$
$\rightarrow$ `ex1 k:k in S & k>=0 & k<=length &`
`0<k &` `ex1 k1:cons(S,0,k1)` `&` $trans(\texttt{k1}, \texttt{k}, A)$ `&`
`k<length &` `ex1 k2:cons(S,k,k2)` `&` $trans(\texttt{k2}, \texttt{length}, B)$
$\rightarrow$ $P_M \triangleq$
```
ex1 k:k in S & k>=0 & k<=length &
    0<k & ex1 k1:cons(S,0,k1) & k1<=k & k1 in A &
    k<length & ex1 k2:cons(S,k,k2) & k2<=length & k2 in B
```

Figure 7.5: Translating $P \triangleq \bigcirc A \,;\, \bigcirc B$ to a MONA formula, P_M

```
var1 length;
var2 A,B;
pred cons(var2 Set, var1 s1,s2) =
        s1 in Set & s2 in Set & s1<s2 &
        ~ ex1 s':s' in Set & s1<s' & s'<s2;
ex2 S: (all1 s:  s<=length <=> s in S) &
       ex1 k:k in S & k>=0 & k<=length &
       0<k & ex1 k1:cons(S,0,k1) & k1<=k & k1 in A &
       k<length & ex1 k2:cons(S,k,k2) & k2<=length & k2 in B;
```

Figure 7.6: A MONA specification P_S for $P \triangleq \bigcirc A \,;\, \bigcirc B$

Discussion: Soundness and Model Generation. The translation just proposed is sound in the sense that, for any PITL formula P, a semantically equivalent MONA specification P_S can be obtained. Therefore, we can check satisfiability of P by checking satisfiability of P_M (using MONA), the WS1S formula represented by P_S. Models and counterexamples for P can be derived directly from those obtained by MONA for P_M. This is formalised by Theorem 7.3.

THEOREM **7.3.** *Let P be a PITL formula and P_S the corresponding MONA specification. Then, the following holds,*

1. *P is satisfiable w.r.t. PITL semantics (i.e., there exists a σ s.t. $(\sigma, 0) \models P$) iff P_M is satisfiable w.r.t. WS1S semantics, where P_M is the WS1S formula expressed by P_S.*

2. *Let $\mathbb{V}^f$ be the set of free propositional variables occurring in P, and $\mathbb{V}_2^f$ the corresponding (w.r.t. the renaming bijection) free second-order variables declared in P_S. Let $[\![\ ]\!]_M : \mathbb{V}_2^f \rightarrow \mathcal{F}(\mathbb{N})$ denote the valuation for variables in $\mathbb{V}_2^f$ that MONA returns as a model (resp. counterexample) for P_M. Then, there exists an interval σ that is a model (resp. counterexample) for P, s.t. $[\![V]\!]_\sigma = [\![V]\!]_M$, for all $V \in \mathbb{V}_2^f$.*

Proof. First, note that P_M is satisfiable w.r.t. WS1S semantics if and only if there exists some n and $\mathcal{M}$ s.t. $(0, 0, n, \{0, 1, \ldots, n\}, \mathcal{M}) \models_W P$ is satisfiable w.r.t. WS1S semantics. This result follows from the fact that P_M is just expressing the semantic definition of

$$(0, 0, n, \{0, 1, \ldots, n\}, \mathcal{M}) \models_W P$$

in MONA (i.e., the difference is only in the syntax). By Theorem 7.2, for any PITL formula P there exists n and $\mathcal{M}$ s.t. $(0, 0, n, \{0, 1, \ldots, n\}, \mathcal{M}) \models_W P$, if and only if there exists σ s.t. $(\sigma, 0) \models P$, where $|\sigma| = n$ and $[\![\ ]\!]_\sigma = \mathcal{M}$.

It follows, then, that P_M is satisfiable w.r.t. WS1S semantics if and only if P is satisfiable w.r.t. PITL semantics. Following a similar reasoning, models (counterexamples) for P_M correspond to models (counterexamples) for P, where the behaviour of propositional variables in the corresponding interval is represented as finite sets of natural numbers (as explained in Section 7.3.1).[5] □

7.3.4 A Simpler WS1S-semantics for Formulae without Projection

The WS1S semantics for PITL can be refined by taking into account *sparse sets* on demand. Sparse sets represent intervals made of non-consecutive states; these are generated when formulae with projection operators are translated to MONA. PITL formulae that do not contain projection operators can be translated to simpler WS1S formulae, which contain less quantifiers and set operations. This results, in turn, in a more efficient satisfiability analysis.[6]

[5] Although, models for P_M are valuations for the free variables declared in it, which in turn correspond to the free propositional variables occurring in P. Thus, models for P_M are "incomplete", in the sense that they do not provide information about quantified variables in P.

[6] In fact, PITL2MONA performs the translation according to this simpler WS1S semantics (Section 7.4).

So far, PITL formulae were expressed in WS1S according to the satisfiability relation,

$$(i, j, n, S, \mathcal{M}) \models_W P$$

where S denotes some (but maybe not all) states in a given interval σ, between σ_i and σ_n, and including σ_j. This structure is necessary to deal with PITL formulae under projection. However, if the formula in question is not under the scope of a projection operator, the resulting WS1S formula is unnecessarily complicated. By way of example, consider the WS1S interpretation of $\bigcirc Q$, given below.

$$\begin{aligned}(i, j, n, S, \mathcal{M}) \models_W \bigcirc Q \quad \text{iff} \quad & j < n \text{ and exists } k \in \mathbb{N} \text{ s.t.} \\ & \mathit{cons}(S, j, k) \text{ and } (i, k, n, S, \mathcal{M}) \models_W Q\end{aligned}$$

Consider the case when $\bigcirc Q$ is a subformula of some top-level formula P, s.t. $\bigcirc Q$ is not under the scope of a projection operator. Thus the satisfiability of $\bigcirc Q$ is determined by a contiguous subinterval of σ,

$$({}^i\sigma^n, j - i) \models \bigcirc Q$$

Correspondingly, S includes all indices between i and n (i.e., $s \in S$ for all $i \leq s \leq n$).[7] Therefore S becomes redundant, because a WS1S interpretation of $\bigcirc Q$ can be given just in terms of the indices i, j and n (and the mapping $\mathcal{M}$).

The following satisfiability relation, $\models_s$, gives a simpler WS1S semantics to PITL formulae that are not under the scope of projection. We have omitted those cases where the WS1S formulae obtained from $\models_s$ and $\models_W$ (the original relation, shown in Section 7.3.2) are the same.[8]

$$\begin{aligned}
(i, j, n, \mathcal{M}) \models_s \bigcirc P \quad & \text{iff} \quad j < n \text{ and } (i, j+1, n, \mathcal{M}) \models_s P \\
(i, j, n, \mathcal{M}) \models_s P \ ; Q \quad & \text{iff} \quad \text{exists } k \in \mathbb{N},\ j \leq k \leq n \text{ s.t.} \\
& \quad (i, j, k, \mathcal{M}) \models_s P \text{ and } (i, k, n, \mathcal{M}) \models_s Q \\
(i, j, n, \mathcal{M}) \models_s P^* \quad & \text{iff} \quad j = n \text{ or exists } k_0, k_1, \ldots, k_m \in \mathbb{N} \text{ s.t.} \\
& \quad k_0 < k_1 < \ldots < k_m,\ k_0 = j,\ k_m = n, \text{ and} \\
& \quad \text{for all } r,\ 0 \leq r < m,\ (i, k_r, k_{r+1}, \mathcal{M}) \models_s P \\
(i, j, n, \mathcal{M}) \models_s P \textit{ until } Q \quad & \text{iff} \quad \text{exists } k \in \mathbb{N},\ j \leq k \leq n \text{ s.t.} \\
& \quad (i, k, n, \mathcal{M}) \models_s Q \text{ and} \\
& \quad \text{for all } r \in \mathbb{N},\ j \leq r < k,\ (i, r, n, \mathcal{M}) \models_s P
\end{aligned}$$

[7] This is the case, for example, when $P \triangleq \bigcirc Q$ is the top-level formula, and $i = j = 0$, $n = |\sigma|$ and $S = \{0, 1, \ldots, n\}$ are the initial settings.

[8] Note that, the semantics of P *proj* Q must necessarily be defined in terms of the relation $\models_W$, because Q, and all the subformulae of Q, are under the scope of a projection operator.

$(i,j,n,\mathcal{M}) \models_s P \; proj \; Q$ iff $j = n$ or exists S s.t.
$S = \{i, \ldots, k_0, k_1, \ldots, k_m\}$ and
for all $t,\ i \leq t \leq k_0$, $t \in S$ and
$k_0 < k_1 < \ldots < k_m$, $k_0 = j$, $k_m = n$, and
for all $r,\ 0 \leq r < m$,
$(i, k_r, k_{r+1}, \mathcal{M}) \models_s P$ and $(i,j,n,S,\mathcal{M}) \models_W Q$

$(i,j,n,\mathcal{M}) \models_s P \ \widetilde{;}\ Q$ iff exists $k \in \mathbb{N},\ i \leq k \leq j$ s.t.
$(i,k,n,\mathcal{M}) \models_s P$ and $(k,j,n,\mathcal{M}) \models_s Q$

$(i,j,n,\mathcal{M}) \models_s P \; since \; Q$ iff exists $k \in \mathbb{N},\ i \leq k \leq j$ s.t.
$(i,k,n,\mathcal{M}) \models_s Q$ and
for all $r \in \mathbb{N},\ k < r \leq j$, $(i,r,n,\mathcal{M}) \models_s P$

$(i,j,n,\mathcal{M}) \models_s \ominus P$ iff $i < j$ and $(i, j-1, n, \mathcal{M}) \models_s P$

P	$simple(i,j,n,P)$
$\bigcirc P$	j `<` n `&` $simple(i, j+1, n, P)$
$P\,;Q$	`ex1 k:k >=` j `& k <=` n `&` $simple(i,j,$`k`$,P)$ `&` $simple(i,$`k`$,n,Q)$
P^*	`ex2 K:min K =` j `& max K =` n `&` `all1 k1,k2:cons(K,k1,k2) =>` $simple(i,$`k1,k2`$,P)$
P *until* Q	`ex1 k:`j `<= k & k <=` n `&` $simple(i,$`k`$,n,Q)$ `&` `all1 k1:k1 >=` i `& k1 < k =>` $simple(i,$`k1`$,n,P)$

P *proj* Q	`ex2 K:min K =` j `& max K =` n `&`
	`all1 k1,k2:cons(K,k1,k2) =>` $simple(i,$`k1,k2`$,P)$ `&`
	$trans(i,j,n,$`K`$,Q)$

$P \ \widetilde{;}\ Q$	`ex1 k:`i `<= k & k <=` j `&` $simple(i,$`k`$,n,P)$ `&` $simple($`k`$,j,n,Q)$
P *since* Q	`ex1 k:`i `<= k & k <=` j `&` $simple(i,$`k`$,n,Q)$ `&` `all1 r:k < r & r <=` j `=>` $simple(i,$`r`$,n,P)$
$\ominus P$	i `<` j `&` $simple(i, j-1, n, P)$

Table 7.2: Translating PITL Formulae According to Simpler WS1S-semantics

Table 7.2 shows the MONA formulae obtained according to the simpler semantics (other PITL operators are translated as in Table 7.1). The translation is given in terms of a function, *simple()*. The translation of P *proj* Q is given in terms of the general function, *trans()* (Table 7.1).

7.4 PITL2MONA

PITL2MONA implements the translation from PITL formulae to MONA specifications, according to functions *trans()* and *simple()* (as explained in Sections 7.3.3 and 7.3.4). Given a source PITL formula, P, the translation will produce the simpler WS1S formulae for every subformula

P' that is not under projection (i.e., according to Table 7.2); otherwise, it will produce equivalent WS1S formulae according to Table 7.1. It can be seen, from Tables 7.1 and 7.2, that this translation produces a WS1S formula that is linear in the size of the source PITL formula. PITL2MONA outputs a single MONA program, which is semantically equivalent to the source PITL formula. In this way, PITL2MONA is a front-end to MONA, which is used as a decision procedure for PITL.

PITL2MONA supports the full PITL language considered in this chapter, plus a wealth of other (derived) operators. Input and output are simply handled as ASCII files. We used GNU Flex 2.5.4 and GNU Bison 1.875 to automatically generate the lexical analyser and parser, respectively. C code, which performs the syntactic translation from PITL to MONA, is embedded in the Bison grammar rules.

This section shows two different PITL specifications for the well-known Dining-philosophers problem, and a specification for the Symphony Problem (adapted from the Beethoven Problem presented in [41]). Our intention is to exercise the expressiveness of PITL as a requirements and specification language. Later, Section 7.4.3 benefits from these (and other) examples to compare the performance of PITL2MONA to that of LITE [98, 99], the other known implementation of a decision procedure for PITL.

7.4.1 The Dining Philosophers Problem

Consider the following PITL specification of the well-known Dining Philosophers problem. There are n philosophers sitting at a circular table, sharing n chopsticks. Philosophers spend most of the time thinking, but now and then, they become hungry and try to pick up the chopsticks and eat. Intervals represent the sequence of states philosophers may go through (although, because PITL is a finite time logic, only finite behaviour can be modelled).

For any $i,\ 1 \leq i \leq n$, the propositional variable Hungry_i denotes that the i-th philosopher is hungry and will attempt to eat (the i-th philosopher sits between the $(i+1)$-th, at his left, and the $(i-1)$-th, at his right). The variables Left_i and Right_i denote that the i-th philosopher is currently holding the left or right chopstick, respectively. The behaviour of philosophers is given by the formula

$$\mathit{Beh} \triangleq \bigwedge_{j=1}^{7} P_j$$

where[9]

[9]Throughout this section, $+$ and $-$ operations on indices are interpreted modulo n.

$$P_1 \triangleq \bigwedge_{i=1}^{n} \Diamond \mathit{Hungry}_i$$

$$P_2 \triangleq \bigwedge_{i=1}^{n} \Box(\mathit{Hungry}_i \Rightarrow (\neg \mathit{Left}_i \Rightarrow \mathit{Right}_{i+1}) \wedge (\neg \mathit{Right}_i \Rightarrow \mathit{Left}_{i-1}))$$

$$P_3 \triangleq \bigwedge_{i=1}^{n} \Box(\mathit{Pick}_i \Rightarrow \mathit{Hungry}_i)$$

$$P_4 \triangleq \bigwedge_{i=1}^{n} \Box\neg(\mathit{Left}_i \wedge \mathit{Right}_{i+1})$$

$$P_5 \triangleq \bigwedge_{i=1}^{n} \boxed{\mathrm{m}}((\mathit{Left}_i \wedge \neg \mathit{Right}_i \Rightarrow \bigcirc \mathit{Left}_i) \wedge (\mathit{Right}_i \wedge \neg \mathit{Left}_i \Rightarrow \bigcirc \mathit{Right}_i))$$

$$P_6 \triangleq \bigwedge_{i=1}^{n} \boxed{\mathrm{m}}(\mathit{Hungry}_i \wedge \neg \mathit{Eating}_i \Rightarrow \bigcirc \mathit{Hungry}_i)$$

$$P_7 \triangleq \bigwedge_{i=1}^{n} \Box(\mathit{Eating}_i \Rightarrow \Diamond \neg \mathit{Hungry}_i)$$

and

$$\begin{aligned} \mathit{Pick}_i &\triangleq \mathit{Left}_i \vee \mathit{Right}_i \\ \mathit{Eating}_i &\triangleq \mathit{Left}_i \wedge \mathit{Right}_i \end{aligned}$$

The behaviour of philosophers follows a few simple rules. Eventually, philosophers become hungry (P_1). When they do, they will try to pick up the chopsticks (P_2) and eat. Philosophers that are not hungry will not attempt to pick up the chopsticks, though (P_3). Obviously, chopsticks cannot be picked up by more than one philosopher at a time (P_4). Once a philosopher has picked up a chopstick, he does not release it until he manages to eat (P_5). Philosophers remain hungry until they eat (P_6). Philosophers that manage to eat become satiated at some point (P_7).

We might be interested in verifying a number of correctness properties. For example, the satisfiability of Prop_1,

$$\mathit{Prop}_1 \triangleq \mathit{Beh} \wedge \mathit{more} \wedge \bigwedge_{i=1}^{n} \Diamond \mathit{Eating}_i$$

determines whether the formalisation is sound, in the sense that it characterises at least one behaviour in which all philosophers eventually eat. The conjunct *more* is added to evaluate the correctness properties on intervals of a sufficiently "interesting" length (empty intervals are trivial models for $\boxed{\mathrm{m}}$).

Provided that $Prop_1$ is satisfiable, it makes sense to ensure that, in every possible behaviour, all philosophers get their chance to eat. This is done by checking the validity[10] of $Prop_2$,

$$Prop_2 \triangleq Beh \Rightarrow \bigwedge_{i=1}^{n} \Diamond Eating_i$$

PITL2MONA can be used to translate these formulae into equivalent MONA specifications. Then, MONA can be used to analyse the corresponding specifications. Intervals denoting models or counterexamples are returned as valuations of second-order variables. For the formulae $Prop_1$ and $Prop_2$, PITL2MONA produces a MONA specification that declares one first-order variable, `length` (whose value denotes the length of the model found), plus one second-order variable for every $Hungry_i$, $Left_i$ and $Right_i$ propositional variables occurring in the source PITL formula. Consider, for example, the satisfiability analysis of $Prop_1$ for 2 philosophers. MONA returns the model,

```
length = 2
Hungry1 = {1}   Left1 = {1}   Right1 = {1}
Hungry2 = {0}   Left2 = {0}   Right2 = {0}
```

This model describes a finite behaviour where the second philosopher becomes hungry in the state 0 (i.e., `Hungry2 = {0}`) and picks up both chopsticks immediately (i.e., `Left2 = {0}` and `Right2 = {0}`). He then eats, and gives the chopsticks to the other philosopher in state 1. The first philosopher eats in state 1 and puts the chopsticks back on the table in state 2.

However, some models of *Beh* represent deadlocks, i.e. situations where every philosopher is holding his left or right chopstick, but not both. Because philosophers never release a chopstick until they eat (P_5), a circular wait occurs that prevents every philosopher from eventually eating. Consequently, $Prop_2$ is satisfiable, but not valid.

Consider again a simple setting with 2 philosophers. When MONA checks the validity of $Prop_2$, it returns a deadlock as a counterexample:

```
length = 0
Hungry1 = {0}   Left1 = {}   Right1 = {0}
Hungry2 = {0}   Left2 = {}   Right2 = {0}
```

This interval represents a behaviour where both philosophers become hungry in state 0, and then each one of them picks up its right chopstick. They cannot eat, because each philosopher holds the chopstick the other one needs. None of them is willing to put the chopstick on the table either; formula P_5 has made them "selfish", forcing the philosophers to hold onto their chopsticks until they manage to eat.

[10] Checking *Beh* for satisfiability first ensures that the implication is not vacuously true due to an inconsistent antecedent.

One way to break this circular wait is to force an order among philosophers, such that the i-th philosopher cannot pick his chopsticks if the $(i+1)$-th philosopher is currently holding any of his. The following formula, Beh_{free}, constrains the behaviour Beh appropriately to ensure deadlock freedom. MONA confirms that is indeed the case, by checking that $Prop_3$ is valid.

$$Beh_{free} \triangleq Beh \wedge \bigwedge_{i=1}^{n} \Box(Pick_i \Rightarrow \neg\, Pick_{i+1})$$

$$Prop_3 \triangleq Beh_{free} \Rightarrow \bigwedge_{i=1}^{n} \Diamond Eating_i$$

Next, we give a different PITL specification for the Dining Philosophers problem, which benefits from the chop operator.[11] Again, we consider n philosophers and use the propositional variables $Left_i$ and $Right_i$ to represent the state of the i-th philosopher, but this time we do not use an explicit variable $Hungry_i$. We also define:

$$\begin{aligned} Thinking_i &\triangleq \neg\,(Left_i \vee Right_i) \\ Pick_i &\triangleq Left_i \vee Right_i \\ Eating_i &\triangleq Left_i \wedge Right_i \\ Put_i &\triangleq \neg\,(Left_i \wedge Right_i) \end{aligned}$$

In this specification, the behaviour of philosophers is described as cycles; philosophers think for awhile, then become hungry, pick up the chopsticks and eat, and finally they put the chopsticks back on the table and continue thinking. This behaviour is expressed by the PITL formula Beh',

$$Beh' \triangleq more \wedge \bigwedge_{i=1}^{n} Phil_i{}^{*} \wedge \Box\neg\,(Left_i \wedge Right_{i+1})$$

where

$$Phil_i \triangleq (more \wedge \boxed{\mathrm{m}}\, Thinking_i)\,;(more \wedge \boxed{\mathrm{m}}\, Pick_i)\,;(more \wedge \boxed{\mathrm{m}}\, Eating_i)\,;\Box Put_i$$

The use of *more* prevents empty models for $\boxed{\mathrm{m}}$ (in $Phil_i$) and chop-star (in Beh'). Note that, models of $Phil_i$ represent only situations where the i-th philosopher is able to eat. Correspondingly, models of Beh' represent only finite behaviours where all philosophers eat. Indeed, MONA confirms that Beh' is satisfiable, and that formula $Prop_4$ is valid:

$$Prop_4 \triangleq Beh' \Rightarrow \bigwedge_{i=1}^{n} \Diamond Eating_i$$

[11] This is inspired by one of the test examples given by Kono for his tool LITE.

This is in contrast with the previous specification of the problem, *Beh*, where soundness (i.e., that there is at least a model for *Beh* where all philosophers eat) had to be established explicitly by the formula $Prop_1$. On the other hand, the formula *Beh'* cannot represent the deadlocks that are implicit in *Beh*. Frequently, the dining philosopher problem is used to reason about situations where processes compete for some shared resources, and where circular waits may occur. Then, a specification such as *Beh'* would not be desirable, because it cannot represent certain behaviours of interest.

7.4.2 The Symphony Problem

Consider the synchronous presentation of an audio and a video stream. The audio stream comprises n movements of a given symphony, and the video stream shows the title (a still image) of every movement when this is played. Each movement M_i ($1 \leq i \leq n$) has a particular length, l_i. The movements are played in sequence, with a gap of g_m time units between consecutive movements. Each title T_i is repeatedly displayed throughout the corresponding movement, for s_t time units, with a gap of g_t time units between each repetition. The overall presentation results from composing the audio and video streams in parallel. The problem is to determine whether slowing down each stream before composition is equivalent to slowing down the composite stream.

In the following specification we use propositional variables M_i and T_i to denote that the i-th movement is being played and the i-th title is being shown, respectively. The multimedia stream can be expressed by the PITL formula *MMS*,

$$MMS \triangleq Synch_1 \,;\, Gap_1 \,;\, Synch_2 \,;\, \ldots \,;\, Gap_{n-1} \,;\, Synch_n \;\wedge\; FrameM \wedge FrameT$$

where ($1 \leq i \leq n$, $1 \leq j < n$)

$$\begin{array}{lcl}
PlayM_i & \triangleq & (\Box M_i) \wedge len(l_i) \\
ShowT_i & \triangleq & (\Box T_i) \wedge len(s_t) \\
Max_i & \triangleq & empty \;\vee (len_{<}(g_t) \wedge \bigcirc\Box\neg\, T_i) \;\vee (len(g_t) \wedge \bigcirc \boxed{\mathrm{m}} \neg\, T_i) \,;((\Box T_i) \wedge len_{<}(s_t)) \\
RepeatT_i & \triangleq & ShowT_i \,;((len(g_t) \wedge \bigcirc \boxed{\mathrm{m}} \neg\, T_i) \,;ShowT_i)^* \,;Max_i \\
Synch_i & \triangleq & PlayM_i \wedge RepeatT_i \\
Gap_j & \triangleq & len(g_m) \wedge \bigcirc \boxed{\mathrm{m}} \neg\, (M_j \wedge M_{j+1}) \\
FrameM & \triangleq & (\Box M_1 \,;\, \bigcirc\Box\neg\, M_1) \;\wedge \\
 & & (halt(M_2) \,;\Box M_2 \,;\, \bigcirc\Box\neg\, M_2) \;\wedge \ldots \;\wedge\; (halt(M_n) \,;\Box M_n)
\end{array}$$

$$FrameT \triangleq \bigwedge_{i=1}^{n} \Box(T_i \Rightarrow M_i)$$

The formulae $PlayM_i$ and $ShowT_i$ define the duration of movements and titles, respectively. $RepeatT_i$ represents the repetitive display of a title, with a gap of g_t time units between consecutive repetitions. Max_i forces a maximal number of iterations for chop-star in $RepeatT_i$, so that titles are repeatedly displayed during the movement, for as long as possible. This is achieved by making the interval for $RepeatT_i$ finish either with an exact number of title iterations, or with a gap less than g_t, in which case there is no time for a new iteration, or with a gap of g_t time units followed by a short period (less than s_t time units), where the title is displayed until the end. After a maximum number of iterations, the length of the remaining interval must necessarily be less than $g_t + s_t$ time units.

The synchronisation between titles and movements is modelled by $Synch_i$. The formula Gap_j models the gap between consecutive movements, $Synch_j$ and $Synch_{j+1}$. $FrameM$ guarantees that movements are played only once during the stream. Similarly, $FrameT$ ensures that titles are shown only during the corresponding movement. We represent the slowing down of the presentation by placing delays between consecutive states in the original model. Figure 7.7 illustrates the concept and its natural PITL interpretation using the projection operator.

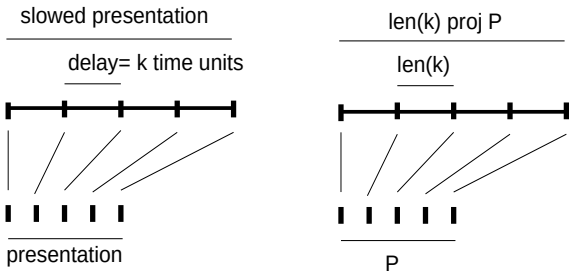


Figure 7.7: Slowing Down a Media Item

PITL2MONA and MONA were used to prove the following equivalence,

$$Prop_5 \triangleq MMS' \equiv MMS''$$

where MMS' denotes the multimedia stream resulting from composing the slower audio and video streams, and MMS'' denotes the stream resulting from slowing down both streams after they are synchronised. These PITL formulae are defined as follows ($k \in \mathbb{N}$ is the delay factor).

$$\begin{aligned}
MMS' &\triangleq Synch'_1 ; Gap_1 ; Synch'_2 ; \ldots ; Gap_{n-1} ; Synch'_n \ \wedge \ FrameM \wedge FrameT \\
MMS'' &\triangleq Synch''_1 ; Gap_1 ; Synch''_2 ; \ldots ; Gap_{n-1} ; Synch''_n \ \wedge \ FrameM \wedge FrameT \\
Synch'_i &\triangleq (len(k) \ proj \ PlayM_i) \wedge (len(k) \ proj \ RepeatT_i) \\
Synch''_i &\triangleq len(k) \ proj \ (PlayM_i \wedge RepeatT_i)
\end{aligned}$$

7.4.3 PITL2MONA vs. LITE

This section compares PITL2MONA with LITE [98, 99]. LITE is an implementation of a tableau-based decision procedure for a propositional subset of ITL. Tableau generation is based on PITL normal forms, and works along the lines of that of Bowman and Thompson [49].

LITE supports chop, chop-star, projection, quantification over propositional variables and a number of high-level, imperative-like iteration operators. Neither until or past operators are supported. LITE also accepts PITL formulae where terms are expressed as finite state automata, and returns PITL models/counterexamples in the form of automata. Compared with an automata-based decision procedure such as MONA, tableau-based implementations are likely to suffer from the overhead associated with expansion rules and the generation of normal forms.[12] Moreover, MONA includes a number of optimisations that allow most formulae to be decided with reasonable time and memory requirements [97].

Table 7.3 shows the performance of both tools for a number of parameterised problems.[13] Table 7.4 shows a characterisation of the complexity of the PITL specifications, with respect to the following parameters.

1. Maximum amount of memory consumed by MONA to analyse the formula (in MB). This memory requirement corresponds to the biggest automaton generated by MONA during the analysis of the source formula.

2. Number of free variables. Free variables in the PITL formula result in a similar number of free variables in the corresponding MONA formula (plus the `length` variable, which encodes the models length), which in turn determines the size of the automaton alphabet and its transition function.

3. Maximum nesting depth of quantifiers of the same kind (*q-depth*), and maximum nesting depth of alternating quantifiers ($\exists\forall$-*depth*). The occurrence of quantifiers in the formula (and particularly, the nesting of existential and universal quantifiers [114]) is likely to bring the worst-case complexity, as the translation to automata requires determinisation, which may result in an exponential number of states.

The n-Dining philosophers (1) and (2) correspond to formulae $Prop_3$ and Beh', respectively. The n-movement Symphony problem corresponds to formula $Prop_5$, with the following parameters: movement length = 10 (l_i), title length = 2 (s_t), movement gap = 3 (g_m), title gap = 3 (g_t), and delay factor = 2 (for projection). The n-floor Lift controller problem is parameterised by

[12] Although, LITE uses Binary Subterm Trees [98] to compactly store normal forms.

[13] The symbol "-" denotes that the analysis ran out of memory, or could not finish in a reasonable time. Only execution time is shown, because LITE does not output memory consumption figures.

the number of floors the lift serves; the PITL specification is adapted from that proposed by Pandya for his tool DCVALID. We run these examples on a Sun v480 station, with 2x900MHZ US3 processors, and 4GB RAM (SICStus Prolog v. 3.10.1. was used for LITE).

Table 7.3 suggests that PITL2MONA is much more efficient than LITE. PITL2MONA could efficiently deal with problems that LITE could not verify, either because the analysis exhausted the available memory or required unreasonable amounts of time (e.g., 7-Dining philosophers (1) and (2), 7- and 10-movement Symphony, and 5- and 10-Lift controller).

Example	n	*PITL2MONA*	*LITE*
n-Dining philosophers (1)	5	1.13 sec	2 hours
	7	28.62 sec	-
	10	(BDD too large)	-
n-Dining philosophers (2)	5	0.17 sec	7 min
	7	2.07 sec	-
	10	(BDD too large)	-
n-movement Symphony	7	7.07 sec	-
	10	1 min, 10 sec	-
n-floor Lift controller	5	3	-
	10	1 min, 50 sec	-

Table 7.3: PITL2MONA and LITE: Comparing Execution Times

Of all specifications tested, 10-Dining philosophers (1) and (2) could not be verified by PITL2MONA (BDDs grew too large for MONA to handle them). Although we have not experimented with different variable orderings, we suspect that the resulting MONA specifications for the Dining philosophers problem are suffering from a bad ordering.

For example, MONA required much more time and memory to solve 7-Dining philosophers (1) than 7-movement Symphony (28.62 against 7.07 sec, and 28 against 4MB, respectively), although the latter specification contains a deeper nesting of quantifiers and alternating quantifiers, as shown in Table 7.4 (quantifier alternation brings non-elementary complexity in WS1S formulae). Note that, the MONA specification for 7-Dining philosophers (1) contains more variables than that of 7-movement Symphony (22 against 15 variables, respectively).

We can also conclude that the number of variables, on its own, cannot be used as an indicator of performance. For example, the MONA specification of 10-Lift controller contains almost twice as many variables than that of 10-Dining philosophers (1). However, MONA was able to solve the former in less than 2 min, while the latter could not be solved at all due to state explosion.

Example	n	*Max. Mem. (MB)*	*Free Var.*	*q-depth*	$\exists\forall$*-depth*
n-Dining philosophers (1)	5	1	16	1	0
	7	28	22	1	0
	10	(BDD too large)	31	1	0
n-Dining philosophers (2)	5	1	11	6	4
	7	3	15	6	4
	10	(BDD too large)	21	6	4
n-movement Symphony	7	1	15	24	4
	10	4	21	29	4
n-floor Lift controller	5	1	28	4	3
	10	87	53	4	3

Table 7.4: The Complexity of MONA Specifications (From PITL Formulae)

Discussion: An analysis of the translation from PITL to WS1S formulae (Section 7.3) reveals those PITL formulae that may exhibit the logic's non-elementary complexity. In fact, these formulae must include a combination of operators that eventually result in WS1S formulae with alternating quantifiers. The following are some examples of such combinations; (a) alternated nesting of chop (chop in the past) and negation, (b) nesting of chop-star (until, *since*, projection) and (c) alternated nesting of *next* (*previous*) and negation under the scope of projection.

7.5 Discussion: Modelling Real-time Systems with PITL

Consider the formula $RepeatT_i$ in the specification of the Symphony Problem (Section 7.4.2). This formula uses chop, chop-star and variants of the *len* operator to express sequence, iteration and quantitative time constraints. This shows that PITL can naturally (and succinctly) represent complex specifications: titles must be played for as long as possible, and $RepeatT_i$ achieves this behaviour independently of the movement's duration.

In addition, $RepeatT_i$ is a good example of how PITL facilitates a constraint-oriented style of specification. The simple addition of Max_i (as the last element of the sequence) reduces the non-determinism inherent to chop-star. Without Max_i, we obtain a more abstract specification where titles are displayed at the right time, but where we do not constrain how often the titles will be displayed.

Similarly, in $Synch'_i$ and $Synch''_i$, *len* and projection describe (quite intuitively) the slowing down of the presentation (also see [76] for other uses of projection). The equivalence of two seemingly different specifications (a complex requirement to check) was simply represented by $Prop_5$. This is one benefit of describing specifications and requirements in the same notation.

These examples should convince the reader of some of the advantages of PITL with respect to timed automata and model-checkers; e.g., it would be arduous (to say the least) to address the specification and verification of the Symphony Problem in Uppaal or Kronos. On the other

hand, the Symphony Problem also reveals limitations of PITL as a specification language. The framing of variables is one these limitations: in PITL, the values of variables are not preserved from one state to the next, and this must be modelled explicitly [64]. For instance, although the intention is to represent a sequence of movements that occur only once, the formula *FrameM* must use chop to express sequence, but also *halt* and negation to enforce single occurrences.

Other limitations of PITL stem from the time model, which is discrete and finite. In this sense, PITL is less expressive than Timed Automata and model-checkers, and the application of PITL to reactive systems is not straightforward (e.g., consider the representation of ongoing behaviour, such as liveness properties). For instance, this limitation is evident in the Dining Philosophers Problem (Section 7.4.1), where we had to bound the length of behaviours. PITL with infinite-time would solve some of these problems, but a translation to WS1S (and mechanisation with MONA) does not seem possible: WS1S deals only with finite subsets. Similarly, discrete time is inherent to the semantics of intervals in PITL (whether we consider a finite or an infinite-time interpretation).

7.6 Related Work

Duan presents a projection operator in [64], which he uses to reason about different time granularities and synchronisation of parallel processes (in the context of temporal logic programming). Duan's *projection* and *chop in the past* behave differently from our operators (*proj*) and ($\widetilde{;}$); nonetheless, Bowman et al. show that Duan's operators can be derived from (*proj*) and ($\widetilde{;}$) [40]. Bowman et al. [40, 41, 37, 42] use projection to describe the slowing down of multimedia presentations (Section 7.4 illustrates such application), and for describing goals for cognitive behaviour in the domain of multi-modal human computer interaction. Hale [76] uses projection (e.g., to model behaviour at different time scales, and exception-handling) in the specification of a lift controller.

The usefulness of past operators has already been demonstrated in the context of LTL [105]. Some specifications can be more naturally modelled in terms of past operators [106], thus making the logic a more natural notation. Moreover, translating a formula with past operators, to one without them, may result in an exponential blow-up [113]. Past operators facilitate, as well, compositional verification, where it is convenient to reason about previous computations [10, 136]. The benefits of past operators can also be observed on PITL specifications. For example, Duan [64] used past operators to semantically characterise assignment in temporal logic programs with framed variables, and Bowman et al. [41] used past operators to facilitate the description of multimedia documents.

There has been substantial research on decision procedures for ITL (and subsets) [48, 49, 128, 127, 126]. Bowman and Thompson [49] present a tableau-based decision procedure and a complete axiomatisation for propositional ITL, which includes projection. The tableau is based on normal forms for ITL formulae [122, 99, 64] and a graph-traversal algorithm to decide satisfiability. Bowman and Thompson [49] also present substitutivity results for chop and projection, which facilitate compositional reasoning.

Moszkowski [123] gives an axiomatisation for finite-time ITL (propositional and first order), which is suitable for compositional reasoning. This work is extended to consider projection and infinite time in [124], and further investigated in [125]. Based on the assumption/commitment approach to compositional proofs, Moszkowski shows how proofs of ITL specifications (including safety and liveness properties, and equivalence of specifications) can be decomposed into simpler tasks. A key aspect of this approach is to restrict the assumptions and commitments to certain ITL fixpoints.

In [127, 128], Moszkowski presents a complete axiomatisation for a subset of ITL with variables in finite domains and quantification, both in finite and infinite time frameworks, based on automata. Recently, Moszkowski [126] presents another axiomatisation for a propositional subset of ITL, without quantification or projection. In this work, a proof of completeness is given in terms of Fusion Logic (FL) and Propositional LTL (PLTL) for finite time (FL is a propositional subset of ITL where the use of chop, and negation under the scope of chop, are restricted).

Moszkowski [119] describes a BDD-based decision procedure for finite time FL. The decision procedure is implemented in Lisp and uses PerlDD, a Perl interface to CUDD[14] (a popular BDD-package). The decision procedure is inspired by new interval-based techniques devised by Moszkowski to analyse PLTL. The current implementation does not include quantification, projection or past operators.

We have compared the performance of Moszkowski's tool with PITL2MONA, in a number of examples. As expected, specifications are easier to obtain and understand in PITL. However, Moszkowski's tool was able to deal with bigger specifications (in the number of free variables). Regarding execution time, neither tool seems to (consistently) outperform the other. Although we have not investigated this issue in depth, we think Moszkowski's tool benefits from ordering heuristics featured by CUDD. As we discussed in Chapter 6, the efficiency of BDDs is highly dependent on good variable orderings. In contrast, MONA does not provide any heuristics to control this ordering, and so the performance of PITL2MONA depends on the order in which propositional variables occur in the PITL formula.

[14] `http://vlsi.colorado.edu/ fabio`

Moszkowski [122], Hale [75], Kono [98, 99] and Duan [64] describe interpreters for ITL. Cau and Moszkowski developed an embedding of ITL in PVS [55].

Pandya's tool DCVALID [135, 134] implements a decision procedure for QDDC, a quantified discrete-time subset of Duration Calculus [172]. Duration Calculus can be considered an extension of ITL to model hybrid systems [171, 74]. DCVALID is implemented as a front-end that translates QDDC formulae to MONA specifications. QDDC is closely related to PITL; DCVALID supports forms of the chop and chop-star operators. It also implements quantification over propositional variables, and a duration operator that counts the number of times a given proposition is true in the interval (note, this operator is a distinctive feature of QDDC).

However, DCVALID does not currently support any form of temporal projection, and only a limited form of the past operator previous (it can be applied only over propositional variables).[15] DCVALID assists model checking of SMV, Verilog and Esterel designs w.r.t. CTL[DC], an extension of QDDC with CTL operators.

Pandya [133], and Chakravorty and Pandya [57], investigate the use of MONA-based decision procedures for (a decidable subset of) Interval Duration Logic (IDL), a dense-time subset of Duration Calculus. Sharma, Pandya and Chakravorty [147] compare different methods to check validity of IDL formulae. Li and Hung [104], and Thai and Hung [153], present algorithms to check a class of Duration Calculus formulae over timed automata.

A recent survey on interval temporal logics can be found in [74]. Industry is showing increasing interest in interval temporal logics. Verisity's *e* language,[16] a property specification language used in model checking tools, was influenced by ITL concepts [87]. IBM's model-checker RuleBase[17] also includes regular expression operators in its requirements language (PSL/Sugar) [14].

7.7 Conclusions

We used MONA to implement a decision procedure for PITL, a rich propositional subset of ITL. We developed a WS1S semantics for PITL formulae, and implemented this translation in our tool PITL2MONA. PITL2MONA translates PITL formulae to equivalent MONA specifications, such that models (and counterexamples) found by MONA correspond directly to those of the PITL formulae.

PITL2MONA+MONA can be considered the first efficient automata-based decision procedure for PITL (despite the logic's non-elementary complexity). We compared our implementation with LITE [99], a tableau-based decision procedure for a (similar) propositional subset of

[15] Pandya's work on QDDC, and our work on PITL described in this chapter, share a number of similarities. These theories were developed independently.

[16] `www.cadence.com/verisity`

[17] `www.haifa.il.ibm.com/projects/verification/Formal_Methods-Home/index.html`

ITL. Our implementation is considerably faster, and is able to deal with more complex specifications than LITE (e.g., LITE does not handle past operators).

Operators in PITL (such as *len*, chop, chop-star, projection, previous, since, etc.) facilitate the modelling of requirements that refer to sequences and iteration of events, and past computations. This makes PITL a more intuitive (and thus, reliable) notation than the requirements language of Uppaal and other real-time model-checkers (Chapter 2). In addition, we have discussed the strengths and weaknesses of PITL as a specification language (Section 7.5).

The translation from PITL formulae to automata (via MONA) can be considered a first step towards the development of model-checkers that use PITL as part of their requirements languages (similar to Pandya's extension of CTL with QDDC formulae [135]).

Our work may also contribute to the development of proof assistants for ITL; for instance, proofs that can be reduced to PITL formulae could be solved automatically by MONA. Work in this direction may benefit from Cau and Moszkowski's embedding of ITL in PVS [55], and from the integration of MONA with theorem provers [131, 11].

From another perspective, this chapter informs on the use of MONA and WS1S in real-time settings. WS1S proved to be a flexible and expressive notation, which allowed an intuitive encoding of PITL. MONA, with its many optimisations, proved to be an efficient decision procedure for PITL, despite the non-elementary complexity of this logic.

However, WS1S and MONA also present limitations. WS1S cannot represent infinite sets; thus, an encoding of PITL with infinite-time does not seem possible. Regarding MONA, performance may suffer from bad variable orderings, for which MONA does not implement heuristics. Although a non-elementary complexity cannot be avoided in the worst-case (this is inherent to PITL), ordering heuristics would improve the average performance. In addition, it would be interesting to develop a decision procedure for PITL based on other efficient BDD-packages, such as CUDD (where some heuristics are already implemented), and compare the performance with MONA.

We will conclude our study of WS1S and MONA in real-time settings in Chapter 8. Chapter 8 will present Discrete Timed Automata, a deductive framework for real-time systems that uses WS1S as the assertion language, and MONA as tool support for invariance proofs. Discrete Timed Automata address limitations of timed automata and model-checkers (which concern the support for unbounded data domains and prevention of timelocks), and presents advantages over PITL as a specification language (the notation adopts an infinite-time model, and is not affected by framing).

Chapter 8

Discrete Timed Automata

Abstract. We present Discrete Timed Automata, a deductive framework for infinite-state, discrete real-time systems. Discrete Timed Automata uses WS1S as the underlying assertion language, which allows MONA to assist invariance proofs. Another distinctive feature is the use of deadlines [33, 39] to express urgent actions, and the adoption of a semantics of urgency and synchronisation that guarantees time reactivity (absence of time-actionlocks) by construction.

This chapter concludes Part III, and our study of WS1S and MONA to address reliability issues of timed automata and model-checkers (Chapter 2). We show that WS1S and MONA facilitate the representation and verification of unbounded data structures, and so certain systems can be more faithfully specified than in Uppaal and other model-checkers. Furthermore, time reactivity makes well-timed specifications easier to obtain in Discrete Timed Automata than in most other notations for real-time systems (including Timed Automata). To complete our evaluation, we compare Discrete Timed Automata to PITL when used as a specification language (Chapter 7), and also point out certain limitations in the use of Discrete Timed Automata that arise due to the restrictive arithmetic of WS1S.

Organization. We introduce Discrete Timed Automata in Section 8.1, and give the formal syntax and semantics in Section 8.2. In Section 8.3 we discuss Basic Transition Systems and invariance proofs; this serves as background to explain the verification of Discrete Timed Automata specifications in Section 8.4. In Section 8.5 we discuss strengths and limitations of Discrete Timed Automata. We comment on related work in Section 8.6. In Section 8.7 we draw conclusions and suggest further research.

8.1 Introduction

Discrete Timed Automata is an infinite-state, discrete-time deductive framework for real-time systems. A discrete timed automaton (DTA) consists of a finite set of typed variables, whose valuations describe the state space, and a finite set of actions to describe state changes. DTAs use WS1S as the underlying assertion language; booleans, natural numbers and finite sets of natural numbers are all types supported in WS1S, and actions are defined by a number of WS1S formulae over the automata variables (we define Discrete Timed Automata formally in Section 8.2).

Actions have a *precondition* formula to denote the set of states where the action is enabled; a *deadline* formula to denote the set of states where the action is urgent, and an *effect* formula to denote a set of next states; the result of performing an action may yield a number of different states, in which case execution proceeds by choosing (non-deterministically) one of such states. Actions are assumed to occur instantaneously, and the current (discrete) time is represented by a special variable, $T \in \mathbb{N}$, which automata can read but not modify; T increments (implicitly) with the passage of time.

Deadlines in DTAs represent time progress conditions: for any action a with deadline d, time cannot pass in any state where d holds, until a is performed. In order to avoid reachable states (timelocks) where the deadline holds but no action is enabled, we expect specifications where *deadlines imply preconditions*. In this way, we encourage specifiers to write safer models (i.e., one where time-actionlocks cannot occur): *actions should not be made urgent unless they are enabled*. Nonetheless, in the spirit of Timed Automata with Deadlines (Section 3.4.1), we provide a semantics for deadlines that guarantees time reactivity, even in those specifications where deadlines may not imply preconditions (in which case, the deadline will be ignored).

Systems can be represented by a network of DTAs executing in parallel, where communication may take place via binary synchronisation and shared variables. Synchronisation is realised by tagging certain actions as either input or output actions, which can only be executed in matching pairs; all other actions (completed actions) are performed autonomously. Furthermore, composition preserves time reactivity: input and output actions may be considered urgent only if synchronisation is possible. Concurrency is achieved by interleaving; at any state in the execution of the network, either a completed action is performed, or synchronisation between components takes place, or time passes by some quantity (a delay).

We provide a mapping from DTAs to Basic Transition Systems (BTSs) [111], which will allow us to prove LTL safety properties [112]. Basic Transition Systems are a well-known computational model for untimed systems, with a notation similar to Discrete Timed Automata: a finite set of transitions (equivalently, actions) and variables, where transitions are described through preconditions and postconditions, and variables determine the state space. The invariance rule

[112] is used to assert the validity of a safety property $\Box\, p$, where p is a state formula (in our case, a WS1S assertion), provided p holds at the initial state and is preserved by every transition in the system (we introduce Basic Transition Systems and the invariance rule in Section 8.3).

For any DTA, we obtain a semantically equivalent BTS, where actions (preconditions and effects) and variables in the DTA are directly translated into transitions and variables of BTS. The passage of time (implicit in DTAs) is represented in the BTS by a transition *tick* (which increments the variable T), and deadlines are mapped to preconditions for the *tick* action. In this way, we obtain an untimed semantics for DTAs in terms of BTSs; moreover, because WS1S is the assertion language of both DTAs and BTSs, MONA can be used to validate the proof obligations generated by the invariance rule (Section 8.4 defines the mapping between BTSs and DTAs, and gives examples of invariance proofs).

Discrete Timed Automata address some limitations of timed automata and model-checkers, concerning data support (Section 2.4) and prevention of timelocks (Chapter 3). We will see that certain unbounded data structures can be naturally represented in WS1S, and that MONA is able to validate the complex assertions that usually arise when reasoning about such structures (e.g., assertions that involve quantification and set operations). In addition, Discrete Timed Automata are time reactive, hence well-timed specifications are easier to obtain than in most other notations for real-time systems. Discrete Timed Automata also overcomes some of the limitations of PITL as a specification language (Chapter 7). We elaborate on these and other strengths of Discrete Timed Automata in Section 8.5; and we also discuss limitations that arise due to the restrictive arithmetic of WS1S.

8.2 Discrete Timed Automata – Syntax and Semantics

This section introduces discrete timed automata formally. Before we discuss syntax and semantics, we illustrate the main elements of the theory with the following example.

Example: A DTA Specification of an Alarm Clock. Figure 8.1 shows a DTA specification of a simple alarm clock. The network is composed of two automata, *Clock* and *Alarm*. For every component, V_L is the set of local variables, Θ_L is the initialisation formula for variables in V_L, TL is the set of action labels, and $\mathcal{A}$ is the set of actions; *true*-preconditions, *false*-deadlines and effects which do not change the current valuation are omitted. Note the use of the time variable T; components can read T, but they cannot change its value (T is assumed to model the current global time of the system, and is therefore implicitly incremented by the passage of time). In this example there are no shared variables.

Network: $|\langle \mathit{Clock},\ \mathit{Alarm}\rangle$
V_S : $\emptyset$
Θ_S : true

DTA: *Clock*
V_L : $\{\, s, m, h \in \mathbb{N} \,\}$
Θ_L : $s = 0 \ \wedge\ m = 0 \ \wedge\ h = 0$
TL : $\{\, \mathit{tick},\ \mathit{minute},\ \mathit{reset},\ \mathit{hour},\ \mathit{sound\_alarm}! \,\}$
$\mathcal{A}$: *minute*
  prec : $T = s + 60 \ \wedge\ m < 60$
  deadline : *prec*
  eff : $m' = m + 1 \ \wedge\ s' = T$
hour
  prec : $m = 60 \ \wedge\ h < 24$
  deadline : *prec*
  eff : $h' = h + 1 \ \wedge\ m' = 0$
reset
  prec : $h = 24$
  deadline : *prec*
  eff : $h' = 0$
sound_alarm!
  prec : $h = 5 \ \wedge\ m = 30$
  deadline : *prec*

DTA: *Alarm*
V_L : $\{\, d \in \mathbb{N},\ \mathit{on} \in \mathbb{B} \,\}$
Θ_L : $d = 0 \ \wedge\ \neg\, \mathit{on}$
TL : $\{\, \mathit{sound\_alarm}?,\ \mathit{stop} \,\}$
$\mathcal{A}$: *sound_alarm?*
  eff : $\mathit{on}' \ \wedge\ d' = T$
stop
  prec : $\mathit{on} \ \wedge\ T = d + 15$
  deadline : *prec*
  eff : $\neg\, \mathit{on}' \ \wedge\ d' = 0$

Figure 8.1: A Simple Alarm Clock

The *Clock* component models a 24-hour clock, where the alarm is set to 5:30 am. The variable s denotes the number of seconds elapsed so far, while m and h denote, respectively, the current minute and hour (we assume a time granularity of 1 second). Actions *minute* and *hour* calculate the current minute and hour; actions *hour* and *reset* keep the values of m and h between 0 and 60, and 0 and 24, respectively (the value of a variable, say v, after an action has been performed, is denoted by its primed version, v'). To keep the specification simple, the value of s is allowed to increment without bound. All these actions are completed actions and can be performed autonomously; on the other hand, *sound_alarm!* is an output action, which synchronises with the input action *sound_alarm?* in *Alarm*. Once synchronisation happens, the alarm will ring for 15 seconds and then *stop*.

Note that all actions, save for *sound_alarm?*, are made urgent as soon as they are enabled: the set of states denoted by deadlines and preconditions are the same.[1] Local variables such as s and d, which we refer to as *time capture variables*, are used to represent time constraints. The value of a time capture variable denotes the last time an event of interest occurred. For example, in action *minute*, the term $T = s + 60$ in the precondition, and the term $s' = T$ in the effect, represent a time span of 60 seconds counting from the last occurrence of *minute*. In other words, and because *minute* is urgent as soon as it is enabled, *minute* is performed once every 60 seconds.

[1] We use the term *prec* in deadlines to denote the action's precondition formulae.

8.2.1 Syntax

Discrete Timed Automata. A discrete timed automaton is defined here as a (and possibly the only) component of a given network. Variables are assumed to be of any type supported in WS1S, and actions are defined in terms of WS1S formulae (therefore, they are expressible in MONA). We assume a set $\mathcal{F}_W$ of WS1S formulae, and (as for Timed Automata in Section 2.1.1) the sets of action labels $CAct$ (for completed actions), $HAct$ (for half actions), and $Act = CAct \cup HAct$.

A single DTA, A, is a tuple,

$$A = (V_L, \Theta_L, TL, \mathcal{A}, V_S, \Theta_S)$$

where

- V_L is a finite set of local variables;
- $\Theta_L(V_L)$ is a formula representing the initial valuation for variables in V_L;
- $TL \subseteq Act$ is a finite set of action labels;
- $\mathcal{A}$ is a finite set of actions (the structure of these actions will become clear shortly);
- V_S is a finite set of shared variables; and
- $\Theta_S(V_S)$ is a formula representing the initial valuation for variables in V_S.

Let us complete the definition of a component DTA. A variable (local or shared) is *rigid* if its value remains constant throughout executions, and *flexible* otherwise. Both V_S and Θ_S are declared (globally) in the network where A is placed. A can read and write variables in $V = V_L \cup V_S$, and can also read (but not write) the time variable $T \in \mathbb{N}$. We use $V^T = V \cup \{T\}$, and $V' = \{v' \mid v \in V\}$ to denote the primed version of variables in V. As is customary in assertional notations, we use primed variables to denote the valuations obtained after actions have been performed.

Actions in $\mathcal{A}$ are tuples (a, p, d, e), where $a \in TL$ is the action's label, and $p(V^T)$, $d(V^T)$ and $e(V^T \cup V')$ $(p, d, e \in \mathcal{F}_W)$ are WS1S formulae denoting the action's precondition (the set of states where the action may be performed), deadline (the set of states where the action must be performed urgently) and effect (the set of states that result when the action is performed), respectively. We use $W(e) \subseteq V$ to denote the set of all variables which are written by e (i.e., any variable $v \in V$ such that v' occurs free in e). Variables in $U(e) = V^T \setminus W(e)$ are assumed unchanged after the action is performed. As in timed automata with deadlines (Section 3.4.1), the deadline of every action must imply its precondition. Note that, left-closed intervals are naturally obtained in DTAs due to discrete time (e.g., $T > 1$ is equivalent to $T \geq 2$).

The set $\mathcal{A}$ is partitioned into three sets; completed ($COMP(\mathcal{A})$), input ($IN(\mathcal{A})$) and output actions ($OUT(\mathcal{A})$); i.e., $\mathcal{A} = COMP(\mathcal{A}) \cup IN(\mathcal{A}) \cup OUT(\mathcal{A})$. We allow actions in the same partition to share labels: this may facilitate the description of actions whose effects depend on the current state. As usual, we use labels in $\{\, a? \mid a? \in HAct \,\}$ to denote input actions, and labels in $\{\, a! \mid a! \in HAct \,\}$ to denote output actions. Two actions in different components, respectively labelled with *a?* and *a!* (for some $a \in CAct$), are referred to as *matching actions*.

Network of DTAs. A network of DTAs is a tuple $|A = (|\langle A_1, \ldots, A_n \rangle, V_S, \Theta_S)$, where $A_i = (V_L^i, \Theta_L^i, TL_i, \mathcal{A}_i, V_S, \Theta_S)$ is a DTA ($1 \leq i \leq n$). We assume the network is well-formed in the following sense: names of local variables and labels of completed actions are local to components, and the conjunction of effects of any two matching input/output actions must yield a consistent WS1S formula (e.g., to prevent inconsistent updates to shared variables).

Product Automaton. Let $|A = (|\langle A_1, \ldots, A_n \rangle, V_S, \Theta_S)$ be a network of DTAs, where $A_i = (V_L^i, \Theta_L^i, TL_i, \mathcal{A}_i, V_S, \Theta_S)$, for $1 \leq i \leq n$. The product automaton of $|A$ is a DTA, Π,

$$\Pi = (V_L, \Theta_L, TL, \mathcal{A}, \emptyset, true)$$

where

- $V_L = V_S \cup \bigcup_{i=1}^{n} V_L^i$
- $\Theta_L \triangleq \Theta_S \ \wedge \ \bigwedge_{i=1}^{n} \Theta_L^i$
- $TL = \{\, a \mid \exists\, p, d, e \in \mathcal{F}_W \,.\, (a, p, d, e) \in COMP(\mathcal{A}) \,\}$
- $COMP(\mathcal{A}) = \bigcup_{i=1}^{n} COMP(\mathcal{A}_i) \cup$
  $\{(a, p, d, e) \mid \exists\, i, j\, (1 \leq i \neq j \leq n) \,.\, \exists\, p_i, p_j, d_i, d_j, e_i, e_j \in \mathcal{F}_W \,.$
  $(a?, p_i, d_i, e_i) \in IN(\mathcal{A}_i) \ \wedge \ (a!, p_j, d_j, e_j) \in OUT(\mathcal{A}_j) \ \wedge$
  $(p \triangleq p_i \wedge p_j) \ \wedge \ (d \triangleq d_i \vee d_j) \ \wedge \ (e \triangleq e_i \wedge e_j) \,\}$
- $IN(\mathcal{A}) = OUT(\mathcal{A}) = \emptyset$

Note that, the shared variables in the network are made local to the product, and that all actions in the product automaton are completed: synchronisation yields one completed action from every two matching input/output actions in the components. In addition, we point out that the disjunction $(d_i \vee d_j)$ makes the synchronisation urgent as soon as one of the parties

is urgent; as we shall see in Section 8.2.2, the semantics of deadlines is such that urgency is enforced only if this synchronisation is possible (i.e., if both parties are enabled).

8.2.2 Semantics

We give here a discrete time, operational semantics for DTAs. This explains the behaviour of a single automaton, equivalently, the behaviour of a network with just one single component. The behaviour of a network with multiple components corresponds to that of the product automaton. By construction (and well-formedness of the network), the product automaton only contains completed actions and local variables. In what follows, we use $\mathbb{N}^+$ to denote the positive natural numbers, and $\models$ to denote satisfiability under WS1S semantics.

Let $A = (V_L, \Theta_L, TL, \mathcal{A}, \emptyset, \mathit{true})$ be a DTA where all actions are completed (i.e., $\mathcal{A} = \mathit{COMP}(\mathcal{A})$), and all variables are local to A ($V = V_L$). The semantics of A are given in terms of a (discrete) timed transition system $TS_A = (S, s_0, \mathit{Lab}, T_S)$, where,

- S is the set of states, which denotes all possible valuations of V^T. Then, $s \in S$ maps any variable $v \in V^T$ to a type-consistent value $s(v)$. Given $s \in S$, $\mathit{primed}(s)$ is the valuation that renames every variable in V^T to its primed version; i.e., $\mathit{primed}(s)$ is the valuation for $\{v' \mid v \in V^T\}$ such that $\mathit{primed}(s)(v') = s(v)$ for all $v \in V^T$. For any $\delta \in \mathbb{N}^+$, the state $s + \delta$ is defined such that $(s+\delta)(T) = s(T) + \delta$ and $(s+\delta)(v) = s(v)$ for all $v \in V$. The set of reachable states in the execution of A, $S_{reach} \subseteq S$, is the smallest set of states which includes the initial state s_0 and is closed under the transition relation T_S,

$$S_{reach} = \{s_0\} \cup \{s' \mid \exists\, \gamma \in \mathit{Lab},\, s \in S_{reach}.\, s \xRightarrow{\gamma} s' \in T_S\}$$

- s_0 is the initial state, s.t. $s_0 \models (\Theta_L \wedge T = 0)$.

- $\mathit{Lab} = TL \cup \mathbb{N}^+$ is the set of labels for transitions in T_S.

- $T_S \subseteq S \times \mathit{Lab} \times S$ is the set of labelled transitions. Transitions can be of one of two types: action transitions (*actions*), e.g. (s, a, s'), where $a \in \mathit{Act}$, or time transitions (*delays*), e.g. (s, δ, s'), where $\delta \in \mathbb{N}^+$, denoting the passage of δ time units. Transitions are denoted, $s \xRightarrow{\gamma} s'$, where $\gamma \in \mathit{Lab}$. The transition relation is defined by the following inference rules,

$$(D1)\quad \frac{(a,p,d,e) \in \mathcal{A} \qquad s_1 \cup \mathit{primed}(s_2) \models (p \wedge e \wedge \bigwedge_{u \in U(e)} u' = u)}{s_1 \xRightarrow{a} s_2}$$

$$(D2)\quad \frac{\delta \in \mathbb{N}^+ \quad \forall \delta' \in \mathbb{N}\,(\delta' < \delta).\, \nexists\, (a,p,d,e) \in \mathcal{A}.\, (s+\delta') \models (p \wedge d)}{s \xRightarrow{\delta} (s+\delta)}$$

We say that an action (a, p, d, e) is *enabled* in s if $s \models p$. Note that, rule (D2) only enforces urgency for enabled actions, and only allows a δ-delay if no enabled action becomes urgent anytime sooner (i.e., if no deadline is reached after any δ'-delay, with $\delta' < \delta$). As we have mentioned before, adopting such a semantics for urgent behaviour makes DTAs time reactive: from every reachable state, either time may pass or some action may be performed. Furthermore, time reactivity is preserved by synchronisation: by definition of product automaton, synchronisation yields a completed action where the preconditions of half actions are conjoined, and the deadlines are disjoined. Therefore, as the product automaton is also interpreted by rule (D2), time will stop as soon as either half action is urgent, but only if both half actions are enabled (i.e., only if synchronisation is possible at the current state).

DTA: *ZenoRuns*
V_L : $\{\, state \in \{0,1\},\ x \in \mathbb{N}\}$
Θ_L : $state = 0 \ \wedge \ x = 0$
TL : $\{\, a,\ b\,\}$
$\mathcal{A}$: a
$prec$: $state = 0$
$deadline$: $state = 0 \ \wedge \ T = 2$
eff : $x' = x + 1$
b
$prec$: $state = 0$
eff : $state' = 1$

DTA: *ZenoTimelocks*
V_L : $\{\, state \in \{0,1\},\ x \in \mathbb{N}\}$
Θ_L : $state = 0 \ \wedge \ x = 0$
TL : $\{\, a,\ b\,\}$
$\mathcal{A}$: a
$prec$: $state = 0$
$deadline$: $state = 0 \ \wedge \ T = 2$
eff : $x' = x + 1$
b
$prec$: $state = 0 \ \wedge \ T \geq 3$
eff : $state' = 1$

Figure 8.2: Zeno Runs and Zeno-timelocks in DTAs

Runs and Timelocks. The definition of runs (including divergent, convergent and Zeno runs) and timelocks (time-actionlocks and Zeno-timelocks) given in Chapters 2 and 3 are also valid in DTAs, modulo discrete delays and a (slightly) different structure for states. Therefore, we will not repeat these definitions here.

We have shown that DTAs are time reactive, and so is the composition of DTAs. However, Zeno runs and Zeno-timelocks may occur. By way of example, Figure 8.2(left) shows an automaton with two states and two actions, a and b, that are only enabled when $state = 0$. Action a increments the value of x by 1 every time it is executed, and it is urgent when $T = 2$. The automaton engages in Zeno runs whenever a is performed infinitely often; this is possible because action a may be performed arbitrarily fast. However, the automaton is free from timelocks, because action b can be executed at any time, leading to $state = 1$, where time can diverge. Note that a Zeno-timelock would occur if, for instance, we add a lower bound $T \geq 3$ to the precondition of b (see Figure 8.2, right). We will not study detection of Zeno runs and Zeno-timelocks (for DTAs) in this chapter, but we will offer some hints as to how this may be accomplished, later in Section 8.7.

8.2.3 Example: A DTA Model for the Multimedia Stream

Figure 8.3 shows a DTA specification for a discrete-time version of the multimedia stream example (see Section 2.1.3). Here, in addition, we consider a lossy channel and a throughput monitor (a similar example was analysed with Uppaal in [44]).

The *Source* automaton declares a variable $SourceState \in \{0, 1\}$ to represent the current control state of the automaton, and a time capture variable *t1* to keep track of the time elapsed between two consecutive transmissions. Control states denote that the automaton is ready to transmit the first packet, when $SourceState = 0$, or that it is ready to transmit the next packet in the sequence, when $SourceState = 1$. Transmissions are represented by two *sourceOut!* actions, one for the first packet, and the other one for any subsequent packet. Deadlines ensure that the first packet is transmitted urgently when $T = 0$, and that every subsequent packet is transmitted 50 ms after the previous one. This is achieved by setting $t1' = T$ when a packet is sent (i.e., resetting the value of *t1* to the current time), and sending the next packet when the difference between *t1* and the current time is 50 ms (as expressed by the deadline $T = t1 + 50$).

A similar analysis explains the behaviour of the other components. For example, the action *sinkIn!* in *Place1* is enabled when at least 80 ms have passed since a packet was received (i.e., since *sourceOut?* was performed), and must be performed before 90 ms have passed since then (note that the deadline in *sinkIn!* does not allow time to pass beyond $T = t4 + 90$, if *sinkIn!* is still enabled).

A *Monitor* component checks that the system's throughput does not fall below a threshold of 14 packets per second, and signals the outcome of such check in the (shared) boolean variable *error*. The (shared) variable x is used to count the number of packets received by the *Sink* in the current second. In this specification, we have adopted the following simple policy: the *error* signal persists for a period of 1 second, until is updated by the next check. In addition, we have assumed that execution continues even if the throughput falls below the threshold (a different policy could be, for instance, to specify exception-handling in the *Monitor* once the *error* occurs).

The channel is implemented by automata *Place1* and *Place2*, and may lose up to 4 packets per second. This is modelled using a shared variable $l \in \{0, \ldots, 4\}$ to count the number of packets lost in the current second, and an action *loss* in *Place1* (and *Place2*) that nondeterministically prevents *Place1* from delivering the current packet to the *Sink*. Note, l is reset by action *check* in *Monitor*, at the beginning of the current second ($T = t_m + 1000$).

Network: $|\langle \mathit{Source}, \mathit{Place1}, \mathit{Place2}, \mathit{Sink}, \mathit{Monitor}\rangle$
$V_S : \{\, x \in \mathbb{N},\ l \in \{0,\ldots,4\},\ \mathit{error} \in \mathbb{B}\,\}$
$\Theta_S : x = 0 \ \wedge\ l = 0 \ \wedge\ \neg\, \mathit{error}$

DTA: *Source*
$V_L : \{\, \mathit{SourceState} \in \{0,1\},\ \mathit{t1} \in \mathbb{N}\,\}$
$\Theta_L : \mathit{SourceState} = 0 \ \wedge\ \mathit{t1} = 0$
$TL : \{\, \mathit{SourceOut}!\,\}$
$\mathcal{A}$: *sourceOut!*
- *prec* : $\mathit{SourceState} = 0 \wedge T = 0$
- *deadline* : *prec*
- *eff* : $\mathit{SourceState}' = 1$

sourceOut!
- *prec* : $\mathit{SourceState} = 1 \wedge T = \mathit{t1} + 50$
- *deadline* : *prec*
- *eff* : $\mathit{t1}' = T$

DTA: *Sink*
$V_L : \{\, \mathit{SinkState} \in \{1,2\},\ \mathit{t2} \in \mathbb{N}\,\}$
$\Theta_L : \mathit{SinkState} = 1 \ \wedge\ \mathit{t2} = 0$
$TL : \{\, \mathit{sinkIn}?,\ \mathit{play}\,\}$
$\mathcal{A}$: *sinkIn?*
- *prec* : $\mathit{SinkState} = 1$
- *eff* : $\mathit{SinkState}' = 2 \wedge \mathit{t2}' = T \ \wedge\ x' = x + 1$

play
- *prec* : $\mathit{SinkState} = 2 \wedge T = \mathit{t2} + 5$
- *deadline* : *prec*
- *eff* : $\mathit{SinkState}' = 1$

DTA: *Place1*
$V_L : \{\, \mathit{Place1State} \in \{1,2\},\ \mathit{t4} \in \mathbb{N}\,\}$
$\Theta_L : \mathit{Place1State} = 1 \ \wedge\ \mathit{t4} = 0$
$TL : \{\, \mathit{sourceOut}?,\ \mathit{sinkIn}!\,\}$
$\mathcal{A}$: *sourceOut?*
- *prec* : $\mathit{Place1State} = 1$
- *eff* : $\mathit{Place1State}' = 2 \wedge \mathit{t4}' = T$

sinkIn!
- *prec* : $\mathit{Place1State} = 2 \wedge T > \mathit{t4} + 80$
- *deadline* : $\mathit{Place1State} = 2 \wedge T \geq \mathit{t4} + 90$
- *eff* : $\mathit{Place1State}' = 1$

loss
- *prec* : $\mathit{Place1State} = 2 \wedge l < 4$
- *deadline* : $\mathit{prec} \wedge T \geq \mathit{t4} + 90$
- *eff* : $l' = l + 1 \wedge \mathit{Place1State}' = 1$

DTA: *Place2*
$V_L : \{\, \mathit{Place2State} \in \{1,2\},\ \mathit{t3} \in \mathbb{N}\,\}$
$\Theta_L : \mathit{Place2State} = 1 \ \wedge\ \mathit{t3} = 0$
$TL : \{\, \mathit{sourceOut}?,\ \mathit{sinkIn}!\,\}$
$\mathcal{A}$: *sourceOut?*
- *prec* : $\mathit{Place21State} = 1$
- *eff* : $\mathit{Place2State}' = 2 \wedge \mathit{t3}' = T$

sinkIn!
- *prec* : $\mathit{Place2State} = 2 \wedge T > \mathit{t3} + 80$
- *deadline* : $\mathit{Place2State} = 2 \wedge T \geq \mathit{t3} + 90$
- *eff* : $\mathit{Place2State}' = 1$

loss
- *prec* : $\mathit{Place2State} = 2 \wedge l < 4$
- *deadline* : $\mathit{prec} \wedge T \geq \mathit{t3} + 90$
- *eff* : $l' = l + 1 \wedge \mathit{Place2State}' = 1$

DTA: *Monitor*
$V_L : \{\, t_m \in \mathbb{N}\,\}$
$\Theta_L : t_m = 0$
$TL : \{\, \mathit{check}\,\}$
$\mathcal{A}$: *check*
- *prec* : $T = t_m + 1000$
- *deadline* : *prec*
- *eff* : $t_m' = T \ \wedge\ x' = 0 \wedge l' = 0 \ \wedge\ \mathit{error}' = x < 14$

Figure 8.3: Multimedia Stream with Lossy Channel and Throughput Monitor

Figure 8.4 shows the product automaton for the network of Figure 8.3. The product automaton includes four completed actions *sourceOut*, each one representing a possible synchronisation between a *sourceOut!* in *Source* and a *sourceOut?* in either *Place1* or *Place2*, and two completed actions *sinkIn* that correspond to the synchronisation between *Place1* or *Place2* and the *Sink*.

8.3 Basic Transition Systems and Invariance Proofs

Basic Transition Systems (BTSs) are a well-known computational model for (untimed) infinite-state systems [111]. Basic transition systems, as originally defined, assume an underlying first-order language; in this thesis we use WS1S as the assertion language, instead. We use $\models$ to denote satisfiability under WS1S semantics.

8.3.1 Basic Transition Systems

A basic transition system is a tuple $B = (\mathcal{V}, \Sigma, \mathcal{T}, \Theta)$, where,

- $\mathcal{V}$ is a finite set of typed variables (boolean, first-order and second-order variables), which is partitioned into a set of *rigid* variables and *flexible* variables.
- Σ is the set of states which corresponds to all valuations of $\mathcal{V}$. We use $s(v)$ to denote the value of $v \in \mathcal{V}$ in $s \in \Sigma$.
- $\mathcal{T}$ is a finite set of transitions, which denote state changes. A transition $\tau \in \mathcal{T}$ is characterised by an assertion (usually called the *transition relation*),

$$\rho_\tau(\mathcal{V} \cup \mathcal{V}') \triangleq P_\tau(\mathcal{V}) \ \wedge \ E_\tau(\mathcal{V}')$$

  where $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ denotes the primed versions of variables in $\mathcal{V}$; P_τ is the *precondition* of τ, which describes the set of states where τ can be performed; and E_τ is the *effect* of τ, which describes the possible states (through primed variables) which result when τ is performed. We use $\mathcal{W}(\tau) \subseteq \mathcal{V}$ to denote the set of variables that are written by τ (i.e., any $v \in \mathcal{V}$ s.t. v' occurs free in E_τ). All other variables are assumed to remain unchanged after τ is performed, and are denoted by the set $\mathcal{U}(\tau) = \mathcal{V} \setminus \mathcal{W}(\tau)$. $\mathcal{T}$ is also assumed to include the *idling transition*, which does not change the current state:

$$\rho_{idle} \triangleq \bigwedge_{v \in \mathcal{V}} v' = v$$

  All other transitions in $\mathcal{T}$ are referred to as *diligent* transitions.[2]

[2]The term *diligent* here denotes useful computation, rather than urgent behaviour.

DTA: Π

V_L : $\{\, t1, t2, t3, t4, t_m, x \in \mathbb{N},\ l \in \{0, \ldots, 4\},\ error \in \mathbb{B},$
$SourceState \in \{0, 1\},\ Place1State, Place2State, SinkState \in \{1, 2\}\,\}$

Θ_L : $t1 = 0 \ \wedge\ t2 = 0 \ \wedge\ t3 = 0 \ \wedge\ t4 = 0 \ \wedge\ t_m = 0 \ \wedge\ x = 0 \ \wedge\ l = 0 \ \wedge\ \neg\, error \ \wedge$
$SourceState = 0 \ \wedge Place1State = 1 \ \wedge\ Place2State = 1 \ \wedge\ SinkState = 1$

TL : $\{\, sourceOut,\ sinkIn,\ loss,\ play,\ check\,\}$

$\mathcal{A}$: *sourceOut*
- *prec* : $SourceState = 0 \ \wedge\ T = 0 \ \wedge\ Place1State = 1$
- *deadline* : *prec*
- *eff* : $SourceState' = 1 \ \wedge\ Place1State' = 2 \ \wedge\ t4' = T$

sourceOut
- *prec* : $SourceState = 0 \ \wedge\ T = 0 \ \wedge\ Place2State = 1$
- *deadline* : *prec*
- *eff* : $SourceState' = 1 \ \wedge\ Place2State' = 2 \ \wedge\ t3' = T$

sourceOut
- *prec* : $SourceState = 1 \ \wedge\ T = t1 + 50 \ \wedge\ Place1State = 1$
- *deadline* : *prec*
- *eff* : $Place1State' = 2 \ \wedge\ t1' = T \ \wedge\ t4' = T$

sourceOut
- *prec* : $SourceState = 1 \ \wedge\ T = t1 + 50 \ \wedge\ Place2State = 1$
- *deadline* : *prec*
- *eff* : $Place2State' = 2 \ \wedge\ t1' = T \ \wedge\ t3' = T$

sinkIn
- *prec* : $Place1State = 2 \ \wedge\ T > t4 + 80 \ \wedge\ SinkState = 1$
- *deadline* : $Place1State = 2 \ \wedge\ T \geq t4 + 90 \ \wedge\ SinkState = 1$
- *eff* : $Place1State' = 1 \ \wedge\ t2' = T \ \wedge\ SinkState' = 2 \ \wedge\ x' = x + 1$

sinkIn
- *prec* : $Place2State = 2 \ \wedge\ T > t3 + 80 \ \wedge\ SinkState = 1$
- *deadline* : $Place2State = 2 \ \wedge\ T \geq t3 + 90 \ \wedge\ SinkState = 1$
- *eff* : $Place2State' = 1 \ \wedge\ t2' = T \ \wedge\ SinkState' = 2 \ \wedge\ x' = x + 1$

loss
- *prec* : $Place1State = 2 \ \wedge\ l < 4$
- *deadline* : $prec \ \wedge\ T \geq t4 + 90$
- *eff* : $l' = l + 1 \ \wedge\ Place1State' = 1$

loss
- *prec* : $Place2State = 2 \ \wedge\ l < 4$
- *deadline* : $prec \ \wedge\ T \geq t3 + 90$
- *eff* : $l' = l + 1 \ \wedge\ Place2State' = 1$

play
- *prec*: $SinkState = 2 \ \wedge\ T = t2 + 5$
- *deadline*: *prec*
- *eff*: $SinkState' = 1$

check
- *prec* : $t_m + 1000 = T$
- *deadline* : *prec*
- *eff* : $t_m' = T \ \wedge\ x' = 0 \ \wedge\ l' = 0 \ \wedge\ error' = x < 14$

Figure 8.4: Product Automaton (DTA) for the Multimedia Stream

- $\Theta(\mathcal{V})$ (*the initial condition*) is an assertion representing a set of initial states.

For $s \in S$, let *primed*(s) be the valuation that renames every variable in s by its primed version (as defined in Section 8.2.2). Given a state s_1, the set of possible next states after τ is performed[3] is given by:

$$\tau(s_1) \triangleq \{\, s_2 \,|\, s_1 \cup \mathit{primed}(s_2) \models (\rho_\tau \ \wedge \bigwedge_{u \in \mathcal{U}(\tau)} u' = u) \,\}$$

A *computation* is an infinite sequence of states and transitions,

$$s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} \ldots$$

such that $s_0 \models \Theta$, $s_{i+1} \in \tau_i(s_i)$ $(i \geq 0)$, and the sequence contains either infinitely many diligent steps (i.e., the result of executing a diligent transition) or a *terminal state*, i.e., a state where the only enabled transition is the idling transition (note that if this is the case, then the computation contains an infinite suffix of terminal states). A state is reachable if it belongs to some computation. We use $s \xrightarrow{\tau} s' \in B$ to denote a τ-step from s in some computation of B. We also extend this notation to finite subsequences of computations in B, e.g., $s \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_n} s_{n+1} \in B$.

8.3.2 Invariance Proofs: Deductive Verification of Safety Properties

Intuitively, a safety property characterises a certain assertion that must hold for all reachable states in any system execution. We have seen in Chapter 2 how safety properties are expressed in CTL, as this suits the branching-time semantics of timed automata (timed transition systems / computation trees).

Basic transition systems, on the other hand, are given a linear-time semantics in terms of computations. In this context, a safety property can be naturally expressed in LTL [111], by the formula $\Box\,\phi$, where ϕ is an LTL *past formula*. In this thesis, we are concerned just with simple safety properties, where ϕ, called an *invariant*, is a WS1S assertion (which we will also refer to as a state formula). Formally, if σ denotes a computation, then the following states when $\Box\,\phi$ holds in σ (denoted, $\sigma \models \Box\,\phi$).

$$\sigma \models \Box\,\phi \ \text{ iff } \ \forall i, i \geq 0.\ s_i \models \phi$$

where s_i $(i \geq 0)$ is a state in σ. In addition, we say that $\Box\,\phi$ holds in a BTS if $\Box\,\phi$ holds in all

[3]Transition relations are defined in [111] so that the next state is unique (if it exists). We adopt here the more general form, which simplifies some of the DTA specifications presented in this chapter.

its computations (see [111] for further details on the semantics of LTL). Safety properties can be asserted with *invariance proofs*, which are characterised by the rule INV shown in Figure 8.5 [112].[4]

$$\frac{\begin{array}{ll}\text{P1} & \varphi \rightarrow \phi \\ \text{P2} & \Theta \rightarrow \varphi \\ \text{P3} & \forall \tau \in \mathcal{T} \,.\, (\rho_\tau \ \wedge \ \varphi) \rightarrow \varphi' \end{array}}{\Box \phi}$$

Figure 8.5: The Invariance Rule INV

This rule guarantees the invariance of ϕ, provided there exists a (usually stronger) invariant φ such that (P1) φ implies ϕ, (P2) φ holds in every starting state and (P3) φ is preserved by all transitions in $\mathcal{T}$, i.e., if φ holds whenever τ is enabled, then it also holds after τ is performed (φ is called an *inductive* assertion). The formula φ' can be obtained by replacing, in φ, all variables v by their primed version v', where v is a modifiable variable in ρ_τ. Manna and Pnueli [112] proved soundness and completeness of this rule w.r.t. assertions in first-order language. It is not difficult to see that the rule is also sound for assertions in WS1S.

One of the limitations of this rule is that, for any assertion φ, premises may be found to be unsatisfiable for some valuations, even in those cases when φ is, indeed, an invariant (i.e., when φ is not inductive). The following discussion explains this issue in more detail.

Without loss of generality, let us assume that $\varphi \triangleq \phi$ is a good guess to start our deductive proof. Suppose that a valuation is found that invalidates the premise P3 for some transition τ, i.e., there exists a state s that satisfies φ such that, if τ would be performed in s, φ would not hold in (at least one of) the resulting next states. If s is a reachable state, then ϕ is guaranteed not to be an invariant. On the other hand, if s is unreachable, then nothing can be concluded about ϕ: τ has been found not to preserve ϕ, but for all we know, this may be just a consequence of applying a τ-step at an unreachable state (i.e., a τ-step that is not part of any computation). This issue is inherent in the nature of transition relations; transition relations represent a set of states where the transition is enabled, but not every state in this set is necessarily reachable.

One way to approach this problem is to assume $\varphi \triangleq \phi$ initially, and then to strengthen it with auxiliary invariants to rule out unreachable states. This process results in a formula φ for which the deductive rule provides a definite answer; φ will typically have the form:

$$\varphi \triangleq \phi \ \wedge \ \bigwedge_{i=1}^{n} \alpha_i$$

[4]For the sake of consistency with [111, 112], in this chapter we will use $\rightarrow$ to denote implication, $\leftrightarrow$ to denote equivalence, and $\Rightarrow$ to denote entailment: Given a computation σ, $p \rightarrow q$ holds over σ iff it holds on the first state of σ, while $p \Rightarrow q$ holds over σ iff $p \rightarrow q$ holds on every state of σ.

where α_i, for all $1 \leq i \leq n$, is an auxiliary invariant describing some known relationship between the variables in $\mathcal{V}$. Ingenuity is needed to propose (and prove) convenient auxiliary invariants, thus proving safety properties often requires several applications of the deductive rule. Many heuristics have been proposed in the literature to guide the invariant strengthening process, and automatic invariant generation is in itself an active research topic [59, 112, 27, 26, 143].

A Note on Completeness. Manna and Pnueli [112, p. 220] proved the completeness of the invariance rule assuming a first-order language. For any valid safety property $\Box\,\phi$ (where ϕ is a state formula), they proved the existence of an inductive assertion acc_ϕ, s.t. $acc_\phi \rightarrow \phi$ (in other words, acc_ϕ satisfies the premises of rule INV).[5] However, expressing acc_ϕ requires general (integer) addition and multiplication, which are not expressible in WS1S. This suggest that the invariance rule may be incomplete for WS1S: given some *valid* safety property $\Box\,\phi$, there may not exist an inductive assertion φ s.t. $\varphi \rightarrow \phi$. In other words, we may not be able to prove valid safety properties using the invariance rule. Proving completeness (or incompleteness) of the invariance rule for WS1S is still an open problem, and one objective of our future research.

8.4 Proving Safety Properties of DTAs

This section provides a mapping from DTAs to BTSs, so that the invariance rule can be applied to prove safety properties of DTA specifications (Section 8.4.1). We present case studies in Sections 8.4.2 and 8.4.3.

8.4.1 Mapping Discrete Timed Automata to Basic Transition Systems

We presented an operational semantics for DTAs in Section 8.2.2, in terms of (discrete) timed transition systems. In order to construct invariance proofs, though, we need to map DTAs to BTSs.[6] The mapping of preconditions and effects is straightforward; we just need to find a proper representation for delays and deadlines.

As is customary in timed frameworks based on transition systems, delays can be accounted for with a suitably defined *time action*, which increments the time variable T every time it is executed. However, our goal here is to use MONA to assist the verification, and so the time action should be defined in such a way that it can be expressed in WS1S. Thus, we define a transition *tick*, which increments the value of T by 1. The effect of deadlines in computations will be accounted for by constraining (the enabling condition of) the *tick* transition with a conjunct

[5] Actually, Manna and Pnueli [112, p. 372] proved completeness of the deductive rule to prove validity of general safety properties $\Box\,p$, where p is an LTL past formula.

[6] Invariance proofs have also been defined for extensions to Basic Transition Systems, such as Fair Transition Systems [111] and Clock Transition Systems [110]. However, in our case, basic transition systems suffice.

$\neg\,(p\ \wedge\ d)$, for each action (a, p, d, e) in the DTA (time can only pass when no enabled action is urgent). The mapping is formally defined as follows.

Let $|A$ be a network of DTAs, and $\Pi = (V_L, \Theta_L, TL, \mathcal{A}, \emptyset, \mathit{true})$ the product automaton for $|A$. Assume that actions in Π are uniquely identified by their labels (or alternatively, apply a proper renaming). We associate with Π a BTS $B_\Pi = (\mathcal{V}, \Sigma, \Theta, \mathcal{T})$, where

- $\mathcal{V} = V_L \cup \{T\}$;
- Σ is the set of states which corresponds to all type-consistent valuations for $\mathcal{V}$.
- $\Theta = \Theta_L$; and
- $\mathcal{T} = TL \cup \{\, \mathit{idle}, \mathit{tick} \,\}$, where

$$\begin{aligned} \rho_a &\triangleq p\ \wedge\ e \quad \text{for each } (a, p, d, e) \in \mathcal{A} \\ \rho_{tick} &\triangleq (T' = T + 1)\ \wedge \bigwedge_{(a,p,d,e)\in\mathcal{A}} \neg\,(p\ \wedge\ d) \end{aligned}$$

Theorem 8.1 below proves the equivalence of DTAs and BTSs.

THEOREM **8.1.** *Let $|A$ be a network of DTAs with product automaton $\Pi = (V_L, \Theta_L, TL, \mathcal{A}, \emptyset, \mathit{true})$, $TS_\Pi = (S, S_0, Lab, T_S)$ the TTS arising from the operational semantics of Π, and $B_\Pi = (\mathcal{V}, \Sigma, \Theta, \mathcal{T})$ the corresponding BTS. Then, for any $(a, p, d, e) \in \mathcal{A}$ and $\delta \in \mathbb{N}^+$,*

1. $s \stackrel{a}{\Longrightarrow} s' \in T_S$ *iff* $s \xrightarrow{a} s' \in B_\Pi$; *and*
2. $s \stackrel{\delta}{\Longrightarrow} (s+\delta) \in T_S$ *iff* $s \xrightarrow{tick} (s+1) \xrightarrow{tick} (s+2) \xrightarrow{tick} \ldots (s+\delta) \in B_\Pi$.

Proof. The theorem holds by construction of B_Π. First, note that the set of possible states in TS_Π and B_Π are the same, i.e., $S = \Sigma$. Secondly, both TS_Π and B_Π have the same initial state, because $\Theta \triangleq \Theta_L$. In addition, for every transition relation $\rho_a \triangleq p\ \wedge\ e$ in B_Π there exists a corresponding action (a, p, d, e) in Π with the same precondition and effect (and viceversa).

The theorem's second statement is a consequence of the following observations. First, by rule (D2) governing the semantics of a DTA (Section 8.2.2), for any $\delta \in \mathbb{N}^+$ and $s \in S$, the following holds

$$s \stackrel{\delta}{\Longrightarrow} (s+\delta)\ \mathit{iff}\ s \stackrel{1}{\Longrightarrow} (s+1) \stackrel{1}{\Longrightarrow} (s+2) \stackrel{1}{\Longrightarrow} \ldots (s+\delta)$$

Secondly, by rule (D2) and definition of ρ_{tick}, for any $s \in S$, the following holds.

$$s \stackrel{1}{\Longrightarrow} (s+1)\ \mathit{iff}\ s \xrightarrow{tick} (s+1)$$

This concludes the proof. □

Discussion: Timelocks in BTSs. Naturally, having mapped a DTA Π (with a timed semantics) to a BTS B_Π (with untimed semantics), we may wonder about the interpretation of Zeno runs and timelocks of Π, in B_Π. Note that, by definition, the idling transition is included in B_Π. Because time-actionlocks cannot occur in Π, every reachable state in B_Π enables either some action transition (i.e., a transition in B_Π obtained from some action in Π) or the *tick* transition. Consequently, there are no terminal states in B_Π, and computations are always infinite sequences of diligent transitions (in other words, the idling transition loses its meaning in B_Π).

Zeno runs and Zeno-timelocks may occur in Π, and therefore also in B_Π. As is the case for model-checking timed automata (Chapter 3), timelocks in BTS specifications may invalidate the verification of correctness requirements. Moreover, timelocks are a source of incompleteness for proof rules, such as the invariance rule (see, e.g., [1, 93, 25]).

Zeno runs in timelock-free specifications, on the other hand, do not affect the verification of LTL formulae, whose semantics are defined w.r.t. divergent executions. Unlike model-checkers, proof rules can selectively ignore Zeno runs: in general, the verification of liveness properties is performed under fairness assumptions (of which time divergence can be seen as a particular case).[7] Therefore, in accord with the literature (and in particular, with [93]) we require the DTA Π to be timelock-free, and we consider the semantics of LTL formulae to be defined only over *divergent* computations (i.e., computations where the value of T increases beyond any bound).[8]

8.4.2 Example: Safety Properties for the Multimedia Stream

Consider again the multimedia stream example, and the product automaton Π depicted by Figure 8.4. Figure 8.6 shows the equivalent BTS B_Π (we have renamed the actions in Π to unique labels). In this section we prove a number of requirements for the multimedia stream.

Non-blocking Channel. Invariance proofs can be used on B_Π to confirm that synchronisation between *Source* and either *Place1* or *Place2* is always possible, i.e., that packets can be put in the channel whenever the *Source* is ready to send them. This safety property[9] can be expressed by the LTL formula $\psi_{nonblock}$, where

$$\psi_{nonblock} \triangleq \Box\neg((T = 0 \vee T = t1 + 50) \wedge Place1State = 2 \wedge Place2State = 2)$$

[7] Note, we are not claiming that proving liveness is easy (which is far from true!), but instead that proofs in specifications with Zeno runs are possible and meaningful.

[8] Unless where it has been stated otherwise, we deal with DTA specifications that are timelock-free.

[9] We have seen a CTL formula describing the same property in the context of Timed Automata in Section 2.2.2.

$\mathcal{V}$: $\{\, T,\ t1,\ t2,\ t3,\ t4,\ t_m,\ x \in \mathbb{N},\ l \in \{0,\ldots,4\},\ error \in \mathbb{B},$
$SourceState \in \{0,1\},\ Place1State,\ Place2State,\ SinkState \in \{1,2\}\,\}$

Θ : $T = 0\ \wedge\ t1 = 0\ \wedge\ t2 = 0\ \wedge\ t3 = 0\ \wedge\ t4 = 0\ \wedge\ t_m = 0\ \wedge\ x = 0\ \wedge\ l = 0\ \wedge\ \neg\, error\ \wedge$
$SourceState = 0\ \wedge\ Place1State = 1\ \wedge\ Place2State = 1\ \wedge\ SinkState = 1$

$\mathcal{T}$: $\{$ ρ_{tick} : $\neg\,(SourceState = 0\ \wedge\ T = 0\ \wedge\ Place1State = 1)\ \wedge$
$\neg\,(SourceState = 0\ \wedge\ T = 0\ \wedge\ Place2State = 1)\ \wedge$
$\neg\,(SourceState = 1\ \wedge\ T = t1 + 50\ \wedge\ Place1State = 1)\ \wedge$
$\neg\,(SourceState = 1\ \wedge\ T = t1 + 50\ \wedge\ Place2State = 1)\ \wedge$
$\neg\,(Place1State = 2\ \wedge\ T \geq t4 + 90\ \wedge\ SinkState = 1)\ \wedge$
$\neg\,(Place2State = 2\ \wedge\ T \geq t3 + 90\ \wedge\ SinkState = 1)\ \wedge$
$\neg\,(SinkState = 2\ \wedge\ T = t2 + 5)\ \wedge$
$\neg\,(Place1State = 2\ \wedge\ l < 4\ \wedge\ T \geq t4 + 90)\ \wedge$
$\neg\,(Place2State = 2\ \wedge\ l < 4\ \wedge\ T \geq t3 + 90)\ \wedge$
$\neg\,(T = t_m + 1000)\ \wedge$
$T' = T + 1$

$\rho_{sourceOut01}$: $SourceState = 0\ \wedge\ T = 0\ \wedge\ Place1State = 1\ \wedge$
$SourceState' = 1\ \wedge\ t4' = T\ \wedge\ Place1State' = 2$

$\rho_{sourceOut02}$: $SourceState = 0\ \wedge\ T = 0\ \wedge\ Place2State = 1\ \wedge$
$SourceState' = 1\ \wedge\ t3' = T\ \wedge\ Place2State' = 2$

$\rho_{sourceOut11}$: $SourceState = 1\ \wedge\ T = t1 + 50\ \wedge\ Place1State = 1\ \wedge$
$t1' = T\ \wedge\ t4' = T\ \wedge\ Place1State' = 2$

$\rho_{sourceOut12}$: $SourceState = 1\ \wedge\ T = t1 + 50\ \wedge\ Place2State = 1\ \wedge$
$t1' = T\ \wedge\ t3' = T\ \wedge\ Place2State' = 2$

$\rho_{sinkIn1}$: $Place1State = 2\ \wedge\ T > t4 + 80\ \wedge\ SinkState = 1\ \wedge$
$Place1State' = 1\ \wedge\ t2' = T\ \wedge\ SinkState' = 2\ \wedge\ x' = x + 1$

$\rho_{sinkIn2}$: $Place2State = 2\ \wedge\ T > t3 + 80\ \wedge\ SinkState = 1\ \wedge$
$Place2State' = 1\ \wedge\ t2' = T\ \wedge\ SinkState' = 2\ \wedge\ x' = x + 1$

ρ_{loss1} : $Place1State = 2\ \wedge\ l < 4\ \wedge\ l' = l + 1\ \wedge\ Place1State' = 1$

ρ_{loss2} : $Place2State = 2\ \wedge\ l < 4\ \wedge\ l' = l + 1\ \wedge\ Place2State' = 1$

ρ_{play} : $SinkState = 2\ \wedge\ T = t2 + 5\ \wedge\ SinkState' = 1$

ρ_{check} : $T = t_m + 1000\ \wedge\ t_m' = T\ \wedge\ x' = 0\ \wedge\ l' = 0\ \wedge\ error' = x < 14$
$\}$

Figure 8.6: BTS B_Π for the Multimedia Stream

As discussed in Section 8.3, the verification of $\psi_{nonblock}$ is achieved by applying the rule INV (Figure 8.5). We have used MONA to check the validity of the rule premises for every transition in B_Π. The following auxiliary invariants, which describe various relationships among the DTA variables, proved helpful in the verification of $\psi_{nonblock}$. These invariants were also validated with MONA (the MONA predicate *mult50*(n) denotes that n is a multiple of 50; see Figure 8.7).

(1) $(\mathit{Place1State} = 2 \ \wedge \ \mathit{Place2State} = 2) \ \rightarrow \ (\mathit{t3} \geq \mathit{t4} + 50 \vee \mathit{t4} \geq \mathit{t3} + 50)$

(2) $\mathit{t1} \geq \mathit{t3} \ \wedge \ \mathit{t1} \geq \mathit{t4}$

(3) $\mathit{SourceState} = 0 \rightarrow T = 0 \ \wedge \ \mathit{Place1State} = 1 \ \wedge \ \mathit{Place2State} = 1$

(4) $(T > \mathit{t4} + 90 \rightarrow \mathit{Place1State} = 1) \ \wedge \ (T > \mathit{t3} + 90 \ \rightarrow \ \mathit{Place2State} = 1)$

(5) $T \geq \mathit{t1} \ \wedge \ T \geq \mathit{t2} \ \wedge \ T \geq \mathit{t3} \ \wedge \ T \geq \mathit{t4}$

(6) $\mathit{mult50}(\mathit{t1}) \ \wedge \ \mathit{mult50}(\mathit{t3}) \ \wedge \ \mathit{mult50}(\mathit{t4})$

(7) $\mathit{t2} = T \ \wedge \ T > 0 \ \rightarrow ((\mathit{Place1State} = 1 \ \wedge \ T \leq \mathit{t4} + 90 \ \wedge \ T > \mathit{t4} + 80) \vee$
$(\mathit{Place2State} = 1 \ \wedge \ T \leq \mathit{t3} + 90 \ \wedge \ T > \mathit{t3} + 80))$

(8) $\mathit{SinkState} = 2 \ \rightarrow \ T \leq \mathit{t2} + 5$

(9) $T \leq t1 + 50$

```
pred mult50(var1 n) =
     ex2 M:
     (all1 x where (x in M & x < max M): x+50 in M & ~ ex1 y :  x<y & y<x+50 & y in M) &
     0 in M & n in M;
```

Figure 8.7: A MONA Predicate To Test Multiples of 50

By way of example, Figure 8.8 shows the MONA specification that checks whether the assertion $\neg((T = 0 \vee T = \mathit{t1} + 50) \wedge \mathit{Place1State} = \mathit{2} \wedge \mathit{Place2State} = \mathit{2})$ is preserved by the *tick* transition (i.e., one of the proof obligations derived from premise P3 of the invariance rule).

Latency. Latency refers to the time a packet takes to be transmitted and eventually played by the *Sink*: we want to verify that this time must be at most 95 ms. In the network of Figure 8.3, the variable *t1* in *Source* captures the time when each packet is sent (see the effect of *sourceOut!*), and $t2 + 5$ in *Sink* captures the time when a packet is played (see the precondition of *play*). However, the (intuitive) formula $\Box\,(\mathit{t2} + 5 \leq \mathit{t1} + 95)$ does not correctly represent the latency; by the time the *Sink* receives a packet, *t1* has already captured the time when the next packet was sent. Therefore, we need to relate the sending and playing times for each packet. For this specification, it suffices to capture the sending times of the last two packets [44].

```
var1 T,T',t1,t3,t4,t2,l where (0<=l & l<=4),
     SourceState where SourceState in {0,1},
     Place1State where Place1State in {1,2},
     Place2State where Place2State in {1,2},
     SinkState where SinkState in {1,2};

~ (SourceState=0 & T=0 & Place1State=1) &
~ (SourceState=0 & T=0 & Place2State=1) &
~ (SourceState=1 & T=t1+50 & Place1State=1) &
~ (SourceState=1 & T=t1+50 & Place2State=1) &
~ (Place1State=2 & T>=t4+90 & SinkState=1) &
~ (Place2State=2 & T>=t3+90 & SinkState=1) &
~ (SinkState=2 & T=t2+5) &
~ (Place1State=2 & l<4 & T >= t4+90) &
~ (Place2State=2 & l<4 & T >= t3+90) &
~ (T=tm+1000) &
T' = T+1 &
(~ ((T=0 | T=t1+50) & Place1State=2 & Place2State=2))
=>
(~ ((T'=0 | T'=t1+50) & Place1State=2 & Place2State=2));
```

Figure 8.8: A MONA Specification to Characterise a Proof Obligation from the Invariance Rule

Figure 8.9 shows how the original *Source* automaton is modified to introduce the variables t_{10} and t_{11}, which capture the sending times of the last two packets (no other component automaton requires any modification). After sending the first packet (*SourceState=0*), the *Source* enters into a 2-state loop (*SourceState=1*, *SourceState=2*), capturing the time when each packet is sent (t_{10}, t_{11}). Latency can then be expressed by the safety property,

$$\psi_{latency} \triangleq \Box\,((\mathit{SinkState} = 2 \ \wedge\ T = t2 + 5) \ \rightarrow\ (T \leq t_{10} + 95 \ \wedge\ T \leq t_{11} + 95))$$

DTA: *Source*
V_L : $\{\, \mathit{SourceState} \in \{0,1,2\},\ t_{10}, t_{11} \in \mathbb{N} \,\}$
Θ_L : $\mathit{SourceState} = 0 \ \wedge\ t_{10} = 0 \ \wedge\ t_{11} = 0$
TL : $\{\, \mathit{SourceOut}! \,\}$
$\mathcal{A}$: *SourceOut!*
  prec : $\mathit{SourceState} = 0$
  deadline : *prec*
  eff : $\mathit{SourceState}' = 1$
SourceOut!
  prec : $\mathit{SourceState} = 1 \ \wedge\ T = t_{10} + 50$
  deadline : *prec*
  eff : $\mathit{SourceState}' = 2 \ \wedge\ t'_{11} = T$
SourceOut!
  prec : $\mathit{SourceState} = 2 \ \wedge\ T = t_{11} + 50$
  deadline : *prec*
  eff : $\mathit{SourceState}' = 1 \ \wedge\ t'_{10} = T$

Figure 8.9: The *Source* Automaton Modified to Verify Latency

We have proved $\psi_{latency}$ using the invariance rule, with MONA validating the necessary proof obligations (here we omit the auxiliary invariants). We also refer the reader to our technical report [73], where similar proofs are given in detail.

Throughput. We want to verify that the *Sink* never receives less than 14 packets per second. In our DTA specification, this could be confirmed by checking the safety property $\Box \neg \mathit{error}$. The invariance proof would require auxiliary invariants to relate the number of received and lost packets in a given second (variables x and l), and the time elapsed so far (T). However, we need multiplication to formalise such a relationship, which is not expressible in WS1S. Instead, we will perform an analysis of throughput based on the receiving time of the last packet ($t2$).

Assuming a reliable channel, the minimum throughput can be expressed as a function of the packet sending rate and the maximum transmission delay. Using invariance proofs, we can confirm that the maximum accumulated time after n packets have been received, is given (in ms) by equation 8.1, as follows.

$$at(n) \quad = \quad 90 + 50(n-1) \tag{8.1}$$

In the DTA specification, the accumulated time up to the last packet received by the *Sink* is given by $t2$, and so, if we add a variable n that is incremented every time a packet is received (but n is never reset), we may attempt to assert the formula:

$$\Box\,(l = 0 \;\rightarrow\; t2 \leq 90 + 50(n-1))$$

However, as we mentioned before, multiplication is not expressible in WS1S. Thus, we introduce a new variable $N_{50} = 50n$ instead of n, which is incremented by 50 every time that a new packet is received by the *Sink*. This yields the formula:,

$$\psi_{thr} \;\triangleq\; \Box\,(l = 0 \;\rightarrow\; t2 \leq N_{50} + 40)$$

We have validated ψ_{thr} with MONA (again, we omit the auxiliary invariants), and thus confirmed equation 8.1. Assuming a reliable channel, the minimum throughput in the i-th second, thr^i_{min}, can be calculated as the difference between the accumulated number of packets received up to the i-th and $(i-1)$-th seconds. In terms of accumulated time, thr^i_{min} is given by the following expression.

$$thr^i_{min} = \begin{cases} max(\{n \mid 1000 \geq at(n)\}) & \text{if } i = 1 \\ max(\{n \mid 1000i \geq at(n)\}) - max(\{n \mid 1000(i-1) \geq at(n)\}) & \text{otherwise} \end{cases}$$

Induction on i will prove that the minimum throughput at any second, thr_{min}, is 19 packets per second. In order to infer the minimum throughput for a lossy channel, the worst-case analysis must consider the following scenario.

When $T = 1000(i-1)$ (i.e., at the beginning of the i-th second), the action *check* in *Monitor* becomes urgent, but the key observation here is that, although time may not pass, components may still evolve if some action is enabled. In particular, note that when $T = 1000(i-1)$, *Source* will synchronise with either *Place1* or *Place2* to put another packet in the channel, and that such a packet may be immediately lost (provided $l < 4$, action *loss* can be performed as soon as *Source* and *Place1*/*Place2* have synchronised). Moreover, suppose that the packet is lost before the action *check* is performed. Then, as the number of lost packets is reset ($l' = 0$ in the *check* action), up to 5 packets can be lost in the current second (4 packets plus the first packet sent in the current second). We conclude that the minimum throughput, assuming a lossy channel of up to 4 packets lost per second, is $thr_{min}^{lossy} = thr_{min} - 5 = 14$ packets per second (thus, the *Monitor* component never triggers an *error* event).

8.4.3 Other Examples

This section offers more examples of DTA specifications, including examples that illustrate the specification of unbounded data structures in DTAs. Thanks to WS1S and MONA, DTAs provide data support for a class of systems that timed automata and model-checkers are not able to analyse (or where finite abstractions must be provided, which may sacrifice generality).

Multimedia Stream with Unbounded Channel

Here we adapt the DTA specification of Figure 8.3 to implement the channel as an unbounded buffer (instead of a two-place buffer); this is shown in Figure 8.10 (the *Source*, *Sink* and *Monitor* remain unchanged). The buffer denotes an unbounded queue storing timestamps, which record the time when the packet is put into the channel. We do not model here the content of packets, although this is possible in WS1S if the content can be represented in some finite domain [148, 29]. We represent the buffer as a finite set of natural numbers ($Buf \in \mathcal{F}(\mathbb{N})$), manipulated as a queue through the following predicates (this is possible because timestamps, t, are monotonically increasing).

Note that, a packet can be lost as soon as it is put into the channel. Correspondingly, the predicate *lose*(*Buf*,*Buf'*) models this event by removing (non-deterministically) one of the timestamps that are currently stored in the buffer (i.e., this denotes the loss of one of the packets in transit).

DTA: *Channel*
V_L : $\{\, Buf \in \mathcal{F}(\mathbb{N}) \,\}$
Θ_L : $Buf = \emptyset$
TL : $\{\, sourceOut?,\ sinkIn!,\ loss \,\}$
$\mathcal{A}$: *sourceOut?*
eff : $enQ(T, Buf, Buf')$
sinkIn!
prec : $\exists t.\ (head(t, Buf)\ \wedge\ T > t + 80)$
deadline : $\exists t.\ (head(t, Buf)\ \wedge\ T \geq t + 90)$
eff : $deQ(Buf, Buf')\ \wedge\ x' = x + 1$
loss
prec : $\neg\, isEmpty(Buf)\ \wedge\ l < 4$
deadline : $l < 4\ \wedge\ \exists t.(head(t, Buf)\ \wedge\ T \geq t + 90)$
eff : $lose(Buf, Buf')\ \wedge\ l' = l + 1$

Figure 8.10: An Unbounded Channel for the Multimedia Stream

$$
\begin{aligned}
isEmpty(Buf) &\triangleq Buf = \emptyset \\
head(t, Buf) &\triangleq t = min(Buf) \\
enQ(t, Buf, Buf') &\triangleq Buf' = Buf \cup \{t\} \\
deQ(Buf, Buf') &\triangleq Buf' = Buf \setminus \{min(Buf)\} \\
lose(Buf, Buf') &\triangleq \exists\, t \in Buf.\, Buf' = Buf \setminus \{t\}\ \wedge\ Buf' \neq Buf
\end{aligned}
$$

Abadi and Lamport's Unbounded Queue

Here we show a DTA specification for the example of Abadi and Lamport [1]. Figure 8.11 illustrates an unbounded queue, which acts as a channel between a sender and a receiver process (we do not model these processes here, just the behaviour of the channel). All components evolve in discrete-steps, according to a single reference clock.

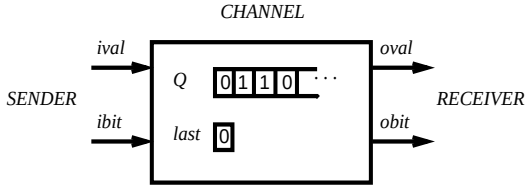


Figure 8.11: A Non-lossy Queue

There are two pairs of boolean-valued wires, one pair on each side of the channel: *ival* and *ibit*, and *oval* and *obit*. The sender inputs a new message by setting the value of *ival* to the message content, and signals this to the channel by complementing *ibit*. The sender may input a message at most once every δ_{snd} time units.

DTA: *Queue*
V_L : $\{\, ibit, obit, ival, oval, last \in \mathbb{B},\ x_{inp}, x_{deq}, size \in \mathbb{N},\ Q, Q_i, Q_e \in \mathcal{F}(\mathbb{N}) \,\}$
Θ_L : $isEmpty(Q, size)\ \wedge\ Q_i = \emptyset\ \wedge\ Q_e = \emptyset\ \wedge\ x_{inp} = 0\ \wedge\ x_{deq} = 0$
TL : $\{\, Inp,\ EnQ,\ DeQ \,\}$
$\mathcal{A}$: *Inp*

prec :	$x_{inp} + \delta_{snd} \leq T$
eff :	$ibit' = \neg\, ibit\ \wedge$
	$x'_{inp} = T\ \wedge\ Q'_i = Q_i \cup \{T\}$

EnQ

prec :	$last = \neg\, ibit$
deadline :	$prec\ \wedge\ x_{inp} + \Delta_{rcv} \leq T$
eff :	$enQ(ival, Q, Q', size, size')\ \wedge$
	$last' = \neg\, last\ \wedge$
	$isEmpty(Q, size)\ \rightarrow\ x'_{deq} = T\ \wedge$
	$Q'_e = Q_e \cup \{T\}$

DeQ

prec :	$\neg\, isEmpty(Q, size)\ \wedge\ x_{deq} + \delta_{snd} \leq T$
deadline :	$prec\ \wedge\ x_{deq} + \Delta_{snd} \leq T$
eff :	$get(oval', Q, Q', size, size')\ \wedge$
	$obit' = \neg\, obit\ \wedge$
	$\neg\, isEmpty(Q, size)\ \rightarrow\ x'_{deq} = T$

Figure 8.12: A Non-lossy Queue (DTA Model)

The channel is either sensing the *ibit* wire to detect a new message from the sender, appending a new message to the queue (Q), or sending the message at the head of the queue to the receiver. Whenever the channel is sensing, it compares the values of *ibit* in the current and previous state (which is recorded in the internal variable *last*); if they differ then the content of *ival* is appended to the queue. The channel passes messages to the receiver by setting the value of *oval* to the content of the head of the queue, and complementing *obit*. The channel enqueues incoming messages no later than Δ_{rcv} time units after they were input by the sender, and passes messages to the receiver between δ_{snd} and Δ_{snd} time units after they reach the head of the queue (with $\delta_{snd} \leq \Delta_{snd}$). The queue is non-lossy for $\Delta_{rcv} < \delta_{snd}$ [1].

Figure 8.12 is a DTA specification of the queue, with constants $\delta_{snd} = 5$, $\Delta_{rcv} = 3$ and $\Delta_{snd} = 7$. To simplify matters, we consider that messages are boolean values. Actions *Inp*, *EnQ* and *DeQ* denote the sender putting a new message, the channel putting a message into the queue and the channel sending the message at the head of the queue to the receiver, respectively.

The queue is represented by a pair $(Q, size)$, where Q is a finite set of indices and *size* is the number of messages in the queue ($max(Q) < size$). For all $0 \leq i \leq size - 1$, the message stored in the i-th slot of the queue has a *true*-value if and only if $i \in Q$. For example, for $Q = \{1, 2\}$ and $size = 4$, the queue contains (in order) the messages *false*, *true*, *true*, *false*. Queue operations are represented by the following MONA predicates.

$$
\begin{array}{lcl}
\mathit{isEmpty}(Q, \mathit{size}) & \triangleq & Q = \emptyset \ \wedge \ \mathit{size} = 0 \\
\mathit{enQ}(\mathit{elem}, Q, Q', \mathit{size}, \mathit{size}') & \triangleq & \mathit{size}' = \mathit{size} + 1 \ \wedge \\
 & & (\mathit{elem} \ \rightarrow \ Q' = Q \cup \{\mathit{size}\}) \ \wedge \\
 & & (\neg \, \mathit{elem} \ \rightarrow \ Q' = Q) \\
\mathit{get}(\mathit{elem}, Q, Q', \mathit{size}, \mathit{size}') & \triangleq & \mathit{size} > 0 \ \wedge \\
 & & (\mathit{elem} \leftrightarrow 0 \in Q) \ \wedge \\
 & & (\forall \, e \in Q. \, e > 0 \ \rightarrow \ e - 1 \in Q') \ \wedge \\
 & & (\forall \, e \in Q'. \, e + 1 \in Q) \ \wedge \\
 & & \mathit{size}' = \mathit{size} - 1
\end{array}
$$

The time capture variables $x_{inp}, x_{deq} \in \mathbb{N}$ are used to enforce the input rate and the maximum delay to put a message in the queue (x_{inp}), and the minimum and maximum delays to pass a message from the queue to the receiver (x_{deq}).

In action *Inp*, the terms $x_{inp} + \delta_{snd} \leq T$ and $x'_{inp} = T$ ensure that messages are input at most once every δ_{snd} time units. Together, the terms $x'_{inp} = T$ in *Inp* and $x_{inp} + \Delta_{rcv} \leq T$ in *EnQ*, ensure that messages are put into the queue no later than Δ_{rcv} time units after they were input (because $\Delta_{rcv} < \delta_{snd}$, the channel has always time to enqueue the last input message before the next one arrives). The terms $\mathit{isEmpty}(Q, \mathit{size}) \ \rightarrow \ x'_{deq} = T$ in *EnQ*, and $x_{deq} + \delta_{snd} \leq T$, $x_{deq} + \Delta_{snd} \leq T$ (deadline) and $\neg \, \mathit{isEmpty}(Q, \mathit{size}) \ \rightarrow \ x'_{deq} = T$ in *DeQ*, ensure that a message is sent to the receiver between δ_{snd} and Δ_{snd} time units after it reaches the head of the queue. Note that x_{deq} is reset ($x'_{deq} = T$) as soon as a new value is available at the head of the queue.

Assume that the system is timelock-free, and that every message that is enqueued is eventually sent. We could prove that the queue does not lose messages, by checking that every input message is eventually put into the queue. One way to express this requirement as a safety property is to add two variables, Q_i and Q_e, which respectively record the time when a message is input, and the time when that message is put into the queue.

If we are to ensure that no messages are lost, then there must exist a temporal correspondence between the timestamps stored in Q_i and Q_e. At any given time, a timestamp t_i is stored in Q_i (except maybe the last one) if and only if a timestamp t_e is stored in Q_e, such that $t_i \leq t_e \leq t_i + \Delta_{rcv}$. The safety property in question is represented by the formula $\psi_{nonlossy}$,

$$
\begin{array}{l}
\psi_{nonlossy} \ \triangleq \ \Box(\phi_{ie} \ \wedge \ \phi_{ei}) \\
\phi_{ie} \ \triangleq \ \forall \, t_i \in Q_i. \, t_i < \mathit{max}(Q_i) \ \rightarrow \ \exists \, t_e \in Q_e. \, t_i \leq t_e \leq t_i + \Delta_{rcv} \\
\phi_{ei} \ \triangleq \ \forall \, t_e \in Q_e. \, \exists \, t_i \in Q_i. \, t_i \leq t_e \leq t_i + \Delta_{rcv}
\end{array}
$$

The following auxiliary invariants helped to prove $\psi_{nonlossy}$.

(1) $max(Q_i) = x_{inp}$

(2) $Q_i \neq \emptyset \rightarrow min(Q_i) \geq \delta_{snd}$

(3) $\forall\, t_i \in Q_i.\, (t < max(Q_i) \rightarrow \exists\, t_i'.\, (cons(Q_i, t_i, t_i') \ \wedge\ t + \delta_{snd} \leq t_i'))$

(4) $\forall\, t_i \in Q_i, t_e \in Q_e.\, T \geq t_i \ \wedge\ T \geq t_e$

(5) $(Q_i \neq \emptyset \ \wedge\ Q_e \neq \emptyset \ \wedge\ max(Q_e) < max(Q_i)) \rightarrow T \leq max(Q_i) + \Delta_{rcv}$

(6) $(last = ibit) \leftrightarrow (max(Q_i) \leq max(Q_e))$

(7) $(last = \neg\, ibit) \rightarrow T \leq max(Q_i) + \Delta_{rcv}$

Intuitively, these assertions can be explained as follows.

(1) The value of x_{inp} always correspond to the last timestamp stored in Q_i.

(2) The first timestamp stored in Q_i is greater than or equal to δ_{snd}.

(3) The difference between any two consecutive timestamps in Q_i is greater than or equal to δ_{snd} (because of the input rate).

(4) Timestamps are always less or equal to the current time.

(5) If the last timestamp of Q_i is greater than the last timestamp of Q_e (meaning that the last input message has not been put into the queue yet), then the difference between the current time and the last input time must be less than or equal to Δ_{rcv}.

(6) The values of *last* and *ibit* are equal if and only if the last timestamp in Q_e is greater than or equal to the last timestamp in Q_i (i.e., every input message has been put into the queue).

(7) If the values of *last* and *ibit* differ, then the difference between the current time and the last input time must be less than or equal to Δ_{rcv} (i.e., there is an input message waiting to be put into the queue).

Fischer's Mutual Exclusion Protocol

Here we model a simplified version of the well-known Fischer's protocol. This DTA specification is inspired by a similar example in Abadi and Lamport [1]. The protocol guarantees mutual exclusion for a number of processes executing the following code, where $x \in \mathbb{N}$ is a variable shared among all processes, and $i \in \mathbb{N}$ is an index uniquely identifying the process.

$test_i$: `await` $\langle x = 0 \rangle$;

set_i: $\langle x := i \rangle$;

$check_i$: `await` $\langle x = i \rangle$;

$crit_i$: (critical section)

The process i waits until $x = 0$ ($test_i$); then sets x to its process identifier at most Δ_{test} time units later (set_i); then waits for at least δ_{check} time units and checks the value of x again ($check_i$). If $x = i$, the process may enter it critical section ($crit_i$).

Processes have only one chance to enter the critical section, and that process i will remain deadlocked in $check_i$ if some other process j updated the shared variable x between set_i and $check_i$. We could easily add repetitive behaviour (e.g., as in the TIOA specification of [92], the code could be included in a loop, and processes may go back to $test_i$ if $check_i$ fails), but this specification suffices to illustrate the timing constraints of the protocol.

Network: $|\langle Proc_1, Proc_2 \rangle$
$V_S : \{ x \in \mathbb{N} \}$
$\Theta_S : x = 0$

DTA: $Proc_i$
$V_L : \{ pc_i \in \{0,1,2,3\}, tt_i, tc_i \in \mathbb{N} \}$
$\Theta_L : pc_i = 0 \ \wedge \ tt_i = 0 \ \wedge \ tc_i = 0$
$TL : \{ test_i, \ set_i, \ check_i \}$
$\mathcal{A} : test_i$
$prec :$ $x = 0 \ \wedge \ pc_i = 0$
$\mathit{eff} :$ $x' = 0 \ \wedge \ pc_i' = 1 \ \wedge \ tt_i' = T$
set_i
$prec :$ $pc_i = 1$
$deadline :$ $prec \ \wedge \ tt_i + \Delta_{test} \leq T$
$\mathit{eff} :$ $x' = i \ \wedge \ pc_i' = 2 \ \wedge \ tc_i' = T$
$check_i$
$prec :$ $x = i \ \wedge \ pc_i = 2 \ \wedge \ tc_i + \delta_{check} \leq T$
$\mathit{eff} :$ $pc_i' = 3$

Figure 8.13: A Process in Fischer's Protocol

Figure 8.13 shows part of our DTA specification of Fischer's protocol, for two processes (the definitions of automata $Proc_1$ and $Proc_2$ are symmetric, so we show just the template). Unfortunately, a DTA specification parameterised on the number of processes is not possible. Even though an unbounded array of program counters could be represented, we also need to represent unbounded arrays of time capture variables. The problem is that the maximum value of time capture variables is not known: MONA can represent unbounded arrays only if the element type is in a bounded domain [148].

It is known that mutual exclusion (i.e., that no two process can ever be in their critical sections at the same time) can be guaranteed if $\Delta_{test} < \delta_{check}$. We have used the constants $\Delta_{test} = 3$ and $\delta_{check} = 5$. Variables pc_i, tt_i and tc_i denote the program counter, and the time capture variables to enforce the Δ_{test} and δ_{check} delays, respectively. The program counter ranges over $\{0, 1, 2, 3\}$, denoting that the process i is in $test_i$, set_i, $check_i$ or in the critical section, respectively. Mutual exclusion, then, can be expressed as the safety property,

$$\psi_{mutex} \triangleq \Box\neg\,(pc_1 = 3 \ \wedge \ pc_2 = 3)$$

The following are auxiliary invariants that proved helpful in the verification of ψ_{mutex}.

(1) $T \geq tt_1 \ \wedge \ T \geq tc_1 \ \wedge \ T \geq tt_2 \ \wedge \ T \geq tc_2$

(2) $(pc_1 = 1 \ \rightarrow \ T \leq tt_1 + \Delta_{test}) \ \wedge \ (pc_2 = 1 \ \rightarrow \ T \leq tt_2 + \Delta_{test})$

(3) $(pc_1 = 3 \ \rightarrow \ T \geq tc_1 + \delta_{check}) \ \wedge \ (pc_2 = 3 \ \rightarrow \ T \geq tc_2 + \delta_{check})$

(4) $(pc_1 > 1 \ \rightarrow \ tt_1 \leq tc_1) \ \wedge \ (pc_2 > 1 \ \rightarrow \ tt_2 \leq tc_2)$

(5) $x = 0 \leftrightarrow (pc_1 < 2 \ \wedge \ pc_2 < 2)$

(6) $pc_1 = 3 \ \rightarrow \ (x = 1 \ \wedge \ tc_1 \geq tc_2 \ \wedge \ tc_1 \geq tt_2)$

(7) $pc_2 = 3 \ \rightarrow \ (x = 2 \ \wedge \ tc_2 \geq tc_1 \ \wedge \ tc_2 \geq tt_1)$

(8) $(tc_1 > tc_2 \ \rightarrow \ x = 1) \ \wedge \ (tc_2 > tc_1 \ \rightarrow \ x = 2)$

(9) $(pc_1 = 2 \ \rightarrow \ tc_1 \geq tt_2) \ \wedge \ (pc_2 = 2 \ \rightarrow \ tc_2 \geq tt_1)$

Assertion (1) denotes that the values of time capture variables cannot be greater than the current time. A process (2) cannot take more than Δ_{test} time units to set the variable; (3) it cannot enter the critical region before δ_{check} time units have passed since it set the variable; and (4) it tests the variable before it sets it. The variable keeps its initial value if and only if no process has yet set it (5). If $Proc_i$ is in its critical section, then $x = i$ and the other process can no longer set the variable, or pass the test (6-7). If $Proc_i$ was the last to set x, then $x = i$ (8). If a process has set the variable, the other process can no longer pass the test (9).

8.5 An Evaluation of Discrete Timed Automata

Discrete Timed Automata address limitations of timed automata and model-checkers; these concern data support (Section 2.4) and prevention of timelocks (Chapter 3). In addition, compared to PITL (Chapter 7), Discrete Timed Automata presents advantages as a specification language for real-time systems. This section also reveals limitations of Discrete Timed Automata.

8.5.1 Discrete Timed Automata vs Timed Automata

Data Support. WS1S is an expressive and natural notation, where certain unbounded data structures can be represented intuitively by finite sets. For instance, Section 8.4.3 studied DTA specifications where unbounded buffers were represented (importantly, MONA allowed us to automatically validate assertions that involved quantification and sets). As we discussed in Section 2.4, unbounded structures (and quantification and sets) are beyond the scope of timed automata and model-checkers, and so Discrete Timed Automata allow a more accurate mapping between systems and specifications.

Structured types such as enumerations, records, lists and arrays (where elements in the collection are in bounded finite domains) can also be expressed in WS1S [29, 148]. In addition, WS1S supports a form of parameterised specifications, where parameters are treated as uninitialised rigid variables. This is possible to the extent that the resulting parametric formulae are expressible in WS1S. For example, if $\Delta \in \mathbb{N}$ is a parameter and $v \in \mathbb{N}$ is a variable in the DTA, then an action can assign $v' = \Delta + 1$, but not $v' = \Delta + v$ or $v' = \Delta v$, because general addition and multiplication are not expressible in WS1S. On the other hand, real-valued clocks, negative integers, and constraints such as $x - y \leq 2v$ (where x, y are clocks and v is an integer variable) are expressible in Uppaal (Chapter 2), but cannot be expressed in DTAs.

In DTAs, constant time-bounds can be expressed using time capture variables. For example, if an action must be executed in the time interval $[low, up]$, with constants $low, up \in \mathbb{N}$, this can be modelled in the DTA by (a) declaring a time capture variable $c \in \mathbb{N}$, (b) adding $T \geq c + low$ to the action's precondition, and (c) adding $T \geq c + low \ \wedge \ T \geq c + up$ to the action's deadline (c is reset by adding $c' = T$ to effect formulae). Unlike in Uppaal, low and up cannot denote variables in DTAs (terms in WS1S cannot contain more than two variables). However, variable bounds are possible if we extend DTAs with clock variables; i.e., variables $T_1, T_2, \ldots, T_n$ that are implicitly (and simultaneously) incremented by the passage of time (as in Clock Transition Systems [110]). In such an extension of DTAs, we could declare a clock variable T_c and add $T_c \geq low$ to the action's precondition, and $T_c \geq low \ \wedge \ T_c \geq up$ to the action's deadline (T_c is reset by adding $T_c = 0$ to effect formulae).

Prevention of Time-actionlocks. In Discrete Timed Automata, deadlines express urgency and imply preconditions, and urgency is not enforced on input or output actions unless synchronisation is possible. This makes DTAs time reactive by construction; thus well-timed specifications are easier to obtain than in timed automata and most other deductive frameworks (as we explained in Chapter 3, prevention of timelocks is not possible in timed automata).[10]

[10] Timed I/O Automata with Urgency [69] is the only other time reactive deductive notation.

In addition, we believe that timelock-freedom may be easier to prove in Discrete Timed Automata than in other deductive frameworks: since time-actionlocks cannot occur, absence of Zeno-timelocks (or absence of Zeno runs) guarantees timelock-freedom.

Time models. DTAs assume a discrete-time model; in this sense, DTAs are less expressive than timed automata. It is generally accepted that dense time has a number of advantages over discrete time; e.g., the possibility to express arbitrarily small delays [5]. Therefore, if dense-time models are to be represented and verified with DTAs, digitization arguments have to be checked both for the system and the property to verify [83, 34].

Nonetheless, the semantics of DTAs and timed automata share a number of properties.[11] Let $\delta, \delta' \in \mathbb{R}^+$ denote delays (resp. $\delta, \delta' \in \mathbb{N}$ for discrete time), and s, s', s'' denote reachable states in the semantics. *Time determinism* holds if delays yield unique states. *Timed additivity* holds if, whenever time can pass by two consecutive delays δ and δ', then it can also pass by a $(\delta + \delta')$-delay. *Time interpolation* holds if, whenever a $(\delta + \delta')$-delay is possible, then time can also pass by two consecutive delays δ and δ'. *Time closure* applies only for dense-time domains; time closure holds if, for any δ such that any δ'-delay is possible, with $\delta' < \delta$, then a δ-delay is also possible.

$$
\begin{array}{ll}
\text{(TDet)} & \forall\, \delta, s, s', s''.\, ((s \stackrel{\delta}{\Longrightarrow} s' \ \wedge\ s \stackrel{\delta}{\Longrightarrow} s'') \ \Rightarrow\ (s' = s'')) \\
\text{(TAdd)} & \forall\, \delta, \delta', s, s', s''.\, ((s \stackrel{\delta}{\Longrightarrow} s' \stackrel{\delta'}{\Longrightarrow} s'' \ \Rightarrow\ (s \stackrel{\delta+\delta'}{\Longrightarrow} s'')) \\
\text{(TInt)} & \forall\, \delta, \delta', s, s''.\, ((s \stackrel{\delta+\delta'}{\Longrightarrow} s'') \ \Rightarrow\ \exists\, s'.\, s \stackrel{\delta}{\Longrightarrow} s' \stackrel{\delta'}{\Longrightarrow} s'') \\
\text{(TClos)} & \forall\, \delta.\, ((\forall \delta' < \delta.\, s \stackrel{\delta'}{\Longrightarrow}) \ \Rightarrow\ s \stackrel{\delta}{\Longrightarrow})
\end{array}
$$

Time determinism, time interpolation and time additivity hold both in timed automata and in DTAs. Timed automata do not satisfy time closure, because right-open invariants are allowed (e.g., $x < 1$).

8.5.2 Discrete Timed Automata vs PITL

The adoption of an infinite-time model, the use of WS1S to represent data, and an "action-oriented" style of specification, are features that allow Discrete Timed Automata to overcome some of the limitations of PITL as a specification language (Chapter 7). The benefits related to infinite-time and WS1S data types in DTAs, compared with the restrictions of finite-time and propositional variables in PITL, are clear. On the other hand, the issue of framing is worth discussing in some more detail.

[11] These properties are typical of analyses of timed transition systems; see e.g., [129, 168, 144].

In PITL, the truth value of a propositional variable is not preserved from one state in the interval to the next. However, this assumption is usually needed when specifying systems; thus, in such cases, PITL specifications must explicitly assert or negate propositions (Section 7.5). On the other hand, DTAs can deal with the issue of framing in an elegant way. By way of example, consider a DTA with variables $\mathcal{V} = \{x, y\}$ and an action *act* with effect:

$$e \triangleq x' = x + 1$$

We do not specify what happens to y when *act* is performed, but we assume the persistence of values for all those variables that are not modified by e; i.e., we assume that $y' = y$. Furthermore, we do not need to explicitly state $y' = y$ when we perform invariance proofs. Suppose that we need to validate a certain proof obligation (generated by the invariance rule):

$$(\phi \ \wedge \ x' = x + 1) \Rightarrow \phi'$$

There are two cases to consider, but in both cases we avoid stating $y' = y$ explicitly: (a) if ϕ does not refer to y, then the value of y' is clearly irrelevant, otherwise (b) ϕ refers to y, in which case we can safely replace y for y' in ϕ' (we have assumed that $y' = y$).

Another limitation of PITL is that consecutive states in an interval necessarily denote a 1 time unit delay. As a result, instantaneous actions (the typical assumption in real-time notations) cannot be expressed in PITL. In addition, interleaving cannot occur unless actions yield consistent next states (i.e., if an action a changes the value of proposition P from *true* to *false*, then any other action that occurs concurrently with a must yield a next state where P is *false*). In contrast, Discrete Timed Automata assume a model with instantaneous actions, and interleaving is independent of the effect of actions.

Finally, consider that operators such as chop, chop-star and projection, and past operators, give PITL flexibility to express complex requirements that refer to sequences, iteration or past events. In DTAs, requirements are represented by LTL formulas (where modal operators are more restrictive than in PITL), although state formulae are not limited to propositional variables (instead, they may be any WS1S formulae).

8.6 Related Work

Smith and Klarlund [148] use (untimed) I/O Automata [109] to model a sliding window protocol, and MONA to assist the verification of safety properties via invariance proofs. Our work on Discrete Timed Automata can be considered an extension of these ideas to real-time systems.

Timed I/O Automata [92, 69], Clock Transition Systems [93],[12] and the Temporal Logic of Actions [1], are deductive frameworks to reason about real-time systems. These notations assume a dense-time domain, and share other elements. Specifications may be parameterised, and consist of a finite set of variables and actions, where actions are defined through preconditions and postconditions.

These notations adopt first-order logic as the assertion language. A comparison between first-order logic and WS1S (the assertion language of Discrete Timed Automata) is beyond the scope of this thesis; let us just mention that both assertion languages present advantages, and that there are properties that may be expressed in one logic but cannot be expressed in the other. For instance, first-order logic permits more general arithmetic operations; however, the logic is undecidable and tool support is confined to theorem provers (although, theorem provers may include decision procedures that will automatically verify certain fragments of the assertion language). On the other hand, WS1S may express second-order quantification, and MONA implements an efficient decision procedure for WS1S.

Timed I/O Automata, Clock Transition Systems and the Temporal Logic of Actions partition the set of actions into *controlled* actions, which are performed by the system, and *external* actions, which are performed by the environment. This facilitates the analysis of so-called open systems (see, e.g., [63]). Communication in these theories is realised through shared variables and multiway synchronisation. In contrast, DTAs model closed systems, where the environment must be represented by components in the network. Only binary synchronisation has been defined for DTAs, but other modes (including multiway synchronisation) can be accommodated by redefining the product automaton. Note, these design decisions affect the way in which DTAs behave, but are orthogonal to the use of WS1S and MONA. We finish this section by comparing Discrete Timed Automata against particular elements of each theory.

Timed I/O Automata. In Timed I/O Automata (TIOAs), the evolution of variables over time can be either discrete or continuous. Discrete changes are expressed by actions. The passage of time is represented by special functions called trajectories, which describe the continuous evolution of dynamic variables over intervals of time (TIOAs support the definition of certain classes of hybrid systems; clocks are just a particular class of dynamic variables).

Urgency is represented by time-stopping conditions over trajectories, with a semantics similar to deadlines and invariants. However, from a pragmatic point of view, stopping conditions are more difficult to use, and may lead more easily to timelocks (this is because the stopping conditions are related to trajectories, and so urgency of actions is imposed indirectly).

[12] Clock transition systems are extended with modules in [25].

Trajectories satisfy certain axioms, including time reactivity; however (unlike in Discrete Timed Automata), the theory does not specify the means to obtain time reactivity by construction. Nonetheless, the theory identifies subclasses of TIOAs that are free from timelocks (*feasible* and *receptive* TIOAs), and free from Zeno runs (*progressive* TIOAs). These notions are defined in the context of open systems, e.g., a TIOA is considered free from Zeno runs or timelocks *assuming* that the environment does not engage in such abnormal behaviours (see [92] for detailed definitions).

Verification in TIOAs is currently supported by the theorem prover PVS [107]. Robson [139] implemented a translator from a subclass of TIOAs to Uppaal models.[13] TIOAs with Urgency [69] are an extension of TIOAs with deadlines attached to actions, similar to DTAs (our theories were developed independently), and time reactivity is guaranteed by construction. To our knowledge, mechanisation of TIOAs with urgency has not been investigated.

Clock Transition Systems. Clock Transition Systems (CTSs) include a set of clocks, which are real-valued variables that increment uniformly with the passage of time. In this sense, clocks in CTSs behave as clocks in timed automata, rather than time capture variables as in DTAs.

A CTS includes an explicit time action, and a time-progress condition to specify urgent behaviour. The time action increments all clocks by some δ-delay, provided no δ'-delay ($\delta' < \delta$) invalidates the time-progress condition. As in TIOAs, urgency is modelled indirectly.

Bjorner et al. [25] propose a deductive method to check timelock freedom in CTSs. The method characterises a sufficient-only condition for the absence of timelocks, in terms of first-order logic proof obligations and ranking functions on well-founded domains. In addition, a similar method is proposed to check receptiveness (as in TIOAs, this is a compositional property that guarantees timelock-freedom in open systems). Deductive verification of LTL formulae is supported by the STeP tool.[14] STeP includes tools for the automatic generation of invariants, and also assists the verification of timelocks [25].

The Temporal Logic of Actions. Specifications in the Temporal Logic of Actions represent the passage of time using a time variable *now* (as we do with T in DTAs), and lower and upper (urgent) time bounds are related with actions through *timers*. Timers are formulae that constrain the enabling conditions of actions to represent lower bounds, and the evolution of *now* to represent upper bounds. Based on timers, the authors give in [1] a number of sufficient-only conditions to guarantee timelock-freedom.

[13] See also `http://tioa.csail.mit.edu/`
[14] `www-step.stanford.edu/`

Verification of TLA specifications is supported by theorem provers, such as the Larch Prover [67], HOL [163] and Isabelle [90]; the model-checker TLC supports automatic verification of finite-state TLA models [167].[15] Lamport [101] gives a method to ensure timelock-freedom in finite-state, discrete-time TLA specifications. Briefly, Lamport's check verifies a condition similar to the TCTL formula $\forall\Box\exists\Diamond_{=1}\, true$.

8.7 Conclusions

We presented Discrete Timed Automata as a notation to specify real-time systems, and used MONA to assist invariance proofs. To do this, we mapped DTAs to basic transition systems, in order to apply the invariance rule [112]. The proof obligations that result from the invariance rule can be automatically validated in MONA, which facilitates the verification of complex requirements (e.g., MONA is able to validate formulae involving arithmetic operations, set operations and quantification).

DTAs consist of a set of variables whose possible valuations represent the state space, and a set of actions to describe discrete changes. Actions are defined through preconditions and effects (i.e., postconditions) in the usual way. A time variable, T, represents the value of the current time; we assume a discrete time domain, and so $T \in \mathbb{N}$ (discrete time is necessary to express DTAs in MONA). Urgency can be modelled by attaching deadlines to actions; deadlines are formulae that impose certain maximal valuations for T, thereby forcing the immediate execution of the corresponding action. Importantly, the semantics of deadlines is such that actions can be made urgent only when they are enabled; in addition, this semantics is preserved by synchronisation. Therefore, networks of DTAs are time reactivity by construction (i.e., networks of DTAs are free from time-actionlocks).

This chapter concludes Part III and our study of WS1S and MONA in the context of real-time systems. Discrete Timed Automata provide solutions to some limitations of timed automata and model-checkers (Chapter 2) , regarding data support and prevention of timelocks; and PITL (Chapter 7), regarding the specification of real-time systems.

In Discrete Timed Automata, a class of unbounded data structures can be represented and analysed; time-actionlocks are prevented by construction; and framing is easily accommodated in specifications (and proofs). As a result, certain systems can be more accurately represented (e.g., in communication protocols, DTA specifications do not need to bound buffers to any predefined capacity), and well-timed specifications are easier to obtain in Discrete Timed Automata than in most other notations (including Timed Automata). Moreover, because time-actionlocks cannot

[15] See also `http://research.microsoft.com/users/lamport/tla/tla.html`

occur in DTAs, the proofs of timelock-freedom can be reduced to proofs for absence of Zeno-timelocks (or absence of Zeno runs), which may facilitate the task considerably.

Therefore, Discrete Timed Automata (and in turn, WS1S and MONA) present solutions to some of the reliability issues that this thesis has tried to address (Chapter 1). In addition, we have revealed some limitations of Discrete Timed Automata, due to the restrictive arithmetic of WS1S (we compared Discrete Timed Automata, Timed Automata and PITL, in Section 8.5).

We conclude this chapter by identifying possible ways in which our work could be extended. This demonstrates the flexibility of Discrete Timed Automata to accommodate new elements and tool support.

Checking Timelock-freedom. There are a number of alternatives to investigate. Following Robson's work on TIOAs [139], a translation from finite-state DTAs to Uppaal timed automata would allow us to apply the detection methods discussed in Chapter 3. We could also derive, from Abadi and Lamport's non-Zenoness theorem for TLA specifications [1], similar conditions that apply to DTAs (the theorem in [1] is formulated for timers, instead of deadlines). A third alternative is to adapt to DTAs the non-Zenoness conditions proposed by Bjorner et al. [25] for Clock Transition Systems.

Tool Support. Deriving BDD variable-ordering heuristics from DTA specifications may improve the performance of MONA. Built-in high-level data-types (e.g., bounded sequences or enumerated types [29, 148]), and tools to automatically generate invariants (e.g., [27]), will facilitate specification and verification of DTAs. Additionally, it would be interesting to investigate the mechanisation of DTAs using tools other than MONA. For example, translating DTAs to TLA specifications may allow mechanisation in HOL [163] and Isabelle [90], and model-checking in TLC for finite-state specifications [167]; similarly, translating DTAs to TIOAs may allow mechanisation in PVS [107], and Uppaal [139] for finite-state specifications. The PAX tool [13] may be used for parametric verification.

LTL Properties. We have proved safety properties of the form $\Box\,\phi$, where ϕ is a state formula (in our case, a WS1S assertion on the automaton's variables), by using the invariance rule. Other deductive rules are known that infer a more general class of LTL safety properties, where ϕ denotes an LTL past formula [112]. These include *waiting-for* properties, $\Box\,(p \rightarrow q\,\mathcal{W}\,r)$ (q holds since p holds, until r holds), and *causality* properties, $\Box\,(p \rightarrow \diamondsuit\!\!\!-\, q)$ (if p holds then q must have held somewhere in the past). These deductive rules also reduce temporal reasoning to checking the validity of a number of state assertions, enabling MONA to assist the proofs.

The deductive verification of (a class of) liveness properties is also possible in DTAs. Deductive rules to verify response properties, $p \Rightarrow \Diamond q$ (whenever p holds, q eventually holds), are given by Kesten et al. [93] for clock transition systems. A mapping from DTAs to clock transition systems would allow us to apply these rules in our framework, and use MONA to check the proof obligations. Other deductive rules (similar to those in [93]) were considered by Henzinger et al. [84] and may be applicable to DTAs (the computational model [84] is closer to Discrete Timed Automata than are Clock Transition Systems, since it assumes discrete time and employs a single time variable *now*) .

Other Extensions. We have mapped Discrete Timed Automata to Basic Transition Systems, which proved to be sufficient to verify simple safety properties. In addition, we may consider extending Discrete Timed Automata with fairness assumptions (as in Fair Transition Systems [112]), clock variables (as in Clock Transition Systems), and timers (as in TLA).

Part IV

Conclusions

Chapter 9

Conclusions

We explained that limitations of timed automata [7] and real-time model-checkers undermine the confidence of users in formal verification. Our search for solutions to these reliability issues resulted in improvements to Uppaal [22] (Chapter 2), and in the evaluation of WS1S and MONA [96] (Chapter 6) in real-time settings. In general, we have improved the available tool support for formal verification of real-time systems.

Timelocks and Zeno runs represent abnormal behaviours in the semantics of a timed automaton. Unless specifications are guaranteed to be timelock-free (and sometimes, also free from Zeno runs), users cannot trust in the verification of safety or liveness properties. We improved the support offered by model-checkers (and in particular, Uppaal) for detecting timelock-freedom and absence of Zeno runs. We improved existing methods, developed new ones, and implemented them in a tool to check the well-timedness of Uppaal specifications (Chapters 3, 4 and 5).

Model-checkers, whose requirements languages are based on branching-time logics, cannot adequately express requirements that refer to sequence and iteration of events, or past computations. This is the case for Uppaal, where requirements are expressed in a subset of CTL. PITL (a propositional subset of Interval Temporal Logic [120]) is a linear-time logic with past operators and the power of regular expressions, which allows sequences, iterations and past computations to be naturally modelled. Therefore, as a requirements language, PITL overcomes a number of expressiveness limitations of the logics of Uppaal and other real-time model-checkers.

However, PITL has a non-elementary worst-case complexity, which has hampered the development of efficient tool support. We proposed to benefit from MONA, which has been successful in dealing with applications of WS1S (despite its non-elementary complexity). We developed a translation from PITL to WS1S, so that MONA could be used as a decision procedure for PITL. Thanks to the many optimisations included in MONA, we obtained a practical and efficient decision procedure for PITL (Chapter 7).

Timed automata cannot represent infinite-state systems, and model-checkers restrict data types to bounded domains. This limits the analysis of certain systems, such as communication protocols with unbounded buffers. In addition, timelocks cannot be prevented by construction in timed automata. These limitations motivated our development of Discrete Timed Automata, a deductive notation based on WS1S, where MONA is used to mechanise invariance proofs. We have shown that certain unbounded data structures could be naturally represented in WS1S, and analysed with MONA. Moreover, Discrete Timed Automata are time reactive by construction: time-actionlocks cannot occur, and so, well-timed specifications are much easier to obtain than in Timed Automata and other notations (Chapter 8).

We have evaluated our solutions in a number of case studies. Importantly, the CSMA/CD protocol (to test our timelock detection methods), the Symphony Problem (to test PITL and our MONA-based decision procedure), and the Multimedia Stream (to test Discrete Timed Automata) are well-known benchmarks in the real-time verification literature.

In this chapter we conclude our thesis with an account of our contributions. In addition, we compare the different notations and tools studied in this thesis, and discuss further research.

9.1 Contributions

We improved the available tool support for the verification of real-time systems, aiming to increase the confidence users can obtain in their specifications and analyses. We made practitioners and researchers aware of reliability issues in real-time model-checkers: such issues have not been addressed before to the extent found in this thesis. This strengthens the use of formal methods to assert the correctness of complex systems.

In particular, our analysis focused on Uppaal, which is probably the most developed real-time model-checker (and the most extensively used). Limitations in methods to detect timelocks and Zeno runs, a restrictive branching-time requirements language, and data support limited to bounded domains, all attempt against the reliability of Uppaal (and model-checkers, in general).

In our search for solutions, we studied the applicability of WS1S and MONA in real-time specification and requirements languages. This analysis resulted in a number of benefits. Firstly, WS1S and MONA helped us to develop solutions to the problems found in Uppaal. Secondly, we extended the use of WS1S and MONA (and identified strengths and weaknesses) to a new application domain for which they were not originally developed; i.e., the formal verification of timed systems. Finally, we were able to compare different notations, such as Timed Automata, PITL and Discrete Timed Automata, and different tools, such as Uppaal and MONA. This comparison is novel, and reveals useful links between timed and untimed methods (see Section 9.2).

The transition between different notations and tools is reflected in this thesis, from Part II, which studied Timed Automata and Uppaal (Chapters 2, 3, 4 and 5), to Part III, which studied WS1S, MONA, PITL and Discrete Timed Automata (Chapters 6, 7 and 8). As we explain next, this transition has been motivated by limitations found in notations and tools.

1. We started by studying timelocks and Zeno runs in timed automata, and we improved the detection methods available in model-checkers such as Uppaal and Kronos (Chapters 2, 3, 4 and 5). Timelocks and Zeno runs are an important source of unreliability in formal verification, but there are others. In particular, certain systems and requirements are difficult to express in timed automata and the branching-time logics of model-checkers. These limitations motivated our work on WS1S and MONA, PITL and Discrete Timed Automata.

2. PITL can naturally express requirements that refer to quantitative time constraints, sequences and iteration of events, and past computations. In this respect, PITL solves some of the expressiveness limitations of requirements language for model-checkers. Unfortunately, PITL has a non-elementary complexity, which makes it difficult to apply. This motivated the use of MONA (Chapter 6) as an efficient decision procedure for PITL (Chapter 7). Furthermore, this was justified by the success of MONA in dealing with WS1S-based applications, despite its non-elementary complexity.

3. On the other hand, as a specification language, PITL presents a number of shortcomings: data are restricted to propositional variables, and the interleaving of actions and time delays is more difficult to express than in behavioral notations. These limitations of PITL, the limitations related to bounded data types supported by model-checkers, and the difficulty of detecting and preventing time-actionlocks in timed automata, motivated the use of WS1S and MONA to develop Discrete Timed Automata (Chapter 8). WS1S allows Discrete Timed Automata to represent unbounded data structures, MONA assists in the proofs of complex requirements (e.g., those which arise when we must reason about unbounded data structures), and time reactivity by construction (i.e., prevention of time-actionlocks) is obtained by the semantics of urgency and synchronisation.

The following sections explain our contributions in more detail.

9.1.1 New Methods to Assert Absence of Timelocks and Zeno Runs

Timelocks (states where time cannot diverge) and Zeno runs (arbitrarily fast executions) are abnormal behaviours in the semantics of a timed automaton. Verification cannot be trusted unless specifications are guaranteed to be timelock-free; in some cases (e.g., for the verification

of certain liveness properties) absence of Zeno runs is an additional sanity requirement. On the other hand, there are cases where Zeno runs are perfectly acceptable, as long as the specification is timelock-free. Therefore, we need the means to assert timelock-freedom and absence of Zeno runs, and methods to assert timelock-freedom that are insensitive to Zeno runs.

However, only a few model-checkers provide support to assert the absence of timelocks and Zeno runs, and this support is limited in many ways. A liveness formula can be verified in Uppaal that denotes absence of timelocks and Zeno runs, but the algorithm involves a demanding nested reachability analysis, and rules out timelock-free specifications where Zeno runs occur.

In the case of model-checkers such as Kronos [169] and Red [165], timelock-freedom can be characterised more precisely than in Uppaal, but verification requires fixpoint algorithms (which can be computationally expensive). Another limitation of Kronos is that absence of Zeno runs cannot be checked, and the product automaton must be constructed before timelock-freedom can be checked. We note that both, Red and the tool Profounder [159] (which checks emptiness of Timed Büchi Automata) are able to detect Zeno runs and avoid them during verification, but these require demanding algorithms.

We studied the nature of timelocks and Zeno runs in detail (Chapter 3), and developed new methods to assert absence of Zeno-timelocks (an important class of timelocks) and Zeno runs. Our work improves and complements the existing tool support.

- We refined the application of strong non-Zenoness, a static check that guarantees absence of Zeno runs in networks of timed automata [156] (Chapter 4). We proved that, in order to guarantee absence of Zeno runs, (a) it is sufficient to check strong non-Zenoness just on simple loops, and (b) not every loop in the specification is required to be strongly non-Zeno (SNZ). In particular, we proved that a non-simple loop is SNZ if and only if any of its constituent simple loops is SNZ, and we proved that strong non-Zenoness is preserved by synchronisation (even if only one component loop is SNZ). Therefore, we provide a more efficient and comprehensive check for strong non-Zenoness. Thanks to our improvements, a class of systems that are free from Zeno runs, but where not every loop is SNZ, can now be positively analysed.

- We introduced new static checks to guarantee absence of Zeno-timelocks in a timed automaton, based on the syntax of invariants in simple loops. These checks recognise specifications that are free from Zeno-timelocks, even if they are not SNZ (Chapter 4).

- We developed semantic checks to detect Zeno-timelocks and Zeno-runs in a timed automaton, which are based on reachability formulae obtained from the syntax of loops in a timed automaton (Chapter 5). These checks can be conclusive in specifications where strong non-Zenoness, invariant-based properties, and Uppaal's check for timelock-freedom,

fail to hold. Interestingly, this is the first time that timelocks and Zeno runs have been characterised in terms of static analysis and reachability (note that, time-divergence is a liveness property).

- We implemented a tool that exploits the complementary nature of our static and semantic checks (Chapter 5). This tool receives an Uppaal specification as input (a network of timed automata), and performs our refined strong non-Zenoness check first, followed by strong non-Zenoness and invariant-based checks on the product automaton, and finally interfaces with Uppaal to verify reachability formulas (those that characterise the semantic checks). Importantly, this strategy applies efficient, sufficient-only checks first, and proceeds to more demanding, sufficient-and-necessary checks as required (i.e., only if previous checks have been inconclusive).

Zeno-timelocks cannot occur in specifications that are free from Zeno runs (by definition). Furthermore, provided the specification is deadlock-free, the absence of Zeno-timelocks (or the absence of Zeno runs) guarantees timelock-freedom. Typically, deadlock-freedom is asserted (and specifications are corrected accordingly, otherwise) before any other verification takes place.

Note that, strong non-Zenoness is static and compositional, and holds often in specifications. Therefore, in most cases, this property suffices to guarantee both absence of Zeno runs and timelock-freedom. In other words, we have proposed an efficient method that avoids the complexity of model-checking liveness properties (as done in Uppaal, Kronos and Red), but which can be equally effective for asserting the sanity of specifications.

In addition, strong non-Zenoness complements timelock detection in Kronos (which cannot assert absence of Zeno runs), and simplifies the verification in Red and Profounder (which must run demanding algorithms to avoid Zeno runs).

Our invariant-based checks guarantee absence of Zeno-timelocks but are insensitive to Zeno runs. Thus, we are able to guarantee timelock-freedom in (deadlock-free) specifications that Uppaal's check would classify as unsafe (Uppaal's check demands absence of Zeno runs).

Unlike Uppaal, Kronos and Red, which all depend on liveness properties to assert timelock-freedom, our static and semantic checks are available to all timed automata specifications, and (in the case of semantic checks) only require reachability analysis. Thus, our checks can be implemented in all real-time model-checkers.

Our checks improve the support provided by Uppaal, but they also complement it: in some cases, Uppaal's liveness formula may be the best solution. For instance, the invariant-based properties, and the semantic checks, require the product automaton to be constructed a priori (in Uppaal, on the other hand, verification is on-the-fly). Thus, Uppaal may be more efficient if a timelock is detected after exploring just a small part of the reachability graph. Another

advantage of Uppaal is that verification is performed over a richer specification language. For instance, Uppaal's timed automata may include variables and richer reset conditions, while strong non-Zenoness only takes into account zero-resets and guards with constant lower bounds.

9.1.2 An Efficient Decision Procedure for PITL

PITL is a propositional subset of ITL with quantification, future operators such as *len*, *chop*, *chop-star* and *projection*, and past operators such as *previous* and *since*. PITL can naturally model requirements that refer to quantitative time constraints, sequences and iteration of events, and past computations; hence, PITL presents advantages with respect to requirements languages based on branching-time logics (such as the CTL-based language of Uppaal, or the TCTL-based languages of Kronos and Red).

Importantly, PITL facilitates the mapping between requirements and logic formulae, which increases the confidence of users in verification: Validation problems are less likely to occur if formulae are easier to understand.

However, until now, practical tools to analyse PITL specifications have been difficult to obtain, due to the non-elementary (worst-case) complexity of PITL's decision procedure. Thanks to MONA, we believe we have contributed to improve this situation (Chapter 7).

- We demonstrated that WS1S is expressive enough to encode the semantics of PITL formulae, revealing a link between these two logics. Interestingly, our work provides yet another evidence of the power of WS1S. The fact that WS1S (with simple arithmetic and set operators) is able to express intervals, quantitative time constraints, concatenation, iteration, projection and past operators (as featured by PITL) is far from obvious.

- We translated PITL to WS1S, and used MONA as a decision procedure for PITL. We implemented a front-end, PITL2MONA, which takes a PITL formula as input, and produces a semantically equivalent MONA specification. To our knowledge, this is the first time an automata-based decision procedure has been implemented for PITL.

- Thanks to its many optimisations, MONA is an efficient decision procedure for PITL, despite the non-elementary complexity of this logic. We performed experiments that show that PITL2MONA+MONA outperforms LITE, a known tableau-based decision procedure for Propositional ITL (LITE does not implement past operators). Being based on BDDs, we think that MONA would provide even better performance if variable-ordering heuristics are implemented.

We believe that our work on PITL is a helpful step to obtaining real-time model-checkers with more expressive requirements languages. Indeed, this aim is being pursued by industry for

untimed model-checkers; for instance, IBM's model-checker RuleBase includes regular expression operators in its requirements language, PSL/Sugar [14].

9.1.3 Discrete Timed Automata

We introduced Discrete Timed Automata as a deductive notation to specify real-time systems (Chapter 8). WS1S is the assertion language, which is used to define actions and typed variables. In addition, this allows invariance proofs to be mechanised using MONA. Note that, MONA facilitates complex proofs; for instance, MONA is able to automatically validate rich WS1S formulae involving arithmetic operations, set operations and quantification.

Urgency in Discrete Timed Automata is expressed through deadline predicates attached to actions, such that (a) only enabled actions may be urgent, and (b) synchronisation may be urgent only if both parties are enabled. Thus, the semantics of urgency and synchronisation make Discrete Timed Automata time reactive by construction (time-actionlocks cannot occur).

With such distinctive features, Discrete Timed Automata are able to overcome some reliability issues that arise in timed automata and model-checkers.

- Discrete Timed Automata can deal with infinite-state systems, and the representation and analysis of unbounded data structures is possible thanks to WS1S and MONA. For instance, unbounded buffers, and buffers where elements can range over $\mathbb{N}$ (which are common in specifications of communication protocols) can be naturally modelled in Discrete Timed Automata. Note that, such structures must be mapped by finite abstractions in Uppaal (and other model-checkers), hence the analysis loses generality.

- Time-actionlocks are ruled out by construction in Discrete Timed Automata. This has two important benefits. Firstly, well-timed specifications are easier to obtain than in other notations (including Timed Automata, where prevention of timelocks is not possible). Secondly, in Discrete Timed Automata, proving absence of Zeno runs (or absence of Zeno-timelocks) suffices to guarantee timelock-freedom.

As a specification language for real-time systems, Discrete Timed Automata also present advantages with respect to PITL. Unlike in PITL, the interleaving of actions and time is natural in the semantics of Discrete Timed Automata, and the framing of variables is easily accounted for in specifications and proofs.

Discrete Timed Automata also inherit expressiveness limitations from WS1S. The lack of general addition and multiplication in WS1S may prevent certain specifications from being represented (and analysed) in Discrete Timed Automata. WS1S also imposes a discrete-time model for Discrete Timed Automata: the logic is interpreted over $\mathbb{N}$.

Importantly, our work on Discrete Timed Automata also serves as a study of the suitability of WS1S and MONA in real-time settings. This is the first time that WS1S and MONA have been proposed for the formal verification of real-time systems; we have shown that many positive aspects result when they are integrated in a deductive framework.

9.2 Comparing Different Notations and Tools

We have studied different formalisms and tools: Timed Automata and Uppaal, WS1S and MONA, PITL and Discrete Timed Automata. This section highlights key relationships among these methods (see also our discussions in Sections 2.4, 7.4.2 and 8.5).

- PITL can naturally express sequences and iteration of events, and past computations. These are concepts that are difficult (or even impossible) to express in Uppaal (which verifies a subset of CTL) or DTAs (which verify LTL invariance properties). Data variables, however, are limited to propositions.

- Real-valued clocks, negative integers, arrays of clocks, and constraints such as $x - y \leq 2v$ (where x, y are clocks and v is an integer variable), are all expressible in Uppaal but not in WS1S. On the other hand, Uppaal does not support unbounded sets of natural numbers, or first-order and second-order quantification, which are all expressible in WS1S. Note that, structured types such as enumerations, records, lists and arrays, where elements in the collections are in known-size finite domains, can also be expressed in WS1S [29, 148].

- The time model is infinite and dense in Timed Automata, infinite and discrete in Discrete Timed Automata (due to WS1S being interpreted over $\mathbb{N}$), and finite and discrete in PITL (due to models being finite sequences of states). In this respect, Timed Automata are more expressive than Discrete Timed Automata, which in turn are more expressive than PITL.

- The interleaving of actions and time is restricted in PITL (actions are not a primitive concept in PITL). Actions must have a minimal duration (the delay occurring between consecutive states in an interval), and interleaving cannot occur unless the involved actions yield consistent next states. For instance, if an action a changes the value of proposition P from *true* to *false*, then any other action that occurs concurrently with a must yield a next state where P is *false*. On the other hand, actions are instantaneous in Timed Automata and Discrete Timed Automata, and interleaving of actions is independent of the resulting next state.

- Framing of variables must be taken into account in PITL specifications: values are not necessarily preserved from one state to the next. This is not an issue in Timed Automata

(model-checkers preserve values during the exploration of the state space, unless these are explicitly changed by resets) or in Discrete Timed Automata (framing is handled implicitly in the interpretation of primed variables).

- Discrete Timed Automata prevent time-actionlocks by construction; this is not possible in Timed Automata. On the other hand, Zeno runs and Zeno-timelocks may occur both in Timed Automata and Discrete Timed Automata.

9.3 Further Research

We conclude this thesis by suggesting possible extensions to our work. Some of these ideas are explored in more detail in the conclusions of Chapters 4, 5, 7 and 8.

Absence of Zeno Runs and Zeno-timelocks in Timed Automata. Our checks for Zeno-runs and Zeno-timelocks may be extended to deal with a richer timed automata model. In particular, a good target is the specification language of Uppaal, which supports non-zero resets, data variables, urgent channels and C-like functions in transitions. In addition, the compositional nature of strong non-Zenoness can be further exploited (we presented and example of this in Section 4.3).

Tool Support for ITL Specifications. Variable-ordering heuristics may be derived from PITL formulae, to better exploit the efficiency of BDDs (and avoid worst-case scenarios as much as possible).

The translation from PITL to WS1S may be integrated in theorem provers to improve mechanisation for ITL proofs (through MONA). This research may benefit from previous work done on mechanisation of ITL in PVS [55], and of WS1S in PVS [131] and Isabelle [11].

Operators such as chop and chop-star, and past operators, may be used to enhance the requirements languages of real-time model-checkers; e.g., integrating chop into Uppaal's requirements language would allow sequences to be expressed naturally. In this respect, the work of Pandya on extending CTL with QDDC formulae is relevant [135].

Discrete Timed Automata. Proofs to assert timelock-freedom have been developed for frameworks such as TLA [1] and Clock Transition Systems [93]; these proofs could be adapted to DTA specifications, and may be simplified because DTAs are time reactive (thus, absence of Zeno runs, or absence of Zeno-timelocks, guarantee timelock-freedom).

DTAs can be extended with structured data types [29, 148] and clock variables [93]. A more general class of LTL formulas could also be verified and assisted by MONA, such as waiting-for,

causality and response properties, using the inference rules proposed in [112, 93]. In general, deductive proofs will benefit from tools that generate auxiliary invariants [27, 143, 59].

Additionally, a class of finite-state DTA specifications may be identified that can be translated to timed automata. Some expressiveness of DTAs will be sacrificed to gain automatic verification, with the additional benefit that the resulting automata will be free from time-actionlocks. In this respect, Robson's work [139] on translating Timed I/O Automata to Uppaal is relevant.

Appendix A

Reachability Analysis in Uppaal (Chapter 2)

Here we explain the basics of reachability analysis as performed by model-checkers such as Uppaal and Kronos. We also give some details about Uppaal's optimisations.

A.1 Reachability Analysis

Model-checkers are able to verify a number of different correctness properties. Even though different algorithms are needed to verify different classes of properties, *reachability analysis* is central to all of them. Reachability analysis refers to the exploration of the state-space, in order to determine whether a particular state is reachable. The space of clock valuations is infinite, thus, exploration must be performed on a finite abstraction of this space.

This section describes the *reachability graph*, which divides the state space of a network into a finite number of symbolic states (necessarily, a symbolic state represents a possibly infinite set of states in the underlying TTS).[1] Given a formula ϕ, which characterises a set of states S_ϕ in the underlying TTS, an algorithm is presented that explores the reachability graph, and determines whether there exists a symbolic state in the graph that satisfies ϕ (i.e., a symbolic state that contains any $s \in S_\phi$). To simplify the presentation, explanations assume a single timed automaton, but they readily extend to networks.

[1] The reachability graph is based on the region graph of Alur and Dill [7], but it is a much coarser abstraction, i.e., the space of clock valuations is divided in fewer symbolic states [156, 21].

A.1.1 Symbolic States

Let $A = (L, l_0, TL, C, T, I)$ be a timed automaton, and $TS_A = (S, s_0, Lab, T_S)$ the TTS that represents the behaviour of A. A *symbolic state* is a pair (l, Z), where $l \in L$ is a location in A and $Z \in CC_C$ (called a *zone*) is a conjunction of clock constraints (bounds on single clocks, and clock differences). A symbolic state (l, Z) denotes the (usually infinite) set

$$(l, Z) = \{\, s \mid s \in S,\ s = [l, v] \text{ and } v \models Z \,\}$$

Symbolic states constitute the nodes of the reachability graph. We explain here a number of operations on zones, which are required during the construction of the graph. These operations are relevant for the calculation of node successors, as it will become clear shortly.

Conjunction. The conjunction of a zone Z with a clock constraint, $\phi \in CC_C$, corresponds to the intersection of the solution sets. This gives the expected,

$$Z \wedge \phi \triangleq \{\, v \mid v \models Z \text{ and } v \models \phi \,\}$$

Reset. The reset of a zone Z, with respect to a reset set r, denotes the set of valuations that result from valuations in Z, by setting every clock in r to 0.

$$r(Z) \triangleq \{\, r(v) \mid v \models Z \,\}$$

Forward projection. The forward projection of a zone Z is the set of valuations that can be reached from any valuation in Z, by letting some time pass. In terms of clock constraints, you can understand this operation as removing upper bounds in single-clock constraints. Lower bounds, and bounds on clock-difference constraints, are preserved.[2]

$$Z^{\uparrow} \triangleq \{\, v + d \mid v \models Z \wedge d \in \mathbb{R}^{+0} \,\}$$

[2] Zones obtained during reachability include clock-difference constraints that represents that all clocks advance uniformly. Therefore, this property is preserved by forward projection.

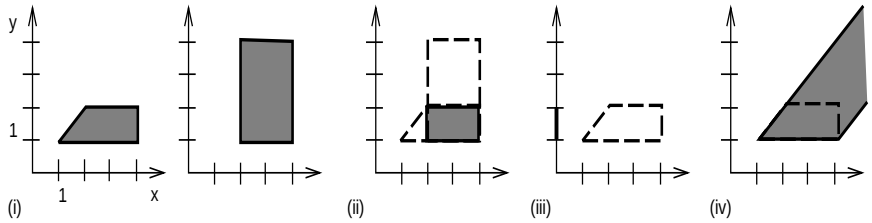


Figure A.1: (i) Z and ϕ, (ii) $Z \wedge \phi$, (iii) $r(Z),\ r = \{x\}$, (iv) $Z^{\uparrow}$

Figure A.1 illustrates the results of these operations, where

$$\begin{aligned}
Z \quad &: \ x \geq 1 \wedge x \leq 4 \wedge y \geq 1 \wedge y \leq 2 \wedge x - y \geq 0 \wedge x - y \leq 3 \\
\phi \quad &: \ x \geq 2 \wedge x \leq 4 \wedge y \geq 1 \wedge y \leq 4 \\
Z \wedge \phi \quad &: \ x \geq 2 \wedge x \leq 4 \wedge y \geq 1 \wedge y \leq 2 \\
r(Z) \quad &: \ x = 0 \wedge y \geq 1 \wedge y \leq 2 \\
Z^{\uparrow} \quad &: \ y \geq 1 \wedge x - y \geq 0 \wedge x - y \leq 3
\end{aligned}$$

The following operation, *normalisation*, ensures that the space of clock valuations can be divided into a finite number of zones.

Normalisation. Let $c_{\max} \in \mathbb{N}$ be the greatest constant occurring in any guard or invariant in A. Let Z be a zone, and $m \in \mathbb{N},\ m > c_{\max}$. We define $norm(Z)$ as the zone that can be obtained from Z by removing all constraints $x \sim m$ and $x - y \sim m$ ($\sim \in \{<, \leq\}$); and replacing all constraints $x \sim m$ and $x - y \sim m$ ($\sim \in \{>, =\}$) with $x > c_{\max}$ and $x - y > c_{max}$, respectively.

Other operations on zones include the check for *consistency*, which determine if the solution set of a zone is nonempty; and *inclusion* between zones, $Z \subseteq Z'$ (in terms of the solution sets).

A.1.2 Reachability Graph

The reachability graph is a finite abstraction of the behaviour (TTS) of a timed automaton. Nodes in the graph denote symbolic states. Moreover, a node (l, Z) represents the set of time transitions

$$\{\, [l, v] \stackrel{d}{\Longrightarrow} [l, v + d] \mid d \in \mathbb{R}^{+} \ \wedge \ v \models Z \ \wedge \ v + d \models Z \,\}$$

An edge in the graph, from (l, Z) to (l, Z'), denotes the set of action transitions

$$\{\, [l, v] \stackrel{a}{\Longrightarrow} [l', v'] \mid a \in Act \ \wedge \ v \models Z \ \wedge \ v' \models Z' \,\}$$

The starting node in the graph is the symbolic state (l_0, Z_0), where l_0 is the initial location of A and Z_0 is the zone describing the maximum time-progress allowed in l_0 (notice that this depends on the invariant in l_0). Given a node (l, Z) and a transition $t = l \xrightarrow{a,g,r} l'$ in the automaton, a direct successor (l', Z') is obtained, where Z' represents the maximum time-progress allowed in l', after t has been performed. Z' is derived from Z, the guard g and reset set r, and the target invariant $I(l')$.

Generating Successors. Let A be a timed automaton; $t = l \xrightarrow{a,g,r} l'$ a transition in A; and (l, Z) a symbolic state. Let Z_t be the zone defined as follows,

$$Z_t \triangleq r(Z \wedge g) \wedge I(l')$$

The symbolic state (l', Z_t) represents the set of all states that can be reached from any state in (l, Z), by performing transition t. Hence, the following holds,

$$\forall\, s' \in (l', Z_t)\,,\ \exists\, s \in (l, Z)\,.\, s \stackrel{a}{\Longrightarrow} s'$$

It is not difficult to see that, if a state $s' = [l', v'] \in (l', Z_t)$ is reachable from some state $s \in (l, Z)$, by performing t, then so is any state $s'' = [l', v' + d],\ d \in \mathbb{R}^+,\ (v' + d) \models I(l')$ (just by allowing time to pass in the target location l'). The set of direct successors of a node (l, Z) is given by,

$$\mathit{Successors}(l, Z) = \{\, (l', \mathit{norm}(Z_t^{\uparrow} \wedge I(l'))) \mid t = l \xrightarrow{a,g,r} l' \,\}$$

We assume that $\mathit{Successors}(l, Z)$ contains only consistent states, i.e. states (l', Z') where Z' represents a non-empty set of valuations. This assumption, in turn, implies the consistency of Z_t. Successors of (l, Z), then, are generated from outgoing transitions in l that are *enabled* by some valuation in Z.

Reachability Graph. The reachability graph is a finite set of nodes, N, and a finite set of edges, E. N is defined as the smallest set of symbolic states which includes the initial state (l_0, Z_0) and is closed under the successor relation (as defined by $\mathit{Successors}(l, Z)$). E is simply defined as the set of edges which connect every symbolic state with its successors.

$$N = \{\, (l_0, Z_0) \,\} \cup \bigcup_{(l,Z) \in N} \mathit{Successors}(l, Z)$$
$$E = \{\, (l, Z) \rightarrow (l', Z') \mid (l, Z) \in N \wedge (l', Z') \in \mathit{Successors}(l, Z) \,\}$$

where the initial zone is given by $Z_0 \triangleq I(l_0) \wedge \bigwedge_{x,y \in C,\, x \neq y} x = y$.

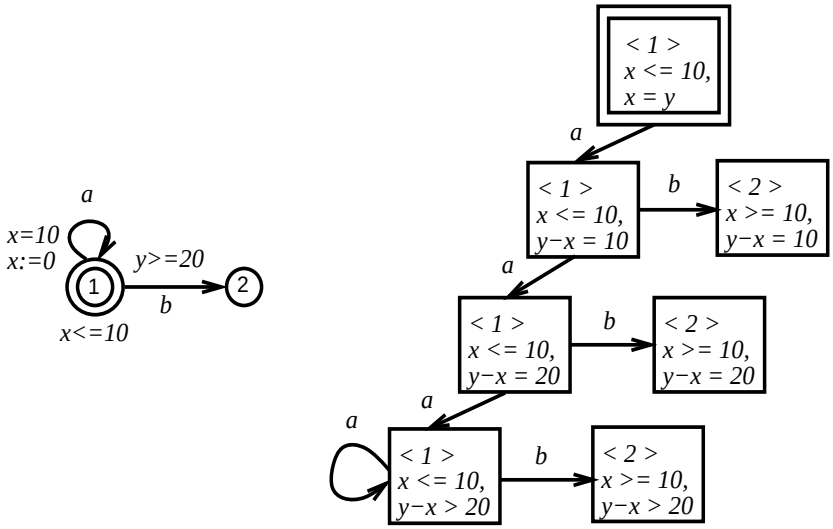


Figure A.2: Timed Automaton (left) and its Reachability Graph (right)

Example.[3] Figure A.2 shows a timed automaton and its reachability graph. The starting node in the reachability graph is the symbolic state

$$(1, x \leq 10 \wedge x = y)$$

This state denotes that initially (location 1) time can pass up to 10 time units and that clocks x and y have the same value. One of the direct successors of the starting node is given by

$$(1, x \leq 10 \wedge y - x = 10)$$

This new symbolic state represents all the states that are reachable from $(1, x \leq 10 \wedge x = y)$ by performing transition a, and then letting some time pass in location 1. Similarly, the symbolic state,

$$(2, x \geq 10 \wedge y - x = 10)$$

represents the states reachable from any state in $(1, Z_1)$, after transition b is performed. Note that y is allowed to "drift" unbounded in location 1. Clock x is constrained by the invariant in location 1, such that $v(x) \leq 10$ whenever the automaton remains in that location. On the other hand, the difference between y and x increases every time a is performed (x is reset, but y is not). If normalisation would not be applied when the successors are generated, then an infinite

[3]This is inspired by an example in [21].

number of symbolic states $(1, Z_i)$ could be derived, where

$$\begin{aligned} Z_0 = &\quad x \leq 10 \wedge x = y \\ Z_1 = &\quad x \leq 10 \wedge y - x = 10 \\ Z_2 = &\quad x \leq 10 \wedge y - x = 20 \\ Z_3 = &\quad x \leq 10 \wedge y - x = 30 \\ &\vdots \\ Z_n = &\quad x \leq 10 \wedge y - x = 2n \end{aligned}$$

and $(1, Z_{i+1})$ is a successor of $(1, Z_i)$. Consider the effect that the different zones $Z_3, Z_4, \ldots$ have on reachability. Every zone contains valuations that enable a, because $v(x) \leq 10$, and valuations that enable b, because $v(y) > 20$. Thus, these zones introduce unnecessary fragmentation into the space of clock valuations. With normalisation, the infinite union, $Z_3 \cup Z_4 \cup \ldots$, is represented by a single zone,

$$norm(Z_3) \;=\; x \leq 10 \wedge y - x > 20$$

and the self-loop in $(1, x \leq 10 \wedge y - x > 20)$, which can be seen in the reachability graph of Figure A.2 (right).[4]

A.1.3 A Reachability Algorithm

A reachability algorithm is shown in Figure A.3. This is sketched as a Boolean function *reachable*(A, ϕ), where A is a timed automaton, and ϕ is a state formula with the following syntax,

$$\phi \;::=\; cc \mid l \mid \phi \wedge \phi \mid \neg\, \phi$$

where cc is a clock constraint and l a location in A.

The algorithm can be thought of as exploring the reachability graph described in the previous section, generating its nodes on demand. The exploration begins with the initial node of the graph, and then generates, in every step, the set of successors for the current node. This is repeated until a node is found which satisfies the reachability formula (line 7), or until the whole graph has been explored and no node has been found to satisfy ϕ (line 19).

A state (l, Z) satisfies ϕ if the intersection between the solution sets is nonempty (i.e., if any of the states characterised by ϕ is reachable). Termination of the algorithm is guaranteed by normalisation, which ensures that only a finite number of different zones can be generated, and by keeping a set of visited nodes, *Visited*. The set *Waiting* contains the nodes to be explored. Note that, given two symbolic states, (l, Z) and (l, Z_v), if $Z \subseteq Z_v$ then all states which are

[4] Normalisation replaces, in Z_3, the constraint $y - x = 30$ with $y - x > 20$ ($c_{\max} = 20$).

reachable from (l, Z) are also reachable from (l, Z_v). Thus, there is no need to explore (l, Z) if (l, Z_v) has been visited already (lines 9 and 10).

```
    Function reachable(A, φ)
1   begin
2       Visited ← ∅;
3       Waiting ← {(l_0, Z_0)};
4       while Waiting ≠ ∅ do
5         Choose (l, Z) ∈ Waiting;
6         Waiting ← Waiting \{(l, Z)};
7         if (l, Z) ⊨ φ then return true;
8         end if
9         if ∄ (l, Z_v) ∈ Visited. Z ⊆ Z_v then
10          Waiting ← Waiting ∪ Successors(l, Z);
11          Visited ← Visited ∪ {(l, Z)};
12        end if
13      end while
14      return false;
15  end.
```

Figure A.3: A Reachability Algorithm

This algorithm readily applies to a network of timed automata, if we interpret symbolic states over location vectors, and the generation of successors over transitions in the product automaton. Note that the product automaton does not need to be constructed before the exploration is performed. Rather, as exploration progresses, only the necessary location vectors are generated on demand. Similarly, transitions are generated only from the reachable location vectors, either from completed actions in some component, or from synchronising half actions (i.e., synchronisation is computed on demand). In the generation of successors, only those transitions that are enabled at the current symbolic state need to be considered.

It is worth mentioning, as well, that the algorithm of Figure A.3 can be modified to avoid keeping visited states in the *Waiting* list. Note that successors are generated and stored in *Waiting* regardless of whether they have been previously visited, thus potentially wasting memory. Additionally, the algorithm can be modified to provide diagnostic information (e.g., the symbolic path leading to a relevant reachable state).

Example: Reachability Analysis on the Multimedia Stream. Figure A.4 shows (part of) the reachability graph for the network of timed automata of Figure 2.5. For example, *S0* corresponds to the initial symbolic state, in which the location vector $\langle 0, 1, 1, 1\rangle$ corresponds to *Source.State0*, *Place1.State1*, *Place2.State1* and *Sink.State1*; and the initial zone represents that all clocks are initially set to 0. Moreover, because of the constraint $t1 = 0$, time is not allowed

to pass (this constraint results from the invariant in *Source.State0*).

The edge connecting *S0* with one of its successors, *S1*, corresponds to the initial, urgent *sourceOut* between the *Source* and *Place1*. The zone in *S1* confirms the intuition that, after performing the first *sourceOut* action, control can remain in *Source.State1* and *Place1.State2* as long as $v(t1) \leq 50$ and $v(t4) \leq 90$. Following a similar reasoning, *S4* represents all those states that can be reached after the first packet has been played and before the second packet arrives at the *Sink*, provided the former was initially put in *Place1*.

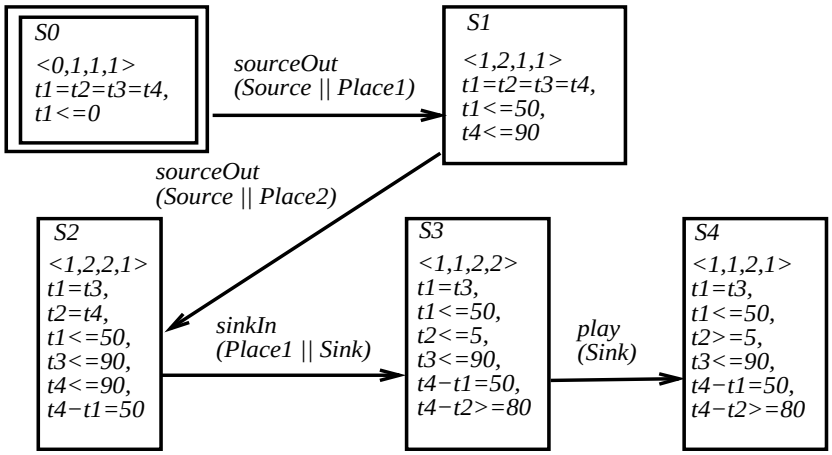


Figure A.4: Reachability Graph (Fragment) for the Multimedia Stream

A.2 Reachability Analysis in Uppaal

Except for the leads-to formula, Uppaal's main verification algorithm is basically an efficient implementation of the algorithm of Figure A.3. Reachability is performed on-the-fly, and networks are verified without building the product automaton (Section A.1.3). We discuss, next, some implementation details and optimisations in Uppaal's reachability algorithm.

A.2.1 The Representation of Symbolic States

Symbolic states are of the form (D, Z), with a *discrete state* D and a zone Z. The discrete state represents both, a location vector and a valuation for all declared variables. Correspondingly, the interpretation of guards, invariants and assignments[5] are extended to consider variables. For example, in order to determine that a transition t, with guard $g \triangleq g_c \wedge g_v$, is enabled in some state (D, Z), the following conditions must be checked.

[5]When synchronisation occurs, assignments in the transition with the output action are performed first. This asymmetry solves the problem of a variable being assigned different values in the output and input actions.

- Clock constraints, g_c, must be satisfiable in Z.
- Variable constraints, g_v, must be satisfiable in D.
- There exists at least one valuation in (D, Z) (both for variables and clocks) that satisfies g, and such that, after the assignments have been performed, the resulting valuation satisfies the invariant in the target location (both for variables and clocks).

Uppaal offers two different representations for zones (the user may select the most convenient for the specification at hand). Difference Bound Matrices (DBMs) result in fast zone operations, but memory may be wasted in redundant constraints (DBMs must store a bound for the difference between *each* pair of clocks). On the other hand, Minimal Constrain Graphs [103] eliminate this redundancy, but may result in slower operations.

Clock Difference Diagrams (CDDs), a BDD-like data structure to efficiently represent list of zones, are described in [17]. Efficient encoding of DBMs, sharing among data structures and other storage optimisations are discussed in [18, 15, 60].

Uppaal also allows the user to control other parameters regarding the state-space representation, such as the size of hash tables, or the use of different zone-inclusion algorithms. These options are described in more detail in [22].

Difference Bound Matrices. Let Z be a zone over the set of clocks $C_Z = \{ x_1, x_2, \ldots, x_n \}$. Let x_0 be a special clock denoting the constant 0. Z can be expressed using just clock-difference constraints $x_i - x_j \sim k$, where $x_i, x_j \in C_Z \cup \{x_0\}$, $\preccurlyeq \in \{<, \leq\}$ and $k \in \mathbb{Z}$. Given Z expressed in this form, a DBM representing Z is a matrix M of dimension $(n+1) \times (n+1)$, where the element M_{ij} stores the bound for the difference $x_i - x_j$. The element M_{ij} is calculated as follows.

- M_{ij} is set to $(k, \preccurlyeq)$, for every constraint $x_i - x_j \preccurlyeq k$ occurring in Z;
- M_{ij} is set to ∞, for every clock difference $x_i - x_j$, $i \neq 0,\ i \neq j$, which is not constrained in Z;
- M_{ii} is set to $(0, \leq)$, denoting that the difference between a clock and itself is 0; and
- M_{0i} is set to $(0, \leq)$, for every clock difference $x_0 - x_i$ which is not constrained in Z. This denotes that every clock is positive.

For example, the zone

$$Z = \quad x_1 \geq 5 \wedge x_1 \leq 10 \wedge x_3 > 20 \wedge x_1 - x_2 = 3$$

can be written using clock-difference constraints as follows.

$$
\begin{aligned}
Z = \quad & x_0 - x_1 \leq -5 \quad \wedge \\
& x_1 - x_0 \leq 10 \quad \wedge \\
& x_0 - x_3 < -20 \quad \wedge \\
& x_1 - x_2 \leq 3 \quad \wedge \\
& x_2 - x_1 \leq -3
\end{aligned}
$$

The DBM implementing Z is then,

$$
\begin{pmatrix}
(0,\leq) & (-5,\leq) & (0,\leq) & (-20,<) \\
(10,\leq) & (0,\leq) & (3,\leq) & \infty \\
\infty & (-3,\leq) & (0,\leq) & \infty \\
\infty & \infty & \infty & (0,\leq)
\end{pmatrix}
$$

Zones can be interpreted as directed graphs. Let M be a DBM with clocks in $\{x_0, \ldots, x_n\}$, corresponding to a certain zone Z. Every clock is represented by a node in the graph. For every pair of nodes x_i and x_j, there is an edge in the graph from x_j to x_i, labelled with M_{ij}. Figure A.5 shows the graph representation for the zone Z of our previous example.

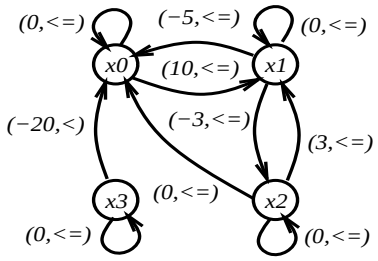


Figure A.5: Graph Representation for a DBM

Uppaal implements operations on zones, such as forward projection and inclusion, in terms of the DBM representation. For efficiency reasons, zones are kept in *canonical form*. A zone is in canonical form if no atomic constraint can be strengthened without losing solutions. The canonical form corresponds to the shortest-path closure of the graph representation for the zone (Uppaal uses Floyd-Warshall's all-pairs shortest-path algorithm [21]).

Bengtsson and Yi [21] discuss how operations on zones can be implemented so that either the canonical form is preserved, or a full shortest-path calculation is not necessary to put the zone back into canonical form. Here we sketch just a few intuitive implementations. We assume that the zones Z and Z' are implemented by DBMs M and M', both in canonical form, and they refer to the same set of clocks, $\{x_0, \ldots, x_n\}$. We also assume $0 \leq i, j \leq n$, and $\preccurlyeq \in \{<, \leq\}$. The minimum between two bounds, $min(b_1, b_2)$, is defined as follows.

$$min(b_1, b_2) = \begin{cases} b1 \quad \text{if} & b_2 = \infty \,; \text{or} \\ & b_1 = (n, \preccurlyeq_1),\ b_2 = (m, \preccurlyeq_2) \text{ and } n < m \,; \text{ or} \\ & b_1 = (n, <) \text{ and } b_2(n, \leq) \\ \\ b2 \quad \text{otherwise.} \end{cases}$$

Operations on zones correspond to the following DBM operations.

- The conjunction of Z with a clock constraint $x_i - x_j \preccurlyeq b$, updates $M_{ij} \leftarrow min((b, \preccurlyeq), M_{ij})$.
- The reset $x_i := 0$ in Z updates $M_{i0} \leftarrow (0, \leq)$ and $M_{0i} \leftarrow (0, \leq)$; and, for all $j \neq i$, it updates $M_{ji} \leftarrow M_{j0}$ and $M_{ij} \leftarrow M_{0j}$.
- Inclusion between zones, $Z \subseteq Z'$, holds if $M_{ij} = min(M_{ij}, M'_{ij})$ for all i, j (this is correct because both zones are in canonical form).
- Forward projection, $Z^{\uparrow}$, updates $M_{i0} \leftarrow \infty$, for all i. This effectively removes the upper bounds on single-clock constraints.

Minimal Constraint Graphs. Given a zone as a directed graph, an algorithm [103] removes redundant edges, and returns an equivalent, minimal graph in canonical form (i.e., with the same shortest-path closure as the original graph). Even though the minimisation algorithm causes some overhead during reachability, experiments show that minimal constraint graphs may actually speedup verification, when compared with DBMs. This can be explained in terms of faster zone-inclusion checks, which is performed by the reachability algorithm every time a new state is generated. Inclusion between zones can be decided in time linear in the number of number of constraints, thus, inclusion is influenced adversely by the redundancy in DBMs.

A.2.2 Optimisations

Keeping Symbolic States in a Single List. In Uppaal, generated states are only saved in the *Visited* list; the *Waiting* list just keep pointers to elements in *Visited*. States are considered visited as soon as they are generated: They are checked for inclusion with respect to states already in *Visited*, and added to this list accordingly. In this way, there cannot be visited states waiting to be explored (and wasting memory in the data structure) [15].

Storing Fewer Visited States. Uppaal implements a technique that reduces the number of symbolic states stored in the *Visited* list [103]. The reachability algorithm needs to save visited

states in order to guarantee termination. However, not every generated state needs to be saved. Based on static information gathered from the network structure, a minimal subset of symbolic states can be identified during reachability: keeping just this subset in *Visited* is enough to guarantee termination. These states correspond to locations in the cycles that can be reached from some outside the cycle. Furthermore, this subset is chosen in such way that not only termination is guaranteed, but also that no symbolic state is explored more than once.

Normalisation. For efficiency reasons, Uppaal reduces zones with respect to a set of maximal constants $c_{\max}(x)$, one for every clock x, instead of one maximal constant for all clocks (as defined in Section A.1.1) [21]. More efficient techniques have recently been proposed in [16].

Over-approximation. During reachability analysis, many symbolic states may be found that refer to the same discrete state (location vector plus data variables), but differ in the zones. This is a natural reflection of executions visiting the same location at different times (for example, when a given location can be reached through different paths in the automaton). When this happens, and there is no inclusion between zones, Uppaal associates a list of zones to the same discrete state. For some systems, the symbolic state-space is too big, and there is no memory available to store all the generated zones.

To help with this problem, Uppaal can be instructed to perform an over-approximation of the state-space (see, e.g., [9, 61]). With over-approximation, every discrete state is associated with a single zone, usually called a *convex hull*. The convex hull is a zone that "tightly" approximates the union of zones that may otherwise be related to discrete states. Storing convex hulls, instead of lists of zones, clearly requires less memory. However, verification on convex hulls may yield inconclusive answers. A convex hull represents, in general, a superset of those valuations included in the original lists of zones. In other words, a convex hull represents a superset of reachable states. Therefore, when verification is performed on convex hulls, only the *unreachability* of states can be guaranteed.

Under-approximation. As is the case for over-approximation, under-approximation reduces the amount of memory required to store visited states, and sacrifices exact verification. When under-approximation is used, the list of visited states just records whether a state has been visited, instead of storing the state itself. So, the list of visited states is implemented as a bit vector, and a hash signature is computed for every generated state: If *Visited* is the list of visited states, and $h(s)$ is the signature for some symbolic state $s = (l, Z)$, then $\mathit{Visited}[h(s)] = 1$ if s is visited, and $\mathit{Visited}[h(s)] = 0$ otherwise.

It may happen, though, that different states with the same signature are found by the reachability algorithm. After a state s is recorded as visited, all other states with the same signature, $s', s'', \ldots$, will be considered visited, and will not be explored. As a consequence, some parts of the state-space will remain unexplored when the reachability analysis is finished (i.e., those states which can be reached only from $s', s'', \ldots$). When verification is performed with under-approximation, only the *reachability* of states can be guaranteed.

Another problem with under-approximation is that inclusion between zones cannot be checked, which causes some redundant states to be explored. Bengtsson compares different implementations of under-approximation in [18].

Active Clock Reduction. A clock is said to be *active* from the point where it is reset, up to the point where it is tested (i.e., in a guard or invariant), without being reset on the way. For every location in the timed automaton, a set of active clocks can be computed based on syntactic information. The technique, proposed by Daws and Yovine [62], is compositional and can be applied to networks of automata.

During reachability analysis, then, generated zones need to include only constraints on active clocks (for the current location). This may result not only in smaller zones, but also in fewer symbolic states: Different symbolic states, sharing a common discrete part, may become indistinguishable when clock constraints are removed from their zones.

Active clock reduction is performed automatically in Uppaal. Similar heuristics, applied to data variables, are suggested in [22].

Other Optimisations. Hendriks and Larsen [80] propose *cycle acceleration* to deal with the problem of different time scales in a system. When a timed automata specification must accommodate time constraints in different time scales (e.g., where some deadlines are specified in milliseconds, whereas others are given in minutes), all constants must be converted to the finer scale. The constants obtained may differ wildly in magnitude, which causes unnecessary fragmentation in the space of clock valuations (i.e., many different zones will be generated that do not contribute to the verification of the property). In certain kind of specifications (e.g., those with busy-waiting behaviour), cycles responsible for this fragmentation can be identified, and syntactically transformed to avoid it (or reduce it).

Hendriks et al. [79] present a prototype implementation of Uppaal with added *symmetry reduction*. The user identifies structural symmetry in the specification, and this information is used by the model-checker to group symbolic states into equivalence classes: Given a state formula ϕ, all states in a class satisfy ϕ, or none does. For every successor state generated during reachability, a representative of that state's class is computed and stored instead. In this way,

only representative states need to be stored and explored, with the obvious improvements in time and memory consumption (experiments described in [79] confirm that, for systems that present a high degree of symmetry, memory consumption decreases drastically).

Appendix B

Simpler Reachability Formulae (Chapter 5)

Sections 5.2 and 5.3 defined the formulae $\alpha(lp)$ and $\beta(lp)$, so that they clearly reflect our intuition behind Zeno loops, maximal valuations and escape transitions. These definitions also facilitated the proofs. However, they contain some redundancy that can be easily eliminated. We present a few of such reductions here.

A Simpler $\alpha(lp)$.[1] Note that, v is a valuation that satisfies all invariants in a loop lp, if and only if $v(x) \leq c_{\min}(x, lp)$ for all clocks occurring in any invariant of lp (by definition of smallest upper bound). In addition, if v is a maximal valuation of lp, then $v(y) = 0$ for all $y \in \mathit{Resets}(lp)$. Then, and because we assumed that invariants are either *true* or right-closed, the value of clocks that are reset in lp trivially satisfy every invariant of lp. We can therefore simplify the original expression of $\alpha(lp)$ and obtain the equivalent $\alpha'(lp)$ instead, where,

$$\begin{aligned} \alpha'(lp): \quad & \bigwedge\nolimits_{z \in \mathit{Clocks}(lp) \setminus \mathit{Resets}(lp)} z \leq c_{\min}(z, lp) \\ \wedge \quad & \bigwedge\nolimits_{g \in \mathit{Guards}(lp)} g \\ \wedge \quad & \bigwedge\nolimits_{y \in \mathit{Resets}(lp)} y = 0 \\ \wedge \quad & \mathit{sub}(lp) \end{aligned}$$

[1] The same reduction can be applied to the formula $\gamma(lp)$ of Section 5.3

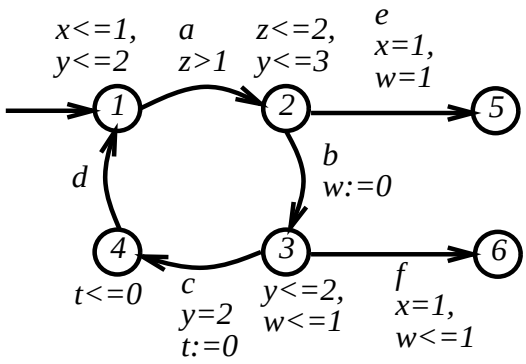


Figure B.6: Example Loop of Figure 5.3(i)

For example, $\alpha'(lp)$ is calculated below for $lp = \langle a, b, c, d\rangle$ in Figure 5.3(i) (we recall this loop in figure B.6).

$$\begin{aligned}
\alpha'(lp): \quad & (x \leq 1 \wedge y \leq 2 \wedge z \leq 2) \\
\wedge \quad & (z > 1 \wedge y = 2) \\
\wedge \quad & (t = 0 \wedge w = 0) \\
\wedge \quad & (\ (x = 1 \wedge z = 2 \wedge y = 2 \wedge t = 0) \vee (x = 1 \wedge z = 2 \wedge w = 1 \wedge t = 0) \vee \\
& (y = 2 \wedge z = 2 \wedge y = 2 \wedge t = 0) \vee (y = 2 \wedge z = 2 \wedge w = 1 \wedge t = 0)\ )
\end{aligned}$$

A Simpler $\beta(lp)$. Consider any transition $l \xrightarrow{a,g,r} l' \in Esc(lp)$ where the guard g is of the form $\bigwedge_{k=1}^{p} x_k \sim c_k$, where $x_k \in C$, $c_k \in \mathbb{N}$ and $\sim \in \{<, >, =, \leq, \geq\}$. Consider, in addition, a maximal valuation v of lp and any conjunct in g, say $x_k \sim c_k$ for some $1 \leq k \leq n$, where x_k is reset in lp. If this conjunct is of the form $x_k \sim' c_k$, where $\sim' \in \{>, =, \geq\}$ and $c_k > 0$, then clearly the transition is not enabled by v (by definition of maximal valuation, $v(x_k) = 0$) and so it does not need to be considered in $\beta(lp)$. If, on the other hand, the conjunct is of the form $x_k \sim'' c_k$, where $\sim'' \in \{<, \leq\}$ and $c_k \in \mathbb{N}$, it is easy to see that it is trivially satisfied by v and so it can be replaced, in g, by $true$. For example, for $lp = \langle a, b, c, d\rangle$ in Figure 5.3(i),

$$\beta(lp) = \ \neg\ (x = 1 \wedge w = 1) \wedge \neg\ (x = 1 \wedge w \leq 1)$$

However, transition e is not enabled by any maximal valuation of lp, because w is reset in the loop. The conjunct $w \leq 1$ in transition f is trivially satisfied by any maximal valuation of lp, for the same reason. Therefore, we can simplify $\beta(lp)$ and obtain the equivalent $\beta'(lp)$, where

$$\beta'(lp) = \ \neg\ (x = 1 \wedge false) \wedge \neg\ (x = 1 \wedge true) \equiv \neg\ x = 1$$

Appendix C

Proofs (Chapter 7)

THEOREM **C.1.** *For any interval σ and any PITL formula P the following holds:*

(Moszkowski's definition for $\sigma \models P^*$)
$|\sigma| = 0$ or exists a sequence $k_0, k_1, \ldots, k_m \in \mathbb{N}$ s.t.
$k_0 \leq k_1 \leq \ldots \leq k_m$, $k_0 = 0$, $k_m = |\sigma|$ and
for all $0 \leq i < m$, ${}^{k_i}(\sigma^{k_{i+1}}) \models P$

iff

(our definition for $\sigma \models P^*$)
$|\sigma| = 0$ or exists a sequence $r_0, r_1, \ldots, r_m \in \mathbb{N}$ s.t.
$r_0 < r_1 < \ldots < r_m$, $r_0 = 0$, $r_m = |\sigma|$ and
for all $0 \leq i < m$, ${}^{r_i}(\sigma^{r_{i+1}}) \models P$

Proof. (by induction on $|\sigma|$)
Case $|\sigma| = 0$:
In both definitions, P^* is trivially satisfiable by empty intervals.

Case $|\sigma| = n + 1,\ n \in \mathbb{N}$:
We will only prove the "sufficient" condition of the equivalence. The "necessary" condition trivially holds, because a $<$-sequence is also a $\leq$-sequence. If σ is a non-empty interval and indeed there exists such a $\leq$-sequence, then there must be an iteration of P holding over a non-empty suffix of the model. In other words, then, there must exist $j,\ 1 \leq j \leq m$ such that $k_{j-1} < k_j = \ldots = k_m = |\sigma|$ and ${}^{k_{j-1}}\sigma \models P$. Otherwise there is no way to link the first and last state of σ with a joint sequence of models for P. Furthermore,

exists a sequence $k_0, k_1, \ldots, k_j \in \mathbb{N}$ s.t.
$k_0 \leq k_1 \leq \ldots \leq k_{j-1} < k_j$, $k_0 = 0$, $k_j = |\sigma|$ and
for all $0 \leq i < j$, ${}^{k_i}(\sigma^{k_{i+1}}) \models P$

As $k_{j-1} \leq n$ we can use the induction hypothesis on the $\leq$-prefix of the previous sequence (i.e. the points $k_0, \ldots, k_{j-1}$), to obtain the required expression:

exists a sequence $r_0, r_1, \ldots, r_j \in \mathbb{N}$ s.t.
$r_0 < r_1 < \ldots < r_{j-1} < r_j$, $r_0 = 0$, $r_j = |\sigma|$ and
for all $0 \leq i < j$, ${}^{r_i}(\sigma^{r_{i+1}}) \models P$

This concludes the proof of Theorem C.1. □

LEMMA **7.1.** For any $\sigma \in \mathcal{I}$, σ' a subinterval of σ (not necessarily contiguous), $0 \leq j' \leq |\sigma'|$, and PITL formula P, the following holds,

$$(\sigma', j') \models P \text{ iff } (i, j, n, S, \mathcal{M}) \models_W P$$

where $S \subseteq \{0, 1, \ldots, |\sigma|\}$, $i, j, n \in S\,(i \leq j \leq n)$, $\mathit{subinterval}(i, j, n, S, \sigma) = (\sigma', j')$ and $\mathcal{M} = [\![\,]\!]_\sigma$.

Proof. We recall, from Section 7.3.2, the definition of the function $\mathit{subinterval}()$. For $\sigma \in \mathcal{I}$, $S = \{s_0, s_1, \ldots, s_m\}$, $S \subseteq \{0, 1, \ldots, |\sigma|\}$, $i, j, n \in S\,(i \leq j \leq n)$,

$$\mathit{subinterval}(i, j, n, S, \sigma) = (\sigma', j')$$

where

$$\sigma' = \sigma_{s_k}\sigma_{s_{k+1}} \ldots \sigma_{s_{k+j'}} \ldots \sigma_{s_{k+|\sigma'|}}$$

for some $k \in \mathbb{N}$ $(0 \leq k + |\sigma'| \leq m)$, s.t. $i = s_k$, $j = s_{k+j'}$ and $n = s_{k+|\sigma'|}$. Intuitively, σ' is a subinterval constructed from those states in σ represented by indices in S between i and n, and where j points to the current state (i.e., $\sigma'_0 = \sigma_i$, $\sigma'_{j'} = \sigma_j$ and $\sigma'_{|\sigma'|} = \sigma_n$). The proof of Lemma 7.1 is by induction on the number of symbols in the formula P, $|P|$ (we use i.h. to refer to the inductive hypothesis).

Case $|P| = 1$:

$P \triangleq v$:

$(i, j, n, S, \mathcal{M}) \models_W v$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$j \in [\![h(v)]\!]_\sigma$	iff	[def. of $[\![\ ]\!]_\sigma$]
$v \in \sigma_j$	iff	[$\sigma_j = \sigma'_{j'}$, by def. of *subinterval*()]
$v \in \sigma'_{j'}$	iff	[def. of $\models$]
$(\sigma', j') \models v$		

$P \triangleq$ *empty*:

$(i, j, n, S, \mathcal{M}) \models_W$ *empty*	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$j = n$	iff	[$\sigma_j = \sigma'_{j'}$ and $\sigma_n = \sigma'_{\lvert\sigma'\rvert}$, by def. of *subinterval*()]
$j' = \lvert\sigma'\rvert$	iff	[def. of $\models$]
$(\sigma', j') \models$ *empty*		

$P \triangleq$ *false*: the proof trivially follows by definition.

Case $|P| > 1$:

Negation ($P \triangleq \neg P_1$):

$(i, j, n, S, \mathcal{M}) \models_W \neg P_1$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$(i, j, n, S, \mathcal{M}) \not\models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$]
$(\sigma', j') \not\models P_1$	iff	[def. of $\models$]
$(\sigma', j') \models \neg P_1$		

Disjunction ($P \triangleq P_1 \vee P_2$):

$(i, j, n, S, \mathcal{M}) \models_W P_1 \vee P_2$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$(i, j, n, S, \mathcal{M}) \models_W P_1 \vee$		
$(i, j, n, S, \mathcal{M}) \models_W P_2$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\lvert P_2\rvert < \lvert P\rvert$]
$(\sigma', j') \models P_1 \vee$		
$(\sigma', j') \models P_2$	iff	[def. of $\models$]
$(\sigma', j') \models P_1 \vee P_2$		

Quantification ($P \triangleq \exists\, v.\, P_1$):

$(i, j, n, S, \mathcal{M}) \models_W \exists\, v.\, P_1$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
exists $\mathcal{M}'$ s.t. $\mathcal{M}' \approx_{h(v)} \mathcal{M}$ and		
$(i, j, n, S, \mathcal{M}') \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\lvert P_2\rvert < \lvert P\rvert$]
exists σ'' s.t. $\mathcal{M}' = [\![\ ]\!]_{\sigma''}$, $\sigma'' \approx_v \sigma'$		
and $(\sigma'', j') \models P_1$	iff	[def. of $\models$]
$(\sigma', j') \models \exists\, v.\, P_1$		

Next ($P \triangleq \bigcirc P_1$):

$(i, j, n, S, \mathcal{M}) \models_W \bigcirc P_1$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$j < n$ and exists $k \in \mathbb{N}$ s.t.		
$cons(S, j, k)$ and $(i, k, n, S, [\![\ ]\!]_\sigma) \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\sigma'_{j'} = \sigma_j$ and $\sigma'_{j'+1} = \sigma_k$, by def. of *subinterval*()]
$j' < \lvert\sigma'\rvert$ and $(\sigma', j' + 1) \models_W P_1$	iff	[def. of $\models$]
$(\sigma', j') \models \bigcirc P_1$		

Chop ($P \triangleq P_1\ ;\ P_2$):

$(i, j, n, S, \mathcal{M}) \models_W P_1\ ;\ P_2$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
exists $k \in S,\ j \le k \le n$ s.t.		
$(i, j, k, S, \mathcal{M}) \models_W P_1$ and		
$(i, k, n, S, \mathcal{M}) \models_W P_2$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\lvert P_2\rvert < \lvert P\rvert$, $\sigma'_{j'} = \sigma_j$, $\sigma'_{\lvert\sigma'\rvert} = \sigma_n$ and $\sigma'_{k'} = \sigma_k$, by def. of *subinterval*()]
exists $k',\ 0 \le k' \le \lvert\sigma'\rvert$ s.t.		
$(\sigma'^{k'}, j') \models P_1$ and $(\sigma', k') \models P_2$	iff	[def. of $\models$]
$(\sigma', j') \models P_1\ ;\ P_2$		

Chop-star ($P \triangleq P_1^*$):

$(i,j,n,S,\mathcal{M}) \models_W P_1^*$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$j = n$ or exists $k_0, k_1, \ldots, k_m \in S$ s.t.		
$k_0 < k_1 < \ldots < k_m$, $k_0 = j$, $k_m = n$, and		
for all $r,\ 0 \le r < m$, $(i, k_r, k_{r+1}, S, \mathcal{M}) \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$]
		$\sigma'_{j'} = \sigma_j$, $\sigma'_{\lvert\sigma'\rvert} = \sigma_n$ and $\sigma'_{k'_r} = \sigma_{k_r}$,
		by def. of *subinterval*()]
$j' = \lvert\sigma'\rvert$ or exists $k'_0, k'_1, \ldots, k'_m \in \mathbb{N}$ s.t.		
$k'_0 < k'_1 < \ldots < k'_m$, $k'_0 = j'$, $k'_m = \lvert\sigma'\rvert$, and		
for all $r,\ 0 \le r < m$, $(\sigma'^{k'_{r+1}}, k'_r) \models_W P_1$	iff	[def. of $\models$]
$(\sigma', j') \models P_1^*$		

Until ($P \triangleq P_1$ *until* P_2):

$(i,j,n,S,\mathcal{M}) \models_W P_1$ *until* P_2	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
exists $k \in S,\ j \le k \le n$ s.t. $(i,k,n,S,\mathcal{M}) \models_W P_2$ and		
for all $r \in S,\ j \le r < k$, $(i,r,n,S,\mathcal{M}) \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\lvert P_2\rvert < \lvert P\rvert$]
		$\sigma'_{j'} = \sigma_j$, $\sigma'_{\lvert\sigma'\rvert} = \sigma_n$,
		$\sigma'_{k'} = \sigma_k$ and $\sigma'_{r'} = \sigma_r$,
		by def. of *subinterval*()]
exists $k',\ j' \le k' \le \lvert\sigma'\rvert$ s.t. $(\sigma', k') \models P_2$ and		
for all $r',\ j' \le r' < k'$, $(\sigma', r') \models_W P_1$	iff	[def. of $\models$]
$(\sigma', j') \models P_1$ *until* P_2		

Projection ($P \triangleq P_1 \; proj \; P_2$):

$(i,j,n,S,\mathcal{M}) \models_W P_1 \; proj \; P_2$ iff [def. of $\models_W$, $\mathcal{M} = [\![\;]\!]_\sigma$]
$j = n$ or exists $S' \subseteq S$ s.t.
$S' = \{i, \ldots, k_0, k_1, \ldots, k_m\}$ and
for all $t, \; i \leq t \leq k_0, \; t \in S \Leftrightarrow t \in S'$ and
$k_0 < k_1 < \ldots < k_m$, $k_0 = j$, $k_m = n$, and
for all $r, \; 0 \leq r < m$,
$(i, k_r, k_{r+1}, S, \mathcal{M}) \models_W P_1$ and
$(i,j,n,S',\mathcal{M}) \models_W P_2$ iff [i.h. on $|P_1| < |P|$, $|P_2| < |P|$]
$\sigma'_{j'} = \sigma_j$, $\sigma'_{|\sigma'|} = \sigma_n$ and $\sigma'_{k'_r} = \sigma_{k_r}$,
by def. of *subinterval*()]
$j' = |\sigma'|$ or exists $k'_0, k'_1, \ldots, k'_m \in \mathbb{N}$ s.t.
$k'_0 < k'_1 < \ldots < k'_m$, $k'_0 = j'$, $k'_m = |\sigma'|$, and
for all $r, \; 0 \leq r < m$,
$(\sigma'^{k'_{r+1}}, k'_r) \models P_1$ and
$(\sigma'_0 \ldots \sigma'_{k'_0} \sigma'_{k'_1} \ldots \sigma'_{k'_m}, k'_0) \models P_2$ iff [def. of $\models$]
$(\sigma', j') \models P_1 \; proj \; P_2$

Chop in the past ($P \triangleq P_1 \overset{\sim}{;} P_2$):

$(i,j,n,S,\mathcal{M}) \models_W P_1 \overset{\sim}{;} P_2$ iff [def. of $\models_W$, $\mathcal{M} = [\![\;]\!]_\sigma$]
exists $k \in S, \; i \leq k \leq j$ s.t.
$(i,k,n,S,\mathcal{M}) \models_W P_1$ and
$(k,j,n,S,\mathcal{M}) \models_W P_2$ iff [i.h. on $|P_1| < |P|$, $|P_2| < |P|$]
$\sigma'_0 = \sigma_i$, $\sigma'_{j'} = \sigma_j$,
$\sigma'_{|\sigma'|} = \sigma_n$ and $\sigma'_{k'} = \sigma_k$,
by def. of *subinterval*()]
exists $k', \; 0 \leq k' \leq j'$ s.t.
$(\sigma', k') \models P_1$ and
$({}^{k'}\sigma', j' - k') \models P_2$ iff [def. of $\models$]
$(\sigma', j') \models P_1 \overset{\sim}{;} P_2$

Since ($P \triangleq P_1$ *since* P_2):

$(i,j,n,S,\mathcal{M}) \models_W P_1$ *since* P_2	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
exists $k \in S,\ i \le k \le j$ s.t.		
$(i,k,n,S,\mathcal{M}) \models_W P_2$ and		
for all $r \in S,\ k < r \le j$,		
$(i,r,n,S,\mathcal{M}) \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$, $\lvert P_2\rvert < \lvert P\rvert$]
		$\sigma'_0 = \sigma_i$, $\sigma'_{j'} = \sigma_j$,
		$\sigma'_{\lvert\sigma'\rvert} = \sigma_n$ and $\sigma'_{k'} = \sigma_k$,
		by def. of *subinterval*()]
exists $k',\ 0 \le k' \le j'$ s.t.		
$(\sigma', k') \models P_2$ and		
for all $r', k' < r' \le j'$,		
$(\sigma', r') \models P_1$	iff	[def. of $\models$]
$(\sigma', j') \models P_1$ *since* P_2		

Previous ($P \triangleq \ominus P_1$):

$\langle i,j,n,S,\mathcal{M}\rangle \models_W \ominus P_1$	iff	[def. of $\models_W$, $\mathcal{M} = [\![\ ]\!]_\sigma$]
$i < j$ and exists $k \in S$ s.t.		
cons(S,k,j) and $(i,k,n,S,\mathcal{M}) \models_W P_1$	iff	[i.h. on $\lvert P_1\rvert < \lvert P\rvert$]
		$\sigma'_{j'} = \sigma_j$ and $\sigma'_{j'-1} = \sigma_k$,
		by def. of *subinterval*()]
$j' > 0$ and $(\sigma', j'-1) \models P_1$	iff	[def. of $\models$]
$(\sigma', j') \models \ominus P_1$		

This concludes the proof of Lemma 7.1. □

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[2] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 1-3(300):411–475, 2003.

[3] F. Aloul, I. Markov, and K. Sakallah. Force: A fast and easy-to-implement variable-ordering heuristic. In *Great Lakes Symposium on VLSI*, pages 116–119, April 2003.

[4] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Mince: A static global variable-ordering heuristic for SAT search and BDD manipulation. *Journal of Universal Computer Science*, 10(12):1562–1596, 2004.

[5] R. Alur. *Techniques for Automatic Verification of Real-time Systems*. PhD thesis, Department of Computer Science, Stanford University, 1991.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[7] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[8] R. Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems. International School on Formal Methods for the design of Computer, Communication and Software Systems, SFM-RT 2004. Revised Lectures*, number 3185 in LNCS, pages 200–236, Bertinoro, Italy, 2004. Springer.

[9] F. Balarin. Approximate reachability analysis of timed automata. In *IEEE Real-Time Systems Symposium*, pages 52–61, 1996.

[10] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 35–61. Springer-Verlag, 1985.

[11] D. Basin and S. Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[12] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods In System Design*, 13:255–288, 1998. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939.

[13] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *TACAS'00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 188–203. Springer-Verlag, 2000.

[14] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *CAV 2001*, LNCS. Springer-Verlag, 2001.

[15] G. Behrmann, J. Bengtsson, A. David, K. Larsen, P. Pettersson, and W. Yi. UPPAAL implementation secrets. In *Proceddings of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *LNCS*, pages 3–22. Springer, 2002.

[16] G. Behrmann, P. Bouyer, K.G. Larsen, and R. Pelanek. Lower and upper bounds in zone based abstractions of timed automata. In *Proceedings of TACAS'04*, LNCS 2988, pages 312–326. Springer, 2004.

[17] G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proceedings of the 11th International Conferenc in Computer Aided Verification, CAV '99*, volume 1633 of *LNCS*, pages 341–353. Springer, 1999.

[18] J. Bengtsson. Efficient symbolic state exploration of timed systems: Theory and implementation. Technical Report 2001-009, Department of Information Technology, Uppsala University, 2001.

[19] J. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conferenc in Computer Aided Verification, CAV '96*, number 1102 in LNCS, pages 244–256. Springer–Verlag, 1996.

[20] J. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52-53:163–181, July-August 2002.

[21] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, LNCS 3098. Springer, 2004.

[22] G. Berhmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems. International School on Formal Methods for the design of Computer, Communication and Software Systems, SFM-RT 2004. Revised Lectures*, LNCS 3185, pages 200–236. Springer, 2004.

[23] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for BDD-based verification of real-time systems. In W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 122–125. Springer-Verlag, 2003.

[24] D. Beyer and A. Noack. Can decision diagrams overcome state space explosion in real-time verification? In H. König, M. Heiner, and A. Wolisz, editors, *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, volume 2767 of *LNCS*, pages 193–208. Springer-Verlag, 2003.

[25] N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253:27–60, 2001.

[26] N.S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, November 1998.

[27] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[28] G.S. Blair, L. Blair, and H. Bowman et. al. *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.

[29] JP. Bodeveix and M. Filali. FMona: A tool for expressing validation techniques over infinite state systems. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 204–219. Springer-Verlag, 2000.

[30] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[31] T. Bolognesi, F. Lucidi, and S. Trigila. Converging towards a timed LOTOS standard. *Computer Standards & Interfaces*, 16:87–118, 1994.

[32] S. Bornot and J. Sifakis. On the composition of hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 1386, pages 49–63. Springer, 1998.

[33] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, LNCS 1536, pages 103–129. Springer, 1998.

[34] D. Bosnacki. Digitization of timed automata. In *Proc. of the Fourth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '99)*, pages 283–302, 1999.

[35] D. Bosnacki and D. Dams. Integrating real time into Spin: A prototype implementation. In *FORTE 1998*, IFIP Conference Proceedings, pages 423–438. Kluwer, 1998.

[36] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium, RTSS '97*, pages 25–35. IEEE Computer Society, 1997.

[37] H. Bowman. An interpretation of cognitive theory in concurrency theory. Technical Report 8-98, Computing Laboratory, University of Kent at Canterbury, October 1998.

[38] H. Bowman. Modelling timeouts without timelocks. In *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop*, LNCS 1601, pages 335–353. Springer-Verlag, 1999.

[39] H. Bowman. Time and action lock freedom properties for timed automata. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *FORTE 2001, Formal Techniques for Networked and Distributed Systems*, pages 119–134, Cheju Island, Korea, 2001. Kluwer Academic.

[40] H. Bowman, H. Cameron, P. King, and S. Thompson. Mexitl: Multimedia in Executable Interval Temporal Logic. Technical Report 3-97, Computing Laboratory, University of Kent at Canterbury, May 1997.

[41] H. Bowman, H. Cameron, P. King, and S.J. Thompson. Mexitl: Multimedia in Executable Interval Temporal Logic. *Formal Methods in System Design*, 22:5–38, January 2003.

[42] H. Bowman and G. Faconti. Analysing cognitive behaviour using LOTOS and MEXITL. *Formal Aspects of Computing*, 11:132–159, November 1999.

[43] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation protocol using UPPAAL. *Formal Aspects of Computing*, 10(5-6):550–575, August 1998.

[44] H. Bowman, G. Faconti, and M. Massink. Specification and verification of media constraints using UPPAAL. In *5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, DSV-IS 98*, Eurographics Series. Springer-Verlag, August 1998.

[45] H. Bowman and R. Gomez. *Concurrency Theory, Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer, January 2006.

[46] H. Bowman and R. Gomez. How to stop time stopping. To appear in Formal Aspects of Computing (Springer), 2006.

[47] H. Bowman, R. Gomez, and L. Su. A tool for the syntactic detection of zeno-timelocks in timed automata. *ENTCS*, 139(1):25–47, November 2005. Proceedings of the 6th AMAST Workshop on Real-time Systems (ARTS 2004).

[48] H. Bowman and S. Thompson. A Tableaux Method for Interval Temporal Logic with Projection. In *International Conference on Analytic Tableaux and Related Methods (TABLEAUX'98)*, volume 1397 of *Lecture Notes in AI*, pages 108–123. Springer-Verlag, May 1998.

[49] H. Bowman and S.J. Thompson. A decision procedure and complete axiomatisation of finite interval temporal logic with projection. *Journal of Logic and Computation*, 13(2), 2003.

[50] M. Bozga, S. Graf, Ileana Ober, Iulian Ober, and J. Sifakis. The IF toolset. In M. Bernardo and F. Corradini, editors, *SFM-RT 2004*, number 3185 in LNCS, pages 237–267, 2004.

[51] V. Braberman and A. Olivero. Extending timed automata for compositional modeling healthy timed systems. *ENTCS*, 52(3), 2001.

[52] R. Bryant. Symbolic boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[53] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[54] J.R. Büchi. On a decision method in restricted second-order arithmetic. *Zeitschrift für Mathemathische Logik and Grundlagen der Mathematik*, 6:66–92, 1960.

[55] A. Cau and B. Moszkowski. Using PVS for interval temporal logic proofs. part 1: The syntactic and semantic encoding. Technical Report 14, SERCentre, De Montfort University, Leicester, 1996.

[56] A. Cau and H. Zedan. Refining Interval Temporal Logic specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software (ARTS '97)*, volume 1231 of *LNCS*, pages 79–94. Springer-Verlag, 1997.

[57] G. Chakravorty and P. Pandya. Digitizing Interval Duration Logic. In *CAV'03*, volume 2725 of *LNCS*, pages 167–179. Springer, 2003.

[58] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs, Workshop, Yorktown Heights, New York, May 1981*, LNCS 131, pages 52–71. Springer-Verlag, 1982.

[59] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[60] A. David, G. Behrmann, K. Larsen, and W. Yi. Unification & sharing in timed automata verification. In *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 225–229. Springer, 2003.

[61] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, pages 313–329, 1998.

[62] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 73. IEEE Computer Society, 1996.

[63] W.P. de Roever. The need for compositional proof systems: A survey. In H. Langmaack W.P. de Roever and A. Pnueli, editors, *Compositionality: the significant difference*, volume 1536 of *LNCS*, pages 1–22. Springer-Verlag, 1998.

[64] Z. H. Duan. *An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming.* PhD thesis, University of Newcastle Upon Tyne, May 1996.

[65] J. Elgaard, A. Møller, and M. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *LNCS*, pages 119–134, 2000.

[66] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.

[67] U. Engberg, P. Gronning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification*, pages 44–55, 1992.

[68] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. Technical Report RT-254, INRIA, France, December 2001. Also see http://www.inrialpes.fr/vasy/cadp/.

[69] B. Gebremichael and F. Vaandrager. Specifying Urgency in Timed I/O Automata. Technical Report NIII-R0459, Radboud University Nijmegen, Netherlands, December 2004.

[70] B. Gebremichael, F. Vaandrager, and M Zhang. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using UPPAAL. Technical Report ICIS-R06XX, Radboud University, Nijmegen, The Netherlands, 2006.

[71] R. Gomez and H. Bowman. Discrete timed automata and MONA: Description, specification and verification of a multimedia stream. In H. Konig, M. Heiner, and A. Wolisz, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2003. Proceedings of the 23rd IFIP WG 6.1 International Conference, Berlin, Germany, September/October 2003*, LNCS 2767, pages 177–192. Springer, 2003.

[72] R. Gomez and H. Bowman. PITL2MONA: Implementing a decision procedure for Propositional Interval Temporal Logic. *Journal of Applied Non-Classical Logics*, 14(1):107–150, 2004.

[73] R. Gomez and H. Bowman. Discrete Timed Automata. Technical Report 3-05, Computing Laboratory, University of Kent, UK, 2005.

[74] V. Goranko, A. Montanari, and G. Sciavicco. A road map of interval temporal logics and duration calculi. *Journal of Applied Non-Classical Logics*, 14(1-2):9–54, 2004. Issue on Interval Temporal Logics and Duration Calculi.

[75] R. Hale. *Programming in temporal logic*. PhD thesis, Cambridge University, 1989.

[76] R. Hale. Using temporal logic for prototyping: The design of a lift controller. In *Proceedings of Temporal Logic in Specification 1987*, volume 398 of *LNCS*, pages 375–408. Springer, 1989.

[77] R. Hale and J. He. A real-time programming language. In *Towards Verified Systems*. Elsevier, 1994.

[78] J. Halpern, Z. Manna, and B. Moszkowski. A Hardware Semantics Based on Temporal Intervals. In J. Diaz, editor, *Proc. ICALP 1983*, volume 154 of *LNCS*, pages 278–291. Springer-Verlag, 1983.

[79] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to UPPAAL. In K. Larsen and P. Niebert, editors, *Proceedings of FORMATS 2003*, LNCS 2791, pages 46–59. Springer-Verlag, 2004.

[80] M. Hendriks and K. Larsen. Exact acceleration of real-time model checking. *ENTCS*, 65(6), 2002.

[81] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95*, volume 1019 of *LNCS*, pages 89–110. Springer-Verlag, 1995.

[82] T. Henzinger, P-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *STTT*, 1(1-2):110–122, 1997.

[83] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.

[84] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, 1994.

[85] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[86] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[87] Y. Hollander, M. Morley, and A. Noy. The *e* language: a fresh separation of concerns. In *TOOLS Europe 2001*, Zurich, Switzerland, 2001. IEEE Computer Press.

[88] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[89] D. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[90] S. Kalvala. A formulation of TLA in Isabelle. In *TPHOLs*, volume 971 of *LNCS*, pages 214–228. Springer, 1995.

[91] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrage. The theory of Timed I/O Automata. Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, April 2004.

[92] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O Automata: a mathematical framework for modelling and analyzing real-time systems. In *Proceedings 24th IEEE International Real-Time Systems Symposium (RTSS03)*, pages 166–177. IEEE Computer Society, 2003.

[93] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In *Hybrid Systems III*, LNCS 1066, pages 13–40. Springer-Verlag, 1996.

[94] N. Klarlund. Mona & fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS, 1998. LNCS 1414.

[95] N. Klarlund. A theory of restrictions for logics and automata. In *Computer Aided Verification, CAV '99*, volume 1633 of *LNCS*, 1999.

[96] N. Klarlund and A. Möller. *MONA Version 1.4 User Manual*. BRICS, University of Aarhus, Denmark, January 2001.

[97] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

[98] S. Kono. Automatic verification of Interval Temporal Logic. In *8th British Colloquium For Theoretical Computer Science*, March 1992.

[99] S. Kono. A combination of clausal and non clausal temporal logic programs. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics: Proc. of the IJCAI-93 Workshop*, pages 40–57. Springer, Berlin, Heidelberg, 1995.

[100] L. Lamport. Real time is really simple. Technical Report MSR-TR-2005-30, Microsoft Research, March 2005.

[101] L. Lamport. Real-time model checking is really simple. In *CHARME*, volume 3725 of *LNCS*, pages 162–175. Springer, 2005.

[102] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proceedings of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[103] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Compact data structure and state-space reduction for model checking real time systems. *Real-Time Systems*, 25(2):255–275, September 2003.

[104] Y. Li and D. V. Hung. Checking temporal duration properties of timed automata. *Journal of Computer Science and Technology*, 17(6):689–698, 2002.

[105] O. Lichtenstein and A. Pnueli. Propositional temporal logics: Decidability and completeness. *Logic Journal of the IGPL*, 8(1), 2000.

[106] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218. Springer-Verlag, 1985.

[107] H. Lim, D. Kaynar, N. Lynch, and S. Mitra. Translating Timed I/O Automata specifications for theorem proving in PVS. In *Proceedings of FORMATS 2005*, volume 3829 of *LNCS*, pages 17–31. Springer, 2005.

[108] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gearbox controller. *Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.

[109] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[110] Z. Manna, Y. Kesten, and A. Pnueli. Verifying clocked transition systems. In *Hybrid Systems III*, LNCS 1066, pages 13–40. Springer-Verlag, 1996.

[111] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[112] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[113] N. Markey. Temporal logic with past is exponentially more succinct. *Bulletin of the EATCS*, 79:122–128, 2003.

[114] A. R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editor, *Logic Colloquium (Proc. Symposium on Logic, Boston, 1972)*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154, 1975.

[115] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[116] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[117] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation (PLDI). SIGPLAN Notices 36(5)*, pages 221–231. ACM, 2001.

[118] F. Morawietz and T. Cornell. The MSO logic-automaton connection in linguistics. In *Logical Aspects of Computational Linguistics, Second International Conference, LACL'97*, volume 1582 of *LNCS*, pages 112–131. Springer, 1997.

[119] B. Moszkowski. A hierarchical analysis of propositional temporal logic based on intervals. To appear in Journal of Logic and Computation.

[120] B. Moszkowski. *Reasoning about digital circuits*. PhD thesis, Stanford University, 1983.

[121] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[122] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge U. Press, 1986.

[123] B. Moszkowski. Some very compositional temporal properties. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 307–326. North-Holland, 1994.

[124] B. Moszkowski. Compositional reasoning about projected and infinite time. In *ICECCS '95: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, page 238, Washington, DC, USA, 1995. IEEE Computer Society.

[125] B. Moszkowski. Compositional reasoning using interval temporal logic and tempura. In H. Langmaack W.P. de Roever and A. Pnueli, editors, *Compositionality: the significant difference*, volume 1536 of *LNCS*, pages 439–464. Springer-Verlag, 1998.

[126] B. Moszkowski. A hierarchical completeness proof for Propositional Interval Temporal Logic with finite time. *Journal of Applied Non-Classical Logics*, 14(1-2):55–104, 2004.

[127] B. Moszkowski. An Automata-Theoretic Completeness Proof for Interval Temporal Logic. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP 2000)*, volume 1853 of *LNCS*, pages 223–234, Geneva, Switzerland, July 9-15, 2000. Springer-Verlag.

[128] B. Moszkowski. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *Proceedings of Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000)*, pages 242–251, Santa Barbara, California, USA, June 26-29, 2000. IEEE Computer Society Press.

[129] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebra. In *Real-time Theory in Practice*, LNCS 600, pages 549–572. Springer-Verlag, June 1991.

[130] M. Nilsson. Analysing parameterized distributed algorithms. Master's thesis, University of Uppsala, 1999.

[131] S. Owre and H. Rueß. Integrating WS1S with PVS. In E.A. Emerson and A.P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, LNCS 1855, pages 548–551. Springer-Verlag, 2000.

[132] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, San Jose, CA, November 1995.

[133] P. Pandya. Interval Duration Logic: Expressiveness and decidability. *ENTCS*, 65(6), 2002.

[134] P. K. Pandya. *DCVALID 1.4: The user manual.* Tata Institute of Fundamental Research, Mumbay, India, September 2000. Available in `http://www.tcs.tifr.res.in/~pandya/dcvalid.html#paper`.

[135] P. K. Pandya. Specifying and deciding Quantified Discrete Duration Calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.

[136] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, Volume F-13 of NATO Advanced Summer Institutes, pages 123–144. Springer-Verlag, 1985.

[137] A. Rasmussen. Symbolic model checking using monadic second-order logic as requirement language. Master's thesis, Technical University of Denmark, 1999.

[138] T. Regan. Multimedia in temporal LOTOS: A lip synchronisation algorithm. In *PSTV XIII, 13th Protocol Spec., Testing & Verification.* North-Holland, 1993.

[139] C. Robson. Tioa and uppaal. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2004.

[140] R. Rosner and A. Pnueli. A choppy logic. In *Proceedings of the First Annual IEEE Symposium on Logics in Computer Science*, pages 306–314. IEEE, 1986.

[141] R. Rudell. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, San Jose, CA, November 1993.

[142] A. Sandholm and M.I. Schwartzbach. Distributed safety controllers for web services. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering (FASE'98)*, LNCS 1382, pages 270–284. Springer-Verlag, 1998.

[143] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *SAS*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.

[144] S. Schneider. *Concurrent and Real-time Systems, the CSP Approach.* Wiley, 2000.

[145] S. Schneider, J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. Timed CSP: Theory and practice. In *Real-Time: Theory in Practice*, LNCS 600, pages 640–675. Springer-Verlag, 1991.

[146] S. Seshia and R. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In *CAV*, volume 2725 of *LNCS*, pages 154–166. Springer, 2003.

[147] B. Sharma, P. Pandya, and S. Chakraborty. Bounded validity checking of Interval Duration Logic. In *TACAS'05*, volume 3440 of *LNCS*, pages 301–316. Springer, 2005.

[148] M.A. Smith and N. Klarlund. Verification of a sliding window protocol using IOA and MONA. In T. Bolognesi and D. Latella, editors, *Formal Methods for Distributed System Development*, pages 19–34. Kluwer Academic, 2000.

[149] W. Stallings. *Data & Computer Communications*. Prentice Hall, 6th. edition, 2000.

[150] L. Su. Verification of concurrent systems. Technical Report 10-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK, October 2003. Master thesis.

[151] J. Szwarcfiter and P. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT*, 16:192–204, 1976.

[152] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[153] P. H. Thai and D. V. Hung. Verifying linear duration constraints of timed automata. In *ICTAC*, volume 3407 of *LNCS*, pages 295–309. Springer, 2004.

[154] D. Thomas, P. Pandya, and S. Chakraborty. Scheduling clusters in model checking of real time systems. Technical Report TR-04-16, Indian Institute of Technology, Bombay, 2004.

[155] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press, Elsevier, 1990.

[156] S. Tripakis. *The analysis of timed systems in practice*. PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[157] S. Tripakis. Verifying progress in timed systems. In *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop*, LNCS 1601. Springer-Verlag, 1999.

[158] S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time. In *TACAS*, volume 1055 of *LNCS*, pages 329–348. Springer, 1996.

[159] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.

[160] T. Tsoronis. Formal specification and verification of real time systems with the timed automata based tools Kronos and Uppaal. Master's thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2001. (Available from Howard Bowman).

[161] M.Y. Vardi. Branching vs linear time: Final showdown. In T. Margaria and W. Yi, editors, *TACAS'2001, Tools and Algorithms for the Construction and Analysis of Systems, held as part of ETAPS'01*, LNCS 2031. Springer-Verlag, 2001. invited talk.

[162] C.A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification styles in formal descriptions of distributed systems. In *Protocol Specification, Testing and Verification, VIII*, pages 189–204. North-Holland, 1988.

[163] J. von Wright and T. Langbacka. Using a theorem prover for reasoning about concurrent algorithms. In *Computer-Aided Verification (CAV'92)*, volume 663 of *LNCS*, pages 56–68. Springer-Verlag, 1993.

[164] F. Wang. Efficient verification of timed automata with BDD-like data structures. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(1):77–97, 2004.

[165] F. Wang. Model-checking distributed real-time systems with states, events, and multiple fairness assumptions. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, AMAST 2004*, volume 3116 of *LNCS*, pages 553–568. Springer, 2004.

[166] P. Wolper. Temporal logic can be more expressive. *Information and Computation*, 56(1/2):72–99, 1983.

[167] P. Manolios Y. Yu and L. Lamport. Model checking TLA+ specifications. In *CHARME*, volume 1703 of *LNCS*, pages 54–66. Springer, 1999.

[168] W. Yi. CCS + time = an interleaving model for real-time systems. In *Automata, Languages and Programming 18*, LNCS 510, pages 217–228. Springer-Verlag, 1991.

[169] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[170] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, pages 114–152, London, UK, 1998. Springer-Verlag.

[171] C. Zhou and M. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS Series of Monographs in Theoretical Computer Science. Springer, 2004.

[172] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.